ESTD. 2001

PRATHYUSHA ENGINEERING COLLEGE
DEPARTMENT OF CSE
2.3.2 C - E CONTENTS DEVELOPED BY THE FACULTY

Sl.No	SUBJECT NAME	FACULTY NAME
1	CS8492/ DATABASE MANAGEMENT SYSTEM	Ms.B.GUNASUNDARI
2	CS8602/COMPILER DESIGN	Ms.K.P.REVATHI
3	CS6008/HUMAN COMPUTER INTERACTION	Ms.N.SRIPRIYA
4	GE6075/PROFESSIONAL ETHICS	Dr.S.PADMAPRIYA
5	CS8651/INTERNET PROGRAMMING	Mr.I.MOHAN

PRATHYUSHA ENGINEERING COLLEGE

Poonamallee – Tiruvallur Road, Chennai – 602025.

CS8492

Database Management Systems

(Anna University - Regulation)

Ms.B.GunaSundari

GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING **L T P C**
3 0 0 3

OBJECTIVES:

- To know the basics of algorithmic problem solving
- To read and write simple Python programs.
- To develop Python programs with conditionals and loops.
- To define Python functions and call them.
- To use Python data structures –- lists, tuples, dictionaries.
- To do input/output with files in Python.

UNIT I ALGORITHMIC PROBLEM SOLVING 9

Algorithms, building blocks of algorithms (statements, state, control flow, functions), notation (pseudo code, flow chart, programming language), algorithmic problem solving, simple strategies for developing algorithms (iteration, recursion). Illustrative problems: find minimum in a list, insert a card in a list of sorted cards, guess an integer number in a range, Towers of Hanoi.

UNIT II DATA, EXPRESSIONS, STATEMENTS 9

Python interpreter and interactive mode; values and types: int, float, boolean, string, and list; variables, expressions, statements, tuple assignment, precedence of operators, comments; modules and functions, function definition and use, flow of execution, parameters and arguments; Illustrative programs: exchange the values of two variables, circulate the values of n variables, distance between two points.

UNIT III CONTROL FLOW, FUNCTIONS 9

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: state, while, for, break, continue, pass; Fruitful functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum an array of numbers, linear search, binary search.

UNIT IV LISTS, TUPLES, DICTIONARIES 9

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters; Tuples: tuple assignment, tuple as return value; Dictionaries: operations and methods; advanced list processing - list comprehension; Illustrative programs: selection sort, insertion sort, mergesort, histogram.

UNIT V FILES, MODULES, PACKAGES 9

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count, copy file.

TOTAL : 45 PERIODS

OUTCOMES:

Upon completion of the course, students will be able to

- Develop algorithmic solutions to simple computational problems
- Read, write, execute by hand simple Python programs.
- Structure simple Python programs for solving problems.
- Identify proper conditionals or iterative statement for problems.
- Decompose a Python program into functions.
- Apply python string functions.
- Represent compound data using Python lists, tuples, dictionaries.
- Read and write data from/to files in Python Programs.
- Identify and handle errors and exceptions in Python Programs.

TEXT BOOKS:

1. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist'', 2nd edition, Updated for Python 3, Shroff/O'Reilly Publishers, 2016 (http://greenteapress.com/wp/think- python/)

2. Guido van Rossum and Fred L. Drake Jr, —An Introduction to Python – Revised and updated for Python 3.2, Network Theory Ltd., 2011.

REFERENCES:

1. John V Guttag, —Introduction to Computation and Programming Using Python'', Revised and expanded Edition, MIT Press , 2013

2. Robert Sedgewick, Kevin Wayne, Robert Dondero, —Introduction to Programming in Python: An Inter-disciplinary Approach, Pearson India Education Services Pvt. Ltd., 2016.

3. Timothy A. Budd, —Exploring Python‖, Mc-Graw Hill Education (India) Private Ltd.,, 2015.

4. Kenneth A. Lambert, —Fundamentals of Python: First Programs‖, CENGAGE Learning, 2012.

5. Charles Dierbach, —Introduction to Computer Science using Python: A Computational Problem-Solving Focus, Wiley India Edition, 2013.

6. Paul Gries, Jennifer Campbell and Jason Montojo, —Practical Programming: An Introduction to Computer Science using Python 3‖, Second edition, Pragmatic Programmers, LLC, 2013.

CS8492 DATABASE MANAGEMENT SYSTEMS

UNIT I RELATIONAL DATABASES

Purpose of Database System – Views of data – Data Models – Database System Architecture – Introduction to relational databases – Relational Model – Keys – Relational Algebra – SQL fundamentals – Advanced SQL features – Embedded SQL– Dynamic SQL.

1. What is DBMS? What are the applications of database systems?

- A **database-management system (DBMS)** is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

- Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

DATABASE-SYSTEM APPLICATIONS:

Databases are widely used. Here are some representative applications:

1) Enterprise Information:

Sales: For customer, product, and purchase information.

Accounting: For payments, receipts, account balances, assets and other accounting information.

Human resources: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.

Manufacturing: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.

Online retailers: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.

2) Banking and Finance:

Banking: For customer information, accounts, loans, and banking transactions.

Credit card transactions: For purchases on credit cards and generation of monthly statements.

Finance: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.

3) Universities: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).

4) Airlines: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.

5) Telecommunication: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

2. **What are the purposes of database systems?**

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

A **database** management **system** is a software tool that makes it possible to organize data in a **database**. It is often referred to by its acronym, DBMS. The functions of a DBMS include concurrency, security, backup and recovery, integrity and data descriptions.

File-processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) were introduced, organizations usually stored information in such systems.

ADVANTAGES OF DATABASE SYSTEMS :

1) **Reducing Data Redundancy**

The file based data management systems contained multiple files that were stored in many different locations in a system or even across multiple systems. Because of this, there were sometimes multiple copies of the same file which lead to data redundancy.

This is prevented in a database as there is a single database and any change in it is reflected immediately. Because of this, there is no chance of encountering duplicate data.

2) **Sharing of Data**

In a database, the users of the database can share the data among themselves. There are various levels of authorisation to access the data, and consequently the data can only be shared based on the correct authorisation protocols being followed.

Many remote users can also access the database simultaneously and share the data between themselves.

3) **Data Integrity**

Data integrity means that the data is accurate and consistent in the database. Data Integrity is very important as there are multiple databases in a DBMS. All of these databases contain data that is visible to multiple users. So it is necessary to ensure that the data is correct and consistent in all the databases and for all the users.

4) **Data Security**

Data Security is vital concept in a database. Only authorised users should be allowed to access the database and their identity should be authenticated using a username and password. Unauthorised users should not be allowed to access the database under any circumstances as it violates the integrity constraints.

5) **Privacy**

The privacy rule in a database means only the authorized users can access a database according to its privacy constraints. There are levels of database access and a user can only view the data he is allowed to. For example - In social networking sites, access constraints are different for different accounts a user may want to access.

6) **Backup and Recovery**

Database Management System automatically takes care of backup and recovery. The users don't need to backup data periodically because this is taken care of by the DBMS. Moreover, it also restores the database after a crash or system failure to its previous condition.

7) **Data Consistency**

Data consistency is ensured in a database because there is no data redundancy. All data appears consistently across the database and the data is same for all the users viewing the database. Moreover, any changes made to the database are immediately reflected to all the users and there is no data inconsistency.

Disadvantages of file-processing system:

1) Data redundancy and inconsistency.

Since different programmers create the files and application programs over a long period, the various files are likely to have different structures and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree. For example, a changed student address may be reflected in the Music department records but not elsewhere in the system.

2) Difficulty in accessing data.

Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of all students. The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk has the preceding two options, neither of which is satisfactory.

Conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

3) **Data isolation**. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

4) **Integrity problems**. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application pro-grams. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

5) **Atomicity problems**.

A computer system is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $500 from the account balance of department A to the account balance of department B. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of department A but was not credited to the balance of department B, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be atomic — it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

6) Concurrent-access anomalies.
For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data.

7) Security problems.
Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.
These difficulties prompted the development of database systems.

3. Describe about the various view of data.

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

Physical level. The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.

Logical level. The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

View level. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

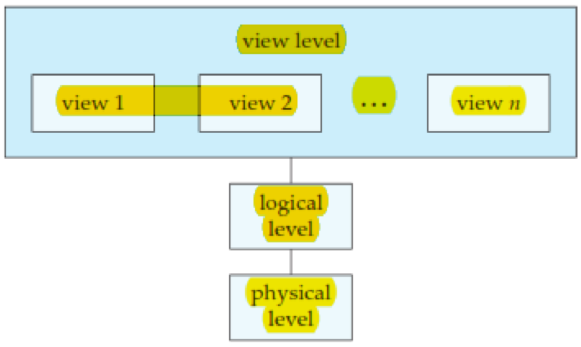


Figure . The three levels of data abstraction.

An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming languages support the notion of a structured type. For example, we may describe a record as follows:[1]

type instructor = **record**
ID : **char** (5);
name : **char** (20);
dept name : **char** (20);
salary : **numeric** (8,2);
end;

This code defines a new record type called instructor with four fields. Each field has a name and a type associated with it. A university organization may have several such record types, including

department, with fields dept name, building, and budget
course, with fields course id, title, dept name, and credits
student, with fields ID, name, dept name, and tot cred

At the physical level, an instructor, department, or student record can be described as a block of consecutive storage locations. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

Instances and Schemas:

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

Database systems have several schemas, partitioned according to the levels of abstraction.

- The **physical schema** describes the database design at the physical level.
- The **logical schema** describes the database design at the logical level.
- A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

4. Explain about various data models in details.

Data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

The data models can be classified into four different categories:

1) **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.
2) **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called entities, and relationships among these objects. An entity is a “thing” or “object” in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.

3) **Object-Based Data Model**. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity.

The object-relational data model combines features of the object-oriented data model and relational data model.

4) **Semistructured Data Model**. The semistructured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **Extensible Markup Language (XML)** is widely used to represent semistructured data.
5) The **network data model** and the **hierarchical data model** preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are used little now, except in old database code that is still in service in some places. They are outlined online in Appendices D and E for interested readers.

5. With the help of a neat block diagram , explain the basic architecture of a data base management system?

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. A gigabyte is approximately 1000 megabytes (actually 1024) (1 billion bytes), and a terabyte is 1 million megabytes (1 trillion bytes). Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed. Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory.

The query processor is important because it helps the database system to simplify and facilitate access to data. The query processor allows database users to obtain good performance while being able to work at the view level and not be burdened with understanding the physical-level details of the implementation of the system. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

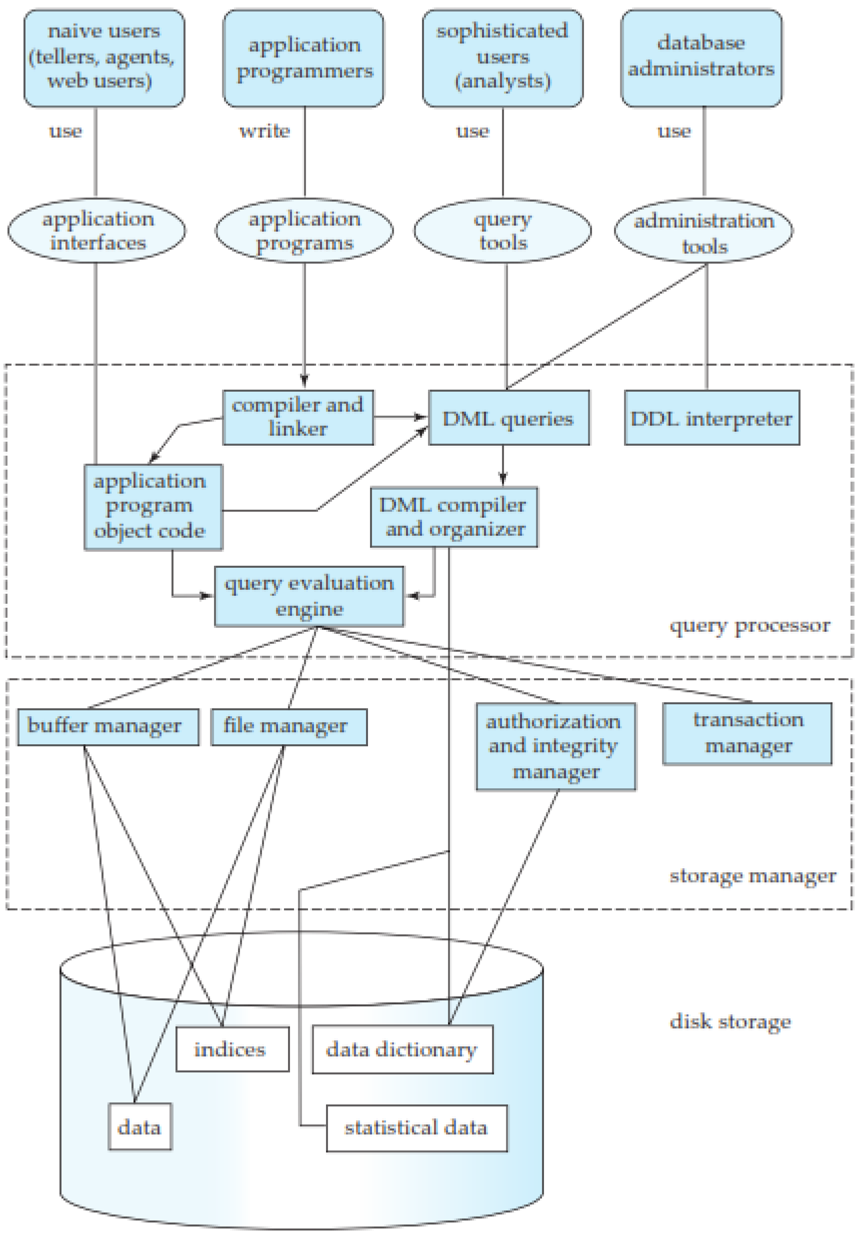


Figure 1.5 System structure.

Storage Manager:

The storage manager is the component of a database system that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system provided by the operating system. The storage man-ager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction manager**, which ensures that the database remains in a consistent) state despite system failures, and that concurrent transaction executions proceed without conflicting.

- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements several data structures as part of the physical system implementation:

- **Data files**, which store the database itself.
- **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices**, which can provide fast access to data items. A database index provides pointers to those data items that hold a particular value. For example, we could use an index to find the instructor record with a particular ID, or all instructor records with a particular name. Hashing is an alternative to indexing that is faster in some but not all cases.

The Query Processor:

The query processor components include:

- **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler**, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**; that is, it picks the lowest cost evaluation plan from among the alternatives.

- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

Database Architecture:

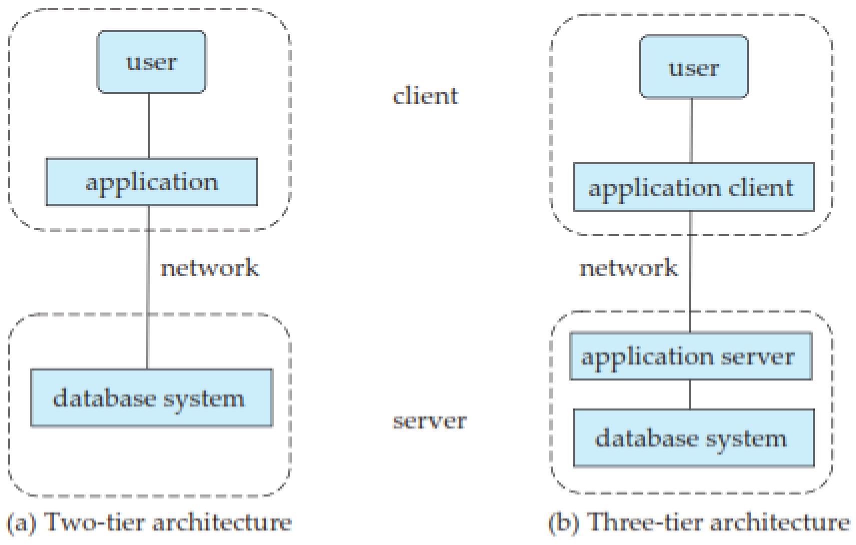


Figure . Two-tier and three-tier architectures.

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between **client** machines, on which remote database users work, and **server** machines, on which the database system runs.
Database applications are usually partitioned into two or three parts. In a **two-tier architecture**, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

In contrast, in a **three-tier architecture**, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an **application server**, usually through a forms interface. The application server in turn communicates with a database system to access data. The **business logic** of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.

Database Users and Administrators:A primary goal of a database system is to retrieve information from and store new information into the database. People who work with a database can be categorized as database users or database administrators.

Database Users and User Interfaces:

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

1) **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a clerk in the university who needs to add a new instructor to department A invokes a program called new hire. This program asks the clerk for the name of the new instructor, her new ID, the name of the department (that is, A), and the salary.

The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read reports generated from the database.
As another example, consider a student, who during class registration period, wishes to register for a class by using a Web interface. Such a user connects to a Web application program that runs at a Web server. The application first verifies the identity of the user, and allows her to access a form where she enters the desired information. The form information is sent back to the Web application at the server, which then determines if there is room in the class (by retrieving information from the database) and if so adds the student information to the class roster in the database.

2) **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that en-able an application programmer to construct forms and reports with minimal programming effort.

3) **Sophisticated users** interact with the system without writing programs. In-stead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.

4) **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided

design systems, knowledge-base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems. Chapter 22 covers several of these applications.

5) Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator (DBA)**. The functions of a DBA include:

- **Schema definition**. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- **Storage structure and access-method definition**.
- **Schema and physical-organization modification**. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- **Granting of authorization for data access**. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- **Routine maintenance**. Examples of the database administrator’s routine maintenance activities are:

✓ Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
✓ Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
✓ Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

6. Explain about relational model.

- The relational model is today the primary data model for commercial data-processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model.

Structure of Relational Databases:

✓ A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example:

1) Consider the *instructor* table stores information about instructors. The table has four column headers: *ID*, *name*, *dept name*, and *salary*.
2) The *course* table stores information about courses, consisting of a *course id*, *title*, *dept name*, and *credits*, for each course.Each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course id*.

3) The third table, *prereq*, stores the prerequisite courses for each course. The table has two columns, *course id* and *prereq id*.

- A row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between *n* values is represented mathematically by an *n-tuple* of values, i.e., a tuple with *n* values, which corresponds to a row in a table.

ID	*Name*	*dept_name*	*Salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Raghu	Physics	87000
45565	Manasa	Comp. Sci.	75000
76543	Singh	Finance	80000
98345	Ravi	Elec. Eng.	80000

Figure . The *instructor* relation.

course_id	*Title*	*dept_name*	*Credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4

Figure . The *course* relation

Course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101

Figure . The *prereq* relation.

- In the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

- We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows. The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order or are unsorted does not matter;
- For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.
- For all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, suppose the table *instructor* had an attribute *phone number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in the set.

- The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone number* would have an atomic domain.
- The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible. We shall assume null values are absent initially.

Database Schema:

- ✓ **Database schema** is the logical design of the database, and the **database instance** is a snapshot of the data in the database at a given instant in time.
- ✓ The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.
- ✓ In general, a relation schema consists of a list of attributes and their corresponding domains.
- ✓ The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time;

Schema Diagrams:

- A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure shows the schema diagram for our university organization. Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

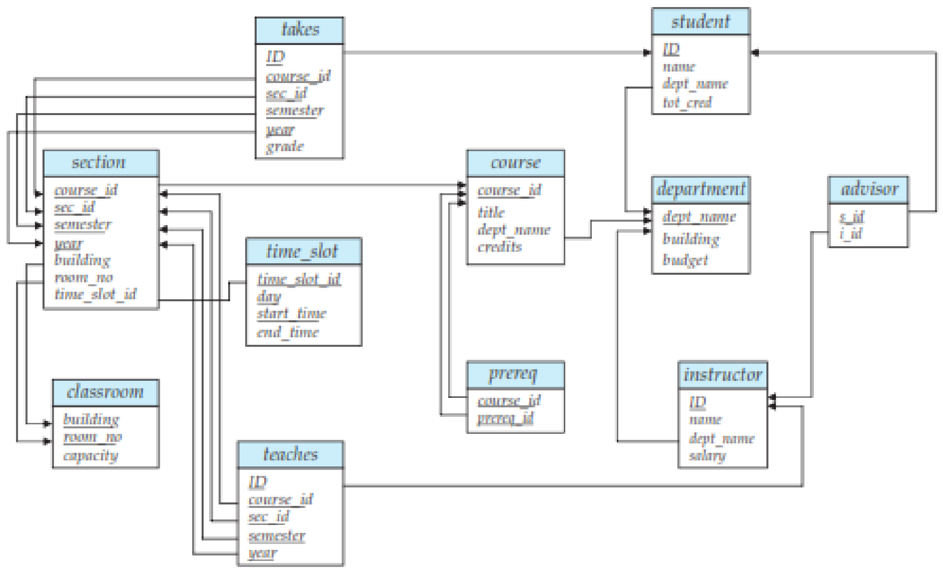


Figure. Schema diagram for the university database.

Relational Query Languages:

A **query language** is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language. Query languages can be categorized as either procedural or nonprocedural.

1) In a **procedural language**, the user instructs the system to perform a sequence of operations on the database to compute the desired result.
2) In a **nonprocedural language**, the user describes the desired information without giving a specific procedure for obtaining that information.

classroom(*building*, *room number*, *capacity*)
department(*dept name*, *building*, *budget*)
course(*course id*, *title*, *dept name*, *credits*)
instructor(*ID*, *name*, *dept name*, *salary*)
section(*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*)
teaches(*ID*, *course id*, *sec id*, *semester*, *year*)
student(*ID*, *name*, *dept name*, *tot cred*)
takes(*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
advisor(*s ID*, *i ID*)
time slot(*time slot id*, *day*, *start time*, *end time*)
prereq(*course id*, *prereq id*)

Figure. Schema of the university database.

- Query languages used in practice include elements of both the procedural and the nonprocedural approaches.
- There are a number of "pure" query languages: The relational algebra is procedural, whereas the tuple relational calculus and domain relational calculus are nonprocedural. These query languages are terse and formal, lacking the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

Relational Operations:

- All procedural relational query languages provide a set of operations that can be applied to either a single relation or a pair of relations. These operations have the nice and desired property that their result is always a single relation. This property allows one to combine several of these operations in a modular way. Specifically, since the result of a relational query is itself a relation, relational operations can be applied to the results of queries as well as to the given set of relations.

1) The most frequent operation is the selection of specific tuples from a single relation (say *instructor*) that satisfies some particular predicate (say *salary* > \$85,000). The result is a new relation that is a subset of the original relation (*instructor*).
2) Another frequent operation is to select certain attributes (columns) from a relation. The result is a new relation having only those selected attributes.

3) The *join* operation allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple. There are a number of different ways to join relations .
4) The *Cartesian product* operation combines tuples from two relations, but unlike the join operation, its result contains *all* pairs of tuples from the two relations, regardless of whether their attribute values match.

Because relations are sets, we can perform normal set operations on relations.

5) The *union* operation performs a set union of two “similarly structured” tables (say a table of all graduate students and a table of all undergraduate students). For example, one can obtain the set of all students in a department. Other set operations, such as *intersection* and *set difference* can be performed as well.

KEYS:

- The values of the attribute of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

1) super key:

- A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name.
- A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*.

2) candidate key:

- A **candidate key** is a 'minimal' **super key** meaning the smallest subset of **superkey** attribute which is unique.
- It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *instructor* relation. Then, both *{ID}* and *{name, dept name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, *{ID, name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

3) Primary key:

- There can be more than one candidate key in a relation out of which one can be chosen as primary key.
- **The primary key** denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.
- A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled.
- Primary keys must be chosen with care.
- The primary key should be chosen such that its attribute values are never, or very rarely, changed.
- It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

4) Foreign key:

- A relation, say r_1, may include among its attributes the primary key of an-other relation, say r_2. This attribute is called a **foreign key** from r_1, referencing r_2.
- The relation r_1 is also called the **referencing relation** of the foreign key dependency, and r_2 is called the **referenced relation** of the foreign key.

7. **List the operations of relational algebra and the purpose of each with example.**

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

TYPES OF RELATIONAL OPERATIONS:

1. Select Operation:

- The select operation selects tuples that satisfy a given predicate.
- It is denoted by sigma (σ).

1. Notation: $\sigma_{p}(r)$

Where:

σ is used for selection prediction
r is used for relation
p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like =, ≠, ≥, <, >, ≤.

For example: LOAN Relation

BRANCH_NAME	**LOAN_NO**	**AMOUNT**
Perryride	L-15	1500
Downtown	L-14	1500
Roundhill	L-11	900
Perryride	L-16	1300

Input:

1. $\sigma_{\text{BRANCH\_NAME="perryride"}}$ (LOAN)

Output:

BRANCH_NAME	**LOAN_NO**	**AMOUNT**
Perryride	L-15	1500
Perryride	L-16	1300

2. Project Operation:

- This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.
- It is denoted by ∏.

1. Notation: $\prod_{A1,\ A2,\ An}$ (r)

Where

A1, **A2**, **A3** is used as an attribute name of relation **r**.

Example: CUSTOMER RELATION

NAME	**STREET**	**CITY**
Jones	Main	Harrison
Smith	North	Rye

Input:

1. $\prod_{\text{NAME, CITY}}$ (CUSTOMER)

Output:

NAME	**CITY**
Jones	Harrison

Smith	Rye

3. Union Operation:

- Suppose there are two tuples R and S. The union operation contains all the tuples that are either in R or S or both in R & S.
- It eliminates the duplicate tuples. It is denoted by ∪.

1. Notation: R ∪ S

A union operation must hold the following condition:

- R and S must have the attribute of the same number.
- Duplicate tuples are eliminated automatically.

Example:

DEPOSITOR RELATION

CUSTOMER_NAME	**ACCOUNT_NO**
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

BORROW RELATION

CUSTOMER_NAME	**LOAN_NO**
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17

Input:

1. $\prod_{\text{CUSTOMER\_NAME}}$ (BORROW) ∪ $\prod_{\text{CUSTOMER\_NAME}}$ (DEPOSITOR)

Output:

CUSTOMER_NAME
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Mayes

4. Set Intersection:

- Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.
- It is denoted by intersection ∩.

1. Notation: R ∩ S

Example: Using the above DEPOSITOR table and BORROW table

Input:

1. ∏ CUSTOMER_NAME (BORROW) ∩ ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Smith
Jones

5. Set Difference:

- Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in R but not in S.
- It is denoted by intersection minus (-).

1. Notation: R - S

Example: Using the above DEPOSITOR table and BORROW table

Input:

1. ∏ CUSTOMER_NAME (BORROW) - ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Jackson
Hayes
Willians
Curry

6. Cartesian product

- The Cartesian product is used to combine each row in one table with each row in the other table. It is also known as a cross product.
- It is denoted by X.

1. Notation: E X D

Example:

EMPLOYEE

EMP_ID	**EMP_NAME**	**EMP_DEPT**
1	Smith	A
2	Harry	C
3	John	B

DEPARTMENT

DEPT_NO	**DEPT_NAME**
A	Marketing
B	Sales
C	Legal

Input:

1. EMPLOYEE X DEPARTMENT

Output:

EMP_ID	**EMP_NAME**	**EMP_DEPT**	**DEPT_NO**	**DEPT_NAME**
1	Smith	A	A	Marketing
1	Smith	A	B	Sales

1	Smith	A	C	Legal
2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal
3	John	B	A	Marketing
3	John	B	B	Sales
3	John	B	C	Legal

7. Rename Operation:

The rename operation is used to rename the output relation. It is denoted by **rho** (ρ).

Example: We can use the rename operator to rename STUDENT relation to STUDENT1.

1. ρ(STUDENT1, STUDENT)

Join Operations:

A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied. It is denoted by ⋈.

Example:

EMPLOYEE

EMP_CODE	EMP_NAME
101	Stephan
102	Jack
103	Harry

SALARY

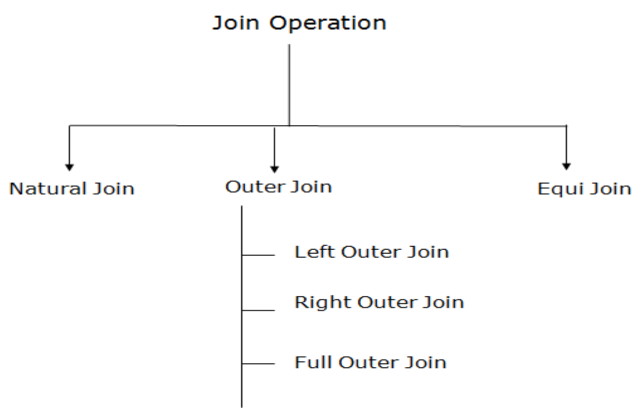
EMP_CODE	SALARY
101	50000
102	30000
103	25000

1. Operation: (EMPLOYEE ⋈ SALARY)

Result:

EMP_CODE	EMP_NAME	SALARY
101	Stephan	50000
102	Jack	30000
103	Harry	25000

8. Types of Join operations:



1) Natural Join:

- A natural join is the set of tuples of all combinations in R and S that are equal on their common attribute names.
- It is denoted by ⋈.

Example: Let's use the above EMPLOYEE table and SALARY table:

Input:

1. $\prod_{\text{EMP\_NAME, SALARY}}$ (EMPLOYEE ⋈ SALARY)

Output:

EMP_NAME	**SALARY**
Stephan	50000
Jack	30000
Harry	25000

2) **Outer Join:**

The outer join operation is an extension of the join operation. It is used to deal with missing information.

Example:

EMPLOYEE

EMP_NAME	**STREET**	**CITY**
Ram	Civil line	Mumbai
Shyam	Park street	Kolkata
Ravi	M.G. Street	Delhi
Hari	Nehru nagar	Hyderabad

FACT_WORKERS

EMP_NAME	**BRANCH**	**SALARY**
Ram	Infosys	10000
Shyam	Wipro	20000
Kuber	HCL	30000
Hari	TCS	50000

Input:

1. (EMPLOYEE ⋈ FACT_WORKERS)

Output:

EMP_NAME	**STREET**	**CITY**	**BRANCH**	**SALARY**
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru nagar	Hyderabad	TCS	50000

An outer join is basically of three types:

a. Left outer join
b. Right outer join
c. Full outer join

a. Left outer join:

- Left outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.
- In the left outer join, tuples in R have no matching tuples in S.
- It is denoted by ⟕.

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

1. EMPLOYEE ⟕ FACT_WORKERS

EMP_NAME	**STREET**	**CITY**	**BRANCH**	**SALARY**
Ram	Civil line	Mumbai	Infosys	10000

Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru street	Hyderabad	TCS	50000
Ravi	M.G. Street	Delhi	NULL	NULL

b. Right outer join:

- Right outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.
- In right outer join, tuples in S have no matching tuples in R.
- It is denoted by ⟖.

Example: Using the above EMPLOYEE table and FACT_WORKERS Relation

Input:

1. EMPLOYEE ⟖ FACT_WORKERS

Output:

EMP_NAME	BRANCH	SALARY	STREET	CITY
Ram	Infosys	10000	Civil line	Mumbai
Shyam	Wipro	20000	Park street	Kolkata
Hari	TCS	50000	Nehru street	Hyderabad
Kuber	HCL	30000	NULL	NULL

c. Full outer join:

- Full outer join is like a left or right join except that it contains all rows from both tables.
- In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.
- It is denoted by ⟗.

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

1. EMPLOYEE ⟗ FACT_WORKERS

Output:

EMP_NAME	STREET	CITY	BRANCH	SALARY
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru street	Hyderabad	TCS	50000
Ravi	M.G. Street	Delhi	NULL	NULL
Kuber	NULL	NULL	HCL	30000

3) Equi join:

It is also known as an inner join. It is the most common join. It is based on matched data as per the equality condition. The equi join uses the comparison operator(=).

Example:

CUSTOMER RELATION

CLASS_ID	NAME
1	John
2	Harry
3	Jackson

PRODUCT

PRODUCT_ID	CITY
1	Delhi
2	Mumbai
3	Noida

Input:
CUSTOMER ⋈ PRODUCT
Output:

CLASS_ID	NAME	PRODUCT_ID	CITY
1	John	1	Delhi
2	Harry	2	Mumbai
3	Harry	3	Noida

8. Describe the six clauses in the syntax of an sql query and show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?

The basic categories of commands used in SQL to perform various functions. These functions include building database objects, manipulating objects, populating database tables with data, updating existing data in tables, deleting data, performing database queries, controlling database access, and overall database administration.

The main categories are

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DQL (Data Query Language)
- DCL (Data Control Language)
- Data administration commands
- Transactional control commands

i) DDL (Data Definition Language)

Data Definition Language, *DDL*, is the part of SQL that allows a database user to create and restructure database objects, such as the creation or the deletion of a table.

Some of the most fundamental DDL commands discussed during following hours include the following:

CREATE TABLE:

```
  CREATE TABLE CUSTOMERS( ID   INT NOT NULL, NAME VARCHAR (20) NOT NULL,    AGE
INT NOT NULL,ADDRESS  CHAR (25) ,SALARY   DECIMAL (18, 2),
  PRIMARY KEY (ID));
```

ALTER TABLE:

```
ALTER TABLE Customers  ADD Email varchar(255);
ALTER TABLE Customers DROP COLUMN Email;
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

DROP TABLE

```
DROP TABLE employee;
```

CREATE INDEX

```
CREATE INDEX idx_lastname  ON Persons (LastName);
```

ALTER INDEX

```
ALTER INDEX <index name> ON <table name> (<column(s)>);
```

DROP INDEX

```
DROP INDEX index_name;
```

CREATE VIEW

```
CREATE VIEW  product11 AS  SELECT ProductName, Price FROM Products WHERE
  Price > (SELECT AVG(Price) FROM Products);
```

DROP VIEW

```
DROP VIEW emp_view;
```

ii) DML (Data Manipulation Language):

Data Manipulation Language, DML, is the part of SQL used to manipulate data within objects of a relational database.

There are three basic DML commands:

INSERT:

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('alex', 'chennai', 'india');
```

UPDATE:

```
UPDATE Customers
SET ContactName = 'Alex', City= 'bangalore'
WHERE CustomerID = 1;
```

DELETE:

```
DELETE FROM Customers WHERE CustomerName='Alex';
```

iii) DQL (Data Query Language)

Though comprised of only one command, Data Query Language (DQL) is the most concentrated focus of SQL for modern relational database users. The base command is as follows:

SELECT:

```
SELECT * FROM table_name;
SELECT CustomerName, City FROM Customers;
```

This command, accompanied by many options and clauses, is used to compose queries against a relational database. Queries, from simple to complex, from vague to specific, can be easily created.

A *query* is an inquiry to the database for information. A query is usually issued to the database through an application interface or via a command line prompt.

iv) Data Control Language(DCL)

Data control commands in SQL allow you to control access to data within the database. These DCL commands are normally used to create objects related to user access and also control the distribution of privileges among users. Some data control commands are as follows:

GRANT:

```
CREATE USER books_admin IDENTIFIED BY MyPassword;
GRANT   SELECT,  INSERT,  UPDATE,  DELETE ON books TO books_admin;
```

REVOKE:

```
GRANT   SELECT,  INSERT,  UPDATE,  DELETE ON books from books_admin;
```

CREATE SYNONYM:

```
CREATE SYNONYM offices  FOR locations;
SELECT * FROM locations;
```

v) Data Administration Commands

Data administration commands allow the user to perform audits and perform analyses on operations within the database. They can also be used to help analyze system performance. Two general data administration commands are as follows:

START AUDIT
STOP AUDIT

Do not get data administration confused with database administration. *Database administration* is the overall administration of a database, which envelops the use of all levels of commands. *Database administration* is much more specific to each SQL implementation than are those core commands of the SQL language.

vi) Transactional Control Commands

In addition to the previously introduced categories of commands, there are commands that allow the user to manage database transactions.

- COMMIT Saves database transactions
- ROLLBACK Undoes database transactions
- SAVEPOINT Creates points within groups of transactions in which to ROLLBACK
- SET TRANSACTION Places a name on a transaction

9. Explain about nested sub queries.

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
- **Example Query**
- Find courses offered in Fall 2009 and in Spring 2010
  - **select distinct** course_id **from** section **where** semester = 'Fall' **and** year= 2009 **and** course_id **in** (**select** course_id **from** section **where** semester = 'Spring' **and** year= 2010);

Find courses offered in Fall 2009 but not in Spring 2010

- **select distinct** course_id **from** section **where** semester = 'Fall' **and** year= 2009 **and** course_id **not in** (**select** course_id **from** section **where** semester = 'Spring' **and** year= 2010);

Example Query:

- Find the total number of (distinct) studentswho have taken course sections taught by the instructor with ID 10101
  - **select count** (**distinct** ID) **from** takes **where** (course_id, sec_id, semester, year) **in** (**select** course_id, sec_id, semester, year **from** teaches **where** teaches.ID= 10101);
- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison:

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.
  select distinct T.name **from** instructor **as** T, instructor **as** S **where** T.salary > S.salary **and** S.dept_name = 'Biology';
- Same query using > **some** clause
  select name **from** instructor**where** salary > **some** (**select** salary **rom** instructor **where** dept_name = 'Biology');

- **Example Query**
  - Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.
    select name from instructor where salary > all (select salary from instructor where dept_name = 'Biology');

Test for Empty Relations

- The exists construct returns the value true if the argument subquery is nonempty.
- Exists r $\Leftrightarrow$ $r \neq \emptyset$
- not exists r $\Leftrightarrow$ $r = \emptyset$
  - Correlation Variables
- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"
  select course_id from section as S where semester = 'Fall' and year= 2009 and exists (select * from section as T where semester = 'Spring' and year= 2010 and S.course_id= T.course_id);
- Correlated subquery
- Correlation name or correlation variable
- **Not Exists**
- Find all students who have taken all courses offered in the Biology department.
  select distinct S.ID, S.name **from** student **as** S**where not exists** ((**select** course_id **from** course **where** dept_name = 'Biology') **except** (**select** T.course_id **from** takes **as** T **where** S.ID = T.ID));
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
  - (Evaluates to "true" on an empty set)
- Find all courses that were offered at most once in 2009

select T.course_id **from** course **as** T **where unique** (**select** R.course_id **from** section **as** R **where** T.course_id= R.course_id
and R.year = 2009);

<u>Subqueries in the From Clause:</u>

- SQL allows a subquery expression to be used in the **from** clause Find the average instructors' salaries of those departments where the average salary is greater than $42,000.
    - **select** dept_name, avg_salary **from** (**select** dept_name, **avg** (salary) **as** avg_salary **from** instructor **group by** dept_name) **where** avg_salary > 42000;
- Note that we do not need to use the **having** clause
- Another way to write above query
  select dept_name, avg_salary **from** (**select** dept_name, **avg** (salary) **from** instructor **group by** dept_name)
  as dept_avg (dept_name, avg_salary) **where** avg_salary > 42000;
- And yet another way to write it: **lateral** clause
  select name, salary, avg_salary **from** instructor I1, **lateral** (**select avg**(salary) as avg_salary **from** instructor I2 **where** I2.dept_name= I1.dept_name);
- Lateral clause permits later part of the **from** clause (after the lateral keyword) to access correlation variables from the earlier part.
- Note: lateral is part of the SQL standard, but is not supported on many database systems; some databases such as SQL Server offer alternative syntax

<u>With Clause:</u>

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget
    - **with** max_budget (value) **as** (**select max**(budget) **from** department) **select** budget **from** department, max_budget **where** department.budget = max_budget.value;

<u>Complex Queries using With Clause:</u>

- With clause is very useful for writing complex queries
- Supported by most database systems, with minor syntax variations
- Find all departments where the total salary is greater than the average of the total salary at all departments

with dept _total (dept_name, value) **as** (**select** dept_name, **sum**(salary) **from** instructor **group by** dept_name), dept_total_avg(value) **as** (**select avg**(value) **from** dept_total)
select dept_name **from** dept_total, dept_total_avg **where**
dept_total.value >= dept_total_avg.value;

<u>Scalar Subquery:</u>

- Scalar subquery is one which is used where a single value is expected
    - E.g. **select** dept_name, (**select count**(*) **from** instructor **where** department.dept_name = instructor.dept_name) **as** num_instructors **from** department;

- E.g. **select** name **from** instructor **where** salary * 10 >
    - (**select** budget **from** department **where** department.dept_name = instructor.dept_name)
- Runtime error if subquery returns more than one result tuple

<u>Modification of the Database:</u>

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating values in some tuples in a given relation

▪ **Modification of the Database – Deletion**

- Delete all instructors **delete from** instructor
- Delete all instructors from the Finance department **delete from** instructor **where** dept_name= 'Finance';
  Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.

delete from instructor **where** dept_name **in** (**select** dept_name **from** department **where** building = 'Watson');

- Delete all instructors whose salary is less than the average salary of instructors
    - **delete from** instructor **where** salary< (**select avg** (salary) **from** instructor);
- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
    - First, compute **avg** salary and find all tuples to delete
    - Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

▪ **Modification of the Database – Insertion**

- Add a new tuple to course **insert into** course **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
- or equivalently **insert into** course (course_id, title, dept_name, credits) **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
- Add a new tuple to student with tot_creds set to null **insert into** student **values** ('3003', 'Green', 'Finance', null);
- Add all instructors to the student relation with tot_creds set to 0
    - **insert into** student **select** ID, name, dept_name, 0 **from** instructor

▪ The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like **insert into** table1 **select** * **from** table1would cause problems, if table1 did not have any primary key defined.

o **Modification of the Database – Updates**

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise

Write two **update** statements:

update instructor **set** salary = salary * 1.03 **where** salary > 100000;

update instructor **set** salary = salary * 1.05 **where** salary <= 100000;

- The order is important
- Can be done better using the **case** statement

Case Statement for Conditional Updates:

- Same query as before but with case statement

update instructor **set** salary = **case when** salary <= 100000 **then** salary * 1.05 **else** salary * 1.03 **end**

Updates with Scalar Subqueries:

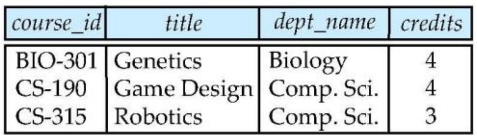
- Recompute and update tot_creds value for all students
  update student S **set** tot_cred = (**select sum**(credits) **from** takes **natural join** course **where** S.ID= takes.ID **and** takes.grade <> ’F’ **and** takes.grade **is not null**);
- Sets tot_creds to null for students who have not taken any course
  - Instead of **sum**(credits), use: **case when sum**(credits) **is not null then sum**(credits) **else** 0 **end**

10. Explain about join operations in SQL.

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause

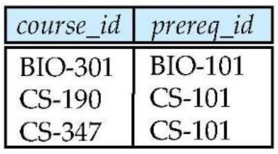
Join operations – Example

- Relation *course*



course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*



course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

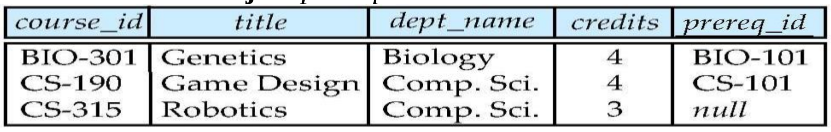
- Observe that prereq information is missing for CS-315 and course information is missing for CS-437

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.

Left Outer Join

□□*course* **natural left outer join** *prereq*



course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join

□□*course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on < predicate>
using $(A_1, A_1, \dots, A_n)$

Full Outer Join

course **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Joined Relations – Examples

- *course* **inner join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above, and a natural join?
- *course* **left outer join** *prereq* **on** *course.course_id = prereq.course_id*

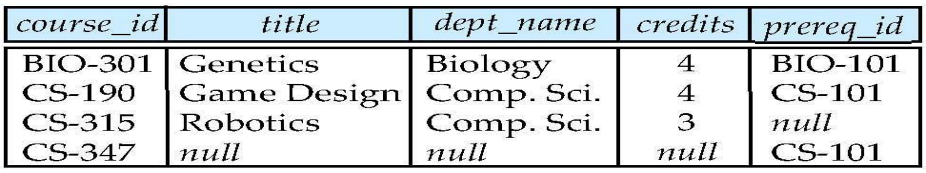
course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Joined Relations – Examples

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- *course* **full outer join** *prereq* **using** (*course_id*)



course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

11. Discuss about view in SQL.

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables, or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation.

```
SQL> CREATE TABLE EMPLOYEE (

                                    EMPLOYEE_NAME   VARCHAR2(10),
                                    EMPLOYEE_NO     NUMBER(8),
                                    DEPT_NAME       VARCHAR2(10),

                                    DEPT_NO         NUMBER (5),
                                    DATE_OF_JOIN    DATE
                                );

Table created.
```

CREATE VIEW

SUNTAX FOR CREATION OF VIEW

CREATE [OR REPLACE] [FORCE] VIEW viewname [(column-name, column-name)] AS Query [with check option];
Include all not null attribute.

CREATION OF VIEW

```
SQL> CREATE VIEW EMPVIEW AS SELECT EMPLOYEE_NAME,

EMPLOYEE_NO,
DEPT_NAME,
DEPT_NO,
DATE_OF_JOIN FROM EMPLOYEE;
View Created.
```

DISPLAY VIEW:

```
SQL> SELECT * FROM EMPVIEW;
EMPLOYEE_N          EMPLOYEE_NO           DEPT_NAME          DEPT_NO
---------------------    -----------------------    -------------------    --------------
RAVI                124                   ECE                89
VIJAY               345                   CSE                21
RAJ                 98                    IT                 22
```

```
GIRI                    100                    CSE                    67

SQL> INSERT INTO EMPVIEW VALUES ('SRI', 120,'CSE', 67);

        1 ROW CREATED.

SQL>DROP VIEW EMPVIEW;
view dropped

SQL> CREATE OR REPLACE VIEW EMP_TOTSAL AS SELECT
EMPNO "EID", ENAME "NAME", SALARY "SAL" FROM EMPL;
```

JOIN VIEW:

EXAMPLE:

```
SQL> CREATE OR REPLACE VIEW DEPT_EMP_VIEW AS SELECT A.EMPNO,
                                                    A.ENAME,
                                                    A.DEPTNO,
                                                    B.DNAME,
                                                    B.LOC
                                                    FROM EMPL A, DEPMT B
                                                    WHERE
                                                    A.DEPTNO=B.DEPTNO;
```

12. Explain about integrity constraints and index creation in SQL.

1) not null
2) primary key
3) unique
4) check (P), where P is a predicate
5) Not Null and Unique Constraints

not null

a. Declare *name* and *budget* to be **not null** *name* **varchar**(20) **not null** *budget* **numeric**(12,2) **not null**

- **unique** ($A_1, A_2, \dots, A_m$)
  - ✓ The unique specification states that the attributes *A*1, *A*2, … *A*m form a candidate key.
  - ✓ Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

check (P):

where P is a predicate

Example: ensure that semester is one of fall, winter, spring or summer:

create table *section* (*course_id* **varchar** (8), *sec_id* **varchar** (8), *semester* **varchar** (6), *year* **numeric** (4,0), *building* **varchar** (15), *room_number* **varchar** (7), *time slot id* **varchar** (4), **primary key** (*course_id, sec_id, semester, year*), **check** (*semester* **in** (’Fall’, ’Winter’, ’Spring’, ’Summer’)));

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

  Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.

- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Cascading Actions in Referential Integrity

- **create table** *course (course_id* **char**(5) **primary key**,*title* **varchar**(20),*dept_name* **varchar**(20) **references** *department)*
  - **create table** *course* (…*dept_name* **varchar**(20),
    foreign key (*dept_name*) **references** *department* **on delete cascade**
    on update cascade, . . .)
- alternative actions to cascade: **set null**, **set default**

Integrity Constraint Violation During *Transactions*

- E.g.
  create table *person* (*ID* **char**(10), *name* **char**(40), *mother* **char**(10), *father* **char**(10), **primary key** *ID,* **foreign key** *father* **references** *person,* **foreign key** *mother* **references** *person*)
- How to insert a tuple without causing constraint violation ?
  - ✓ insert father and mother of a person before inserting person
  - ✓ OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - ✓ OR defer constraint checking (next slide)

Complex Check Clauses

- **check** (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*)) why not use a foreign key here?
- Every section has at least one instructor teaching the section.
  - ✓ how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - ✓ Alternative: triggers (later)
- **create assertion** <assertion-name> **check** <predicate>;
  - ✓ Also not supported by anyone

<u>BUILT-IN DATA TYPES IN SQL</u>

- **date:** Dates, containing a (4 digit) year, month and date
  - ✓ Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - ✓ Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - ✓ Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - ✓ Example: interval '1' day
  - ✓ Subtracting a date/time/timestamp value from another gives an interval value
  - ✓ Interval values can be added to date/time/timestamp values

<u>INDEX CREATION</u>

- **create table** *student* (*ID* **varchar** (5), *name* **varchar** (20) **not null**, *dept_name* **varchar** (20), *tot_cred* **numeric** (3,0) **default** 0, **primary key** (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- Indices are data structures used to speed up access to records with specified values for index attributes

  e.g. **select * from** *student* **where** *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

More on indices in Chapter 11

User-Defined Types

create type construct in SQL creates user-defined type

create type *Dollars* **as numeric (12,2) final**

create table *department* (*dept_name* **varchar** (20), *building* **varchar** (15), *budget Dollars*);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types **create domain** *person_name* **char**(20) **not null**
- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree_level* **varchar**(10) **constraint** *degree_level_test*

check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Large-Object Types

Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:

- **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

13. Describe the GRANT functions and explain how it relates to security. What types of privileges may be granted? How rights could be revoked.

Forms of authorization on parts of the database:

- **Read -** allows reading, but not modification of data.
- **Insert -** allows insertion of new data, but not modification of existing data.
- **Update -** allows modification, but not deletion of data. **Delete -** allows deletion of data.

Forms of authorization to modify the database schema **Index -** allows creation and deletion of indices.

- **Resources -** allows creation of new relations.
- **Alteration -** allows addition or deletion of attributes in a relation.
- **Drop -** allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization **grant** <privilege list> **on** <relation name or view name> **to** <user list> <user list> is:
  - ✓ a user-id
  - ✓ **public**, which allows all valid users the privilege granted
  - ✓ A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view

  Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

  grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization. **revoke** <privilege list> **on** <relation name or view name> **from** <user list> Example:

  revoke select on *branch* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.

- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

14. Explain about functions and procedures in SQL.

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

Program 1:

```
 create or replace procedure p1 as
 begin
 dbms_output.put_line('welcome');
end;
 /

Procedure created.

SQL> execute p1;
welcome
```

Program 2:

```
create PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
  BEGIN
  IF x < y THEN
     z:= x;
  ELSE
      z:= y;
  END IF;
 END;
  /

Procedure created.


 declare
 a number;
 b number;
 c number;
  begin
   a:=23;
  b:=4;
  findMin(a,b,c);
  dbms_output.put_line('Minimum value is:'||c);
 end;
  /
Output:
Minimum value is:4
```

<u>Program 3:</u>

```
SQL> create table emp22(id number,name varchar(20),designation varchar(20),salary number);
Table created.
SQL> insert into emp22 values(3,'john','manager',100000);
1 row created.
SQL> insert into emp22 values(3,'jagan','hr',400000);
1 row created.
```

Functions:

```
CREATE OR REPLACE FUNCTION totalemployee return number as
   total number;
     begin
     SELECT count(*) into total from emp22;
    Return total;
     end;
    /
Function created.
```

```
 declare
  a number:=0;
  begin
  a:=totalemployee();
  dbms_output.put_line('Total employees are '||a);
  end;
  /
Total employees are 2
```

<u>Language Constructs for Procedures & Functions:</u>

- SQL supports constructs that gives it almost all the power of a general purpose programming language.
- Compound statement: **begin … end**,
  - ✓ May contain multiple SQL statements between **begin** and **end**.
  - ✓ Local variables can be declared within a compound statements

While and **repeat** statements:

- ✓ **while** boolean expression **do**
  sequence of statements ;
  end while
- ✓ **repeat**

```
                sequence of statements ;
                until boolean expression
            end repeat
```

- **For** loop
  Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n  integer default 0;
 for r  as select budget from department do
            set n = n + r.budget
 end for
```

- Conditional statements (**if-then-else**)
  SQL:1999 also supports a **case** statement similar to C case statement
- Example procedure: registers student after ensuring classroom capacity is not exceeded
  - ✓ Returns 0 on success and -1 if capacity is exceeded
- Signaling of exception conditions, and declaring handlers for exceptions

```
  declare out_of_classroom_seats  condition
declare exit handler for out_of_classroom_seats
        begin
          …
          .. signal out_of_classroom_seats
          end
```

  - ✓ The handler here is **exit --** causes enclosing **begin..end** to be exited

15. Explain about triggers in SQL.

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - ✓ Specify the conditions under which the trigger is to be executed.
  - ✓ Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - ✓ Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - ✓ For example, **after update of** *STUDENT* **on** *grade*
- Values of attributes before and after an update can be referenced
  - ✓ **referencing old row as :** for deletes and updates
  - ✓ **referencing new row as :** for inserts and updates

Table creation:
Create table employee22(empid number,empname varchar(20),empdept varchar(20),salary number);

Example:

CREATE OR REPLACE TRIGGER display_salary

```
BEFORE DELETE OR UPDATE ON employee22
FOR EACH ROW
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

16. Explain about embedded SQL.

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/1.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor
  EXEC SQL <embedded SQL statement >;
  Note: this varies by language:
  - ✓ In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - ✓ In Java embedding uses # SQL { …. };
- Before executing any SQL statements, the program must first connect to the database. This is done using:
  EXEC-SQL **connect to** *server* **user** *user-name* **using** *password*;

Here, *server* identifies the server to which a connection is to be established.

- Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., :*credit_amount)*
- Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.
  ```
  EXEC-SQL BEGIN DECLARE SECTION}
          int  credit-amount ;
   EXEC-SQL END DECLARE SECTION;
  ```
- To write an embedded SQL query, we use the **declare** *c* **cursor for <SQL query>** statement. The variable *c* is used to identify the query
- Example:
  - ✓ From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount in the host langue Specify the query in SQL as follows:

EXEC SQL
    declare *c* **cursor for select** *ID, name* **from** *student*
    where tot_cred > *:credit_amount*
END_EXEC

```
EXEC-SQL connect to  server user user-name using password;
EXEC-SQL BEGIN DECLARE SECTION}
    int  credit-amount ;
EXEC-SQL END DECLARE SECTION;
EXEC SQL
          declare c cursor for select ID,
          name from student where
          tot_cred > :credit_amount
END_EXEC
EXEC SQL open c ;
EXEC SQL fetch c into :si, :sn END_EXEC

  ……….statement to work with student name and id…………..

EXEC SQL close c ;
```

- The variable *c* (used in the cursor declaration) is used to identify the query
- The open statement for our example is as follows:

EXEC SQL **open** *c* ;

This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

EXEC SQL **fetch** *c* **into** :*si, :sn* END_EXEC

Repeated calls to fetch get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area

(SQLCA) gets set to '02000' to indicate no more data is available

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL **close** *c* ;

Note: above details vary with language. For example, the Java   embedding defines Java iterators to step through result tuples.

Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- Can update tuples fetched by cursor by declaring that the cursor is for update

EXEC SQL

declare *c* **cursor for**
select ***from** *instructor* **where** *dept_name* = 'Music' **for update**

- We then iterate through the tuples by performing **fetch** operations on the cursor and after fetching each tuple we execute the following code:

update *instructor*
set *salary* = *salary* + 1000 **where current of** *c*

17. Discuss about dynamic SQL.

- The embedded SQL queries were written as part of the host program source code. Hence, any time we want to write a different query, we must modify the program code, and go through all the steps involved (compiling, debugging, testing, and so on).
- In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) *dynamically at runtime*. For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs.
- Another example is when a user-friendly interface generates SQL queries dynamically for the user based on point-and-click operations on a graphical schema (for example, a QBE-like interface).
- **Dynamic SQL** is one technique for writing this type of database program, by giving a simple example to illustrate how dynamic SQL can work.
- Program segment E3 in Figure reads a string that is input by the user (that string should be an SQL update command) into the string program variable sqlupdatestring in line 3. It then prepares this as an SQL command in line 4 by associating it with the SQL variable sqlcommand. Line 5 then executes the command.
- Notice that in this case no syntax check or other types of checks on the command are possible *at compile time*, since the SQL command is not available until runtime. This contrasts with our previous examples of embedded SQL, where the query could be checked at compile time because its text was in the program source code.

- Although including a dynamic update command is relatively straightforward in dynamic SQL, a dynamic query is much more complicated. This is because usually we do not know the types or the number of attributes to be retrieved by the SQL query when we are writing the program.

- A complex data structure is sometimes needed to allow for different numbers and types of attributes in the query result if no prior information is known about the dynamic query.

- In E3, the reason for separating PREPARE and EXECUTE is that if the command is to be executed multiple times in a program, it can be prepared only once. Preparing the command generally involves syntax and other types of checks by the system, as well as generating the code for executing it. It is possible to combine the PREPARE and EXECUTE commands (lines 4 and 5 in E3) into a single statement by writing

```
EXEC SQL EXECUTE IMMEDIATE :sqlupdatestring ;
```

- This is useful if the command is to be executed only once. Alternatively, the pro-grammer can separate the two statements to catch any errors after the PREPARE statement, if any.

```
EXEC SQL BEGIN DECLARE SECTION ;

varchar sqlupdatestring [256] ;

 EXEC SQL END DECLARE SECTION ;
    ...

 prompt("Enter the Update Command: ", sqlupdatestring) ;
 EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;

 EXEC SQL EXECUTE sqlcommand ;.
```

CS8492 / DATABASE MANAGEMENT SYSTEMS

UNIT-1

PART-A

1. **What is the purpose of Database Management System? Nov/ Dec 2014**

A DBMS is a software for creating and managing databases. It provides users with a systematic way to create, retrieve, update and manage data. It is a middleware between the database which store all the data and the users or applications which need to interact with that stored database. A DBMS can limit what data the end user sees, as well as how that end user can view the data, providing many views of a single database schema

2. **What are the characteristics that distinguish the database approach with the File – based approach? April/ May 2015**

In File System, files are used to store data while, collections of databases are utilized for the storage of data in DBMS. Although File System and DBMS are two ways of managing data, DBMS clearly has many advantages over File Systems. Typically when using a File System, most tasks such as storage, retrieval and search are done manually and it is quite tedious whereas a DBMS will provide automated methods to complete these tasks. Because of this reason, using a File System will lead to problems like data integrity, data inconsistency and data security, but these problems could be avoided by using a DBMS. Unlike File System, DBMS are efficient because reading line by line is not required and certain control mechanisms are in place.

3. **Is it possible for several attributes to have the same domain? Illustrate your answer with suitable example. Nov/ Dec 2015**

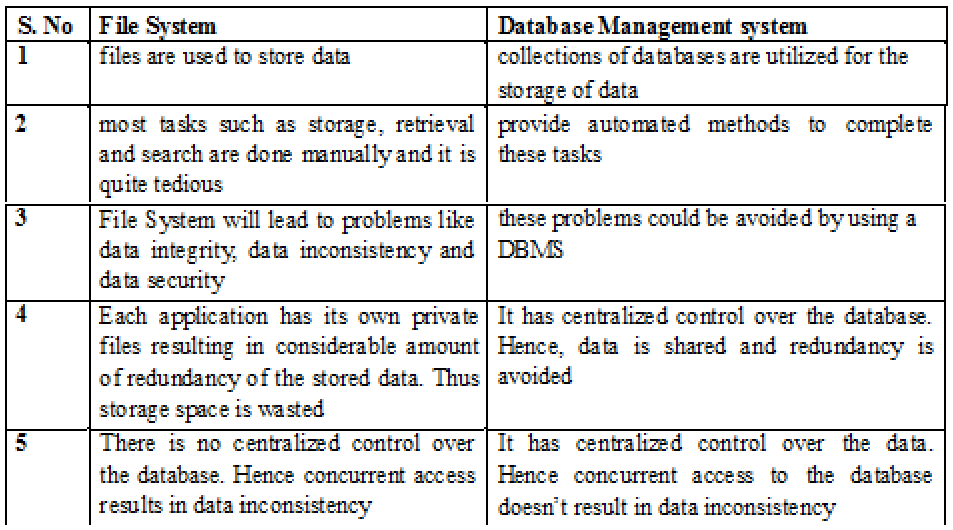
A domain is a pool of values from which the values of specific attributes of specific relations are taken. For example, the domain dept is a set of all possible dept names and the domain emp_name is a set of all employee names. Thus each and every attribute has its own domain. Hence it is not possible for several attributes to have the same domain.

4. **What are the disadvantages of file processing system? May/ June 2016**

The disadvantages of file processing systems are

a) Data redundancy and inconsistency b) Difficulty in accessing data c) Data isolation d) Integrity problems e) Atomicity problems f) Concurrent access anomalies

5. **Differentiate File System with Database Management system. Nov/ Dec 2016**



S. No	File System	Database Management system
1	files are used to store data	collections of databases are utilized for the storage of data
2	most tasks such as storage, retrieval and search are done manually and it is quite tedious	provide automated methods to complete these tasks
3	File System will lead to problems like data integrity, data inconsistency and data security	these problems could be avoided by using a DBMS
4	Each application has its own private files resulting in considerable amount of redundancy of the stored data. Thus storage space is wasted	It has centralized control over the database. Hence, data is shared and redundancy is avoided
5	There is no centralized control over the database. Hence concurrent access results in data inconsistency	It has centralized control over the data. Hence concurrent access to the database doesn't result in data inconsistency

6. **Distinguish key and super key. Nov Dec 2017**

Minimal column which are sufficient to identify row is **primary key**. **Super key** also use for identify row but one **super key** may be contain more than 1 **primary key** or combination of **primary keys** known as **super key**. Both used for uniquely identify of row. Table contain more than 1 candidate **key** but only 1 **primary key**

7. **What are the advantages of using a DBMS?**

a) Controlling redundancy b) Restricting unauthorized access c) Providing multiple user interfaces d) Enforcing integrity constraints. e) Providing backup and recovery

8. **Define instance and schema?**

Instance: Collection of data stored in the data base at a particular moment is called an Instance of the database.

Schema: The overall design of the data base is called the data base schema.

9. **Define the terms 1) Physical schema 2) logical schema.**

Physical schema: The physical schema describes the database design at the physical level, which is the lowest level of abstraction describing how the data are actually stored.

Logical schema: The logical schema describes the database design at the logical level, which describes what data are stored in the database and what relationship exists among the data.

10. **What is storage manager? List the components.**

A storage manager is a program module that provides the interface between the low level data

Stored in a database and the application programs and queries submitted to the system.

The storage manager components include

a) Authorization and integrity manager b) Transaction manager c) File manager d) Buffer manager

11. **What is a data dictionary?**

A data dictionary is a data structure which stores Meta data about the structure of the database ie. The schema of the database.

12. **What are attributes? Give examples.**

An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set.

Example: possible attributes of customer entity are customer name, customer id, Customer Street, customer city.

13. **What is relationship? Give examples**

A relationship is an association among several entities.

Example: A depositor relationship associates a customer with each account that he/she has.

14. **Define the term Relationship set**.

Relationship set: The set of all relationships of the same type is termed as a relationship set.

15. **Define null values**

In some cases a particular entity may not have an applicable value for an attribute or if we do not know the value of an attribute for a particular entity. In these cases null value is used.

16. **List the role of DBA.**

The person who has central control over the system is called database administrator. The functions of the DBA include the following:

- Schema definition
- Storage structure and access-method definition
- Schema and physical-organization modification
- Granting of authorization for data access
- Integrity-constraint specification

17. **Differentiate Static SQL and Dynamic SQL. Nov/ Dec 2014, April/ May 2015, Nov/ Dec 2015, Nov/ Dec 2016**

Static SQL: Static SQL statements are SQL instructions that are a part of the language syntax. It can be used directly in the source code as normal procedural instructions.

Dynamic SQL: It is a programming technique that enables to build SQL statements dynamically at runtime.

18. **Give a brief description in DCL commands. NOV/DEC 2014**

It is a computer language and a subset of SQL, used to control access to data in a database. Examples of DCL commands include:

GRANT: used to allow specified users to perform specified tasks.
REVOKE: used to cancel previously granted permission

19. **Why does SQL allow duplicate tuples in a table or in a query result? Nov/ Dec 2015**

SQL usually treats a table not as a set but rather as a multiset. Duplicate tuples can appear more than once in a table and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries for the following reasons.

i. Duplicate elimination is an expensive operation one way to implement it, is to sort the tuples first and then eliminate duplicates.

ii. The users may want to see duplicate tuples in the result of a query

When an aggregate function is applied to tuples in most cases we do not want to eliminate duplicates

20. **Name the categories of SQL Command. May/ June 2016**

a. data definition language
b. data manipulation language

c. data control language
d. transaction control language

21. What is Data Definition Language? Give example. Nov/ Dec 2016

It is used to define relational database of a system. It creates, changes and removes a table structure.
Ex. CREATE, ALTER, DROP, RENAME and TRUNCATE.

22. Define Data Manipulation Language.

DML: A data manipulation language is a language that enables users to insert, modify and delete the data in the database.

Ex. Insert, delete, modify

23. List the SQL statements used for Transaction Control.

a. Commit
b. Rollback
c. Save point
d. Set Transaction

24. Define database objects.

A database object is any defined object in the database that is used to store or reference data. Some examples include tables, views, clusters, sequences, indexes and synonyms.

25. Name the different types of joins supported in SQL.

a. Inner join
b. Outer join
    1. Left Outer Join
    2. Right Outer Join
    3. Full Outer Join
c. Natural join

26. What is a trigger in SQL?

A trigger is a statement that the system executes automatically as a side effect of a modification to the database.

27. What are primary keys?

A primary key is one or more columns in a table used to uniquely identify each row in the table. Its values must not be null and must be unique across the column.

28. Explain the basic structure of an SQL expression.

An SQL Expression has three clauses: select, from and where.

α. Select- used to list the attributes desired in the result of a query
β. From- lists the relations to be scanned
χ. Where- predicate involving attributes in the from clause

select A1, A2….An from r1, r2,….rn where p;

29. Write a SQL Statement to find the names & loan numbers of all customers who have a loan at Chennai branch from the following relations.

i). Loan (Loan _ no, Branch _ name, amount)
ii). Branch (Branch _ name, Branch _city, Assets)

select Loan_no from Loan, Branch where Loan.Branch_name = Branch.Branch_name and Branch_city =’Chennai’;

Part B

1. Consider a student registration database comprising of the below given table schema .

student file

student number , student name , address , telephone

course file

course number , description , hours , professor number

professor file

professor number , name , office

registration file

student number , course number , date

consider a suitable example of tuples/records for the above mentioned tables and write DML , statements (SQL) to answer for the queries listed below
i) Which courses does a specific professor teach ?
ii) What courses are taught by two specific professors?
iii) Who teaches a specific courses and where is his/her office?
iv)For specific student number in which courses is the student registered and what is his/her name?
v) who are the professors for a specific student ?
vi) Who are the students registered in specific courses?

2) Explain the following with examples:
i) DDL
ii) DML
ii) Embedded SQL

3) Assume the following table :
degree(degcode,name,subject)
candidate(seat no, degcode,semesrer,month,year,result)
Marks(seatno,degcode,semester,month,year,papcode,marks)
Degcode-degree code.Name-name of the degree(MSC.MCOM)
SUBJECT_subject of courses Eg. Phy , pap code –paper code eg.A1
Solve the following queries using SQL
i) Write a SELECT statement to display all the degree codes are there in the candidate table but not present in degree table in the order of degcode.
ii) Write a SELECT statement to display the name of all the candidates who have got less than 40 marks in exactly 2 subjects .
iii) Write a SELECT statement to display the name, subject and number of the candidate for all degrees in which there are less than 5 candidates.
iv) Write a SELECT statement to display the names of all the candidates who have got highest total marks in MSc., (Maths).

4) Describe the GRANT functions and explain how it relates to security. what types of privileges may be granted? How rights could be revoked ?

5) With the help of a neat block diagram , explain the basic architecture of a data base management system?
6) Describe the six clauses in the syntax of an sql query and show what type of constructs can be specified in each of the six clauses.which of the six clauses are required and which are optional?

7) List the operations of relational algebra and the purpose of each with example.
8) consider the relation schema given in figure 1.design and draw an ER diagram that capture the information of this schema.

Employee(empno,name,office,age)

Books(isbn,tittle ,authors,publisher)
Loan(empno,isbn,date)

write the following queries in relational algebra and SQL.
i)find the name of employees who have borrowed a book published by McGraw-Hill.
ii) find the name of employees who have borrowed all books published by McGraw-Hill.

9) Write the DDL,DML,DCL commands for the students database.Which contains
Student details :name,id,DOB,branch,DOJ.
Course details:Course name,Course id,Stud.id,Faculty name,id,marks.

10) Differentiate between foreign key constraints and referential integrity constraints with suitable examples?
11) Justify the need of embedded SQL .consider the relation student(studentno,name,mark and grade).Write embedded dynamic SQL statements in C language to retrieve all the student's records whose marks more than 90.
12) Explain about functions and procedures in SQL.
13) Discuss about triggers.

SQL QUERIES (UNIVERSITY QUESTIONS)

1) Consider a student registration database comprising of the below given table schema.

Student
student number, student name, address, telephone
course
course number, description, hours, professor number
professor
professor number, name, office
registration
student number, course number, date

Consider a suitable example of tuples/records for the above mentioned tables and write DML, statements (SQL) to answer for the queries listed below
i) Which courses does a specific professor teach and what courses are taught by two specific professors?
ii) Who teaches a specific courses and where is his/her office?
iii) For specific student number in which courses is the student registered and what is his/her name?
iv) Who are the professors for a specific student and who are the students registered in specific courses?

Answers:

i)select courseno from course where professor no=10001;
select courseno from course where professor no=10001 or professor no=10004;
ii) select name , office from professor, course where coursed=101 and professor.professorno=course.professorno;

iii) select courseno,studentname from registration . student where student.studentno= registration.studentno;
iv)select professorno from course , registration where studentno=5001 and course.courseno= registration.courseno;

select studentno from registration where courseno=6003;

2) Consider the relation schema given in figure.
Employee(empno, name, office, age)
Books(isbn, tittle, authors, publisher)
Loan(empno, isbn, date)
Write the following queries in relational algebra and SQL.
a) Find the name of employees who have borrowed a book published by McGraw-Hill.
b) Find the name of employees who have borrowed all books published by McGraw-Hill.
c) Find the name of the employee who has loan and age greater than 25

Answers:

a) select name from employee e, books b, loan l where e.empno = l.empno and l.isbn = b.isbn and b.publisher = 'McGrawHill'

b) select distinct e.name from employee e where not exists ((select isbn from books where publisher = 'McGrawHill') except (select isbn from loan l where l.empno = e.empno));

c) select name from employee e, loan l where l.age>25 and e.empno=l.empno;

4) Elaborate about various join types of operations in SQL with example.
Assume the following table:
degree(degcode,name,subject)
candidate(seat no, degcode,semesrer,month,year,result)
Marks(seatno,degcode,semester,month,year,papcode,marks)
Degcode-degree code.Name-name of the degree(MSC.MCOM)
SUBJECT_subject of courses Eg. Phy , pap code –paper code eg.A1
Solve the following queries using SQL
i) Write a SELECT statement to display all the degree codes are there in the candidate table but not present in degree table in the order of degcode.
ii) Write a SELECT statement to display the name of all the candidates who have got less than 40 marks in exactly 2 subjects .
iii) Write a SELECT statement to display the name, subject and number of the candidate for all degrees in which there are less than 5 candidates.
iv) Write a SELECT statement to display the names of all the candidates who have got highest total marks in MSc., (Maths).

Answers:
i) select degcode from candidate where degcode not in select degcode from degree;

ii) select name from candidate,marks where candidate.seatid=marks.seatid and marks.mark <40 ;

iii) Select name,subject, count (seatno) from degree, marks group by degcode;

iv) Select seatno from marks m1,marks m2 where m1.mark>m2.mark;

5. Justify the need of embedded SQL .consider the relation student(studentno, name, mark and grade).Write embedded SQL statements in C language to retrieve all the student's records whose marks more than 90.

```
        EXEC-SQL connect to  server user user-name using password;
                EXEC SQL BEGIN DECLARE SECTION;
                int  studno, mark1,grad ;
                char sname[30];
                EXEC SQL END DECLARE SECTION;
 EXEC SQL
 declare c cursor for select studentno, name, mark , grade from student where mark>90
  END EXEC
EXEC SQL open c ;
       EXEC SQL fetch c into :studno, :sname,  :mark1, :grad END_EXEC

       ……………………………
       …………………..
EXEC SQL close c ;
```

UNIT II DATABASE DESIGN

Entity-Relationship model – E-R Diagrams – Enhanced-ER Model – ER-to-Relational Mapping – Functional Dependencies – Non-loss Decomposition – First, Second, Third Normal Forms, Dependency Preservation – Boyce/Codd Normal Form – Multi-valued Dependencies and Fourth Normal Form – Join Dependencies and Fifth Normal Form.

2.1. ENTITY-RELATIONSHIP MODEL and E-R DIAGRAMS:

- The term **database application** refers to a particular database and the associated programs that implement the database queries and updates.
- For example, a BANK database application that keeps track of customer accounts would include programs that implement database updates corresponding to customer deposits and withdrawals. These programs provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus for the end users of the application— the bank tellers, in this example. Hence, a major part of the database application will require the design, implementation, and testing of these application programs.
- Traditionally, the design and testing of **application programs** has been considered to be part of software engineering rather than database design. In many software design tools, the database design methodologies and software engineering methodologies are intertwined since these activities are strongly related.
- **Entity-Relationship** (**ER**) **model** is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts.
- The diagrammatic notation associated with the ER model is known as **ER diagrams**.

Using High-Level Conceptual Data Models for Database Design:

1) **Requirements collection and analysis** :

- During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates. In software design, it is common to use data flow diagrams, sequence diagrams, scenarios, and other techniques to specify functional requirements.

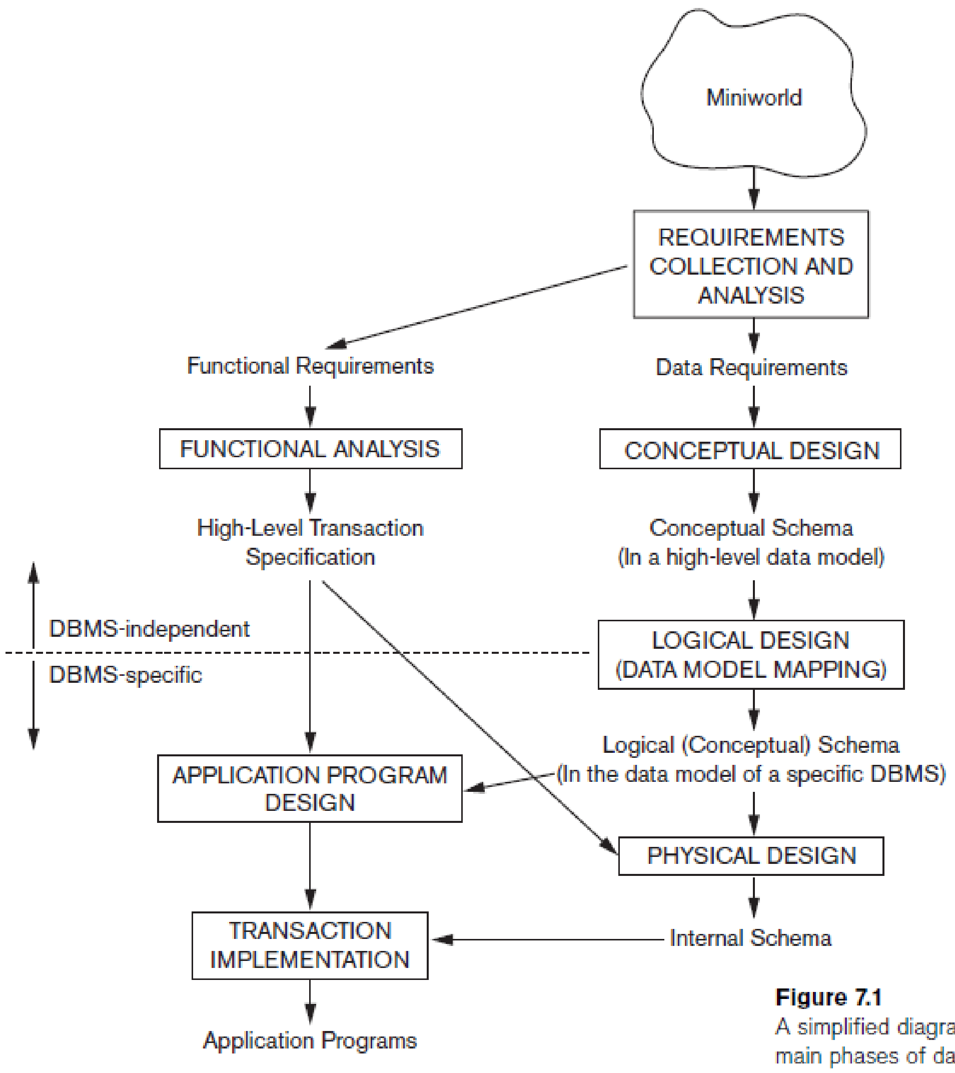

Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Data Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-Level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
APPLICATION PROGRAM DESIGN
PHYSICAL DESIGN
TRANSACTION IMPLEMENTATION
Internal Schema
Application Programs

Figure 7.1
A simplified diagram to illustrate the main phases of database design.

2) Conceptual design:

- Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts pro-vided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details. This makes it is easier to create a good conceptual data-base design.

- During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

3) **Logical design:**

- The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping**; its result is a database schema in the implementation data model of the DBMS. Data model mapping is often automated or semiautomated within the database design tools.

4) **Physical design:**

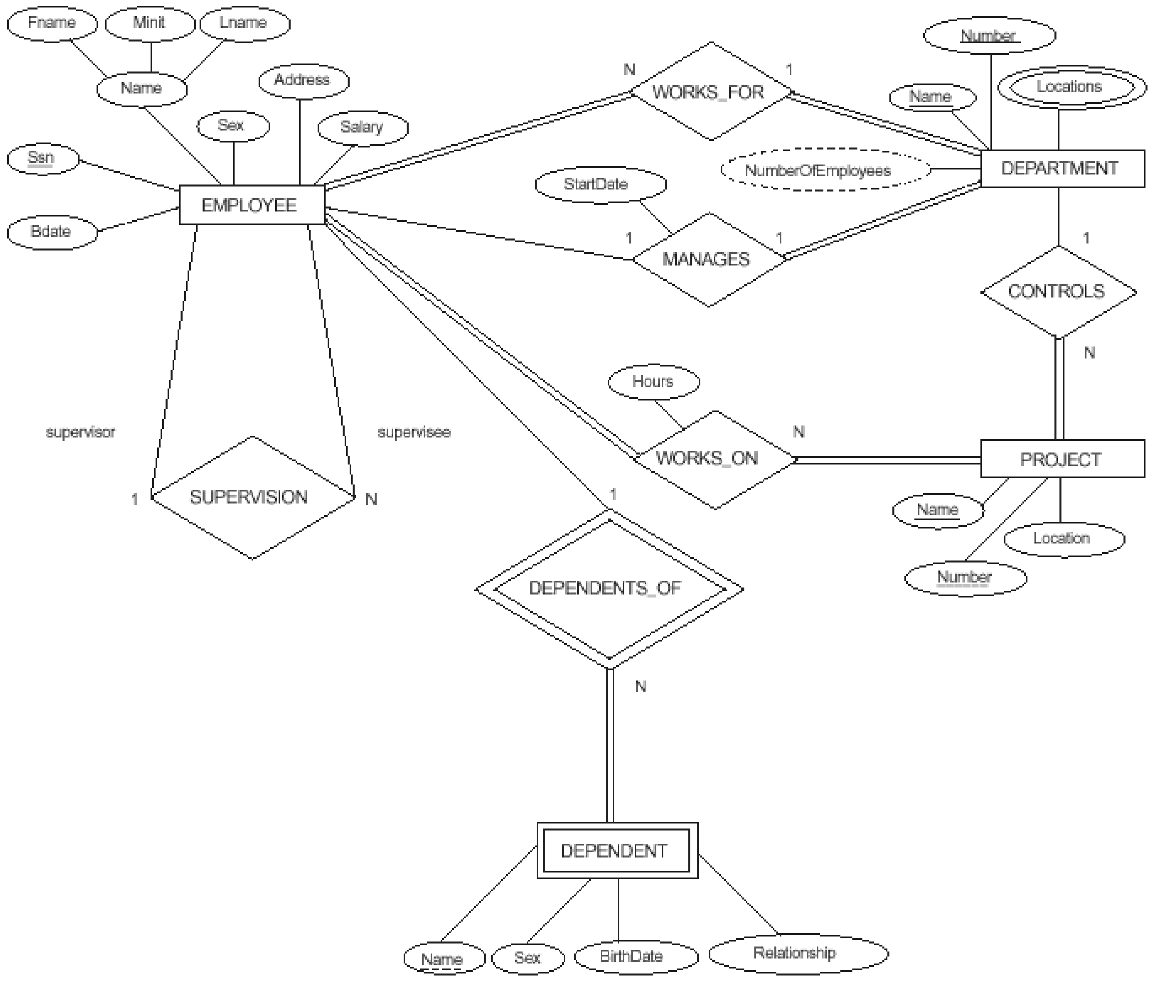
- The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications.

A Sample Database Application

- Consider a sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design.

- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.

- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- We store each employee's name, Social Security number,address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).
- We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relation-ship to the employee.
- Figure shows how the schema for this database application can be displayed by means of the graphical notation known as **ER diagrams**. This figure will be explained gradually as the ER model concepts are presented. We describe the step-by-step process of deriving this schema from the stated requirements—and explain the ER diagrammatic notation—as we introduce the ER model concepts.



Entity Types, Entity Sets, Attributes, and Keys

- The ER model describes data as entities, relationships, and attributes.

1) **Entities and Attributes:**

- The basic object that the ER model represents is an **entity**, which is a thing in the real world with an independent existence.

- An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course).

- Each entity has **attributes**—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job.

- A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

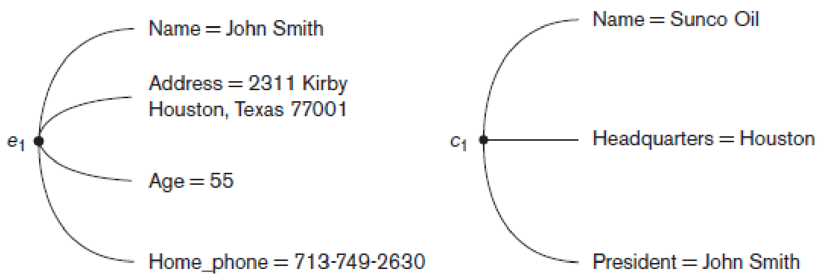


Figure 7.3
Two entities, EMPLOYEE e_1, and COMPANY c_1, and their attributes.

- Figure shows two entities and the values of their attributes. The EMPLOYEE entity e_1 has four attributes: Name, Address, Age, and Home_phone; their values are 'John Smith,' '2311 Kirby, Houston, Texas 77001', '55', and '713-749-2630', respec-tively. The COMPANY entity c_1 has three attributes: Name, Headquarters, and President; their values are 'Sunco Oil', 'Houston', and 'John Smith', respectively.

Several types of attributes occur in the ER model:

i) simple versus composite

ii) single-valued versus multivalued

iii) stored versus derived.

i) Composite versus Simple (Atomic) Attributes.

- **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings.
- For example, the Address attribute of the EMPLOYEE entity shown in Figure can be subdivided into Street_address, City, State, and Zip, with the values '2311 Kirby', 'Ashok nagar', 'Tamilnadu', and '600001.'
- Attributes that are not divisible are called **simple** or **atomic attributes**.
- Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number. The value of a composite attribute is the concatenation of the values of its component simple attributes.

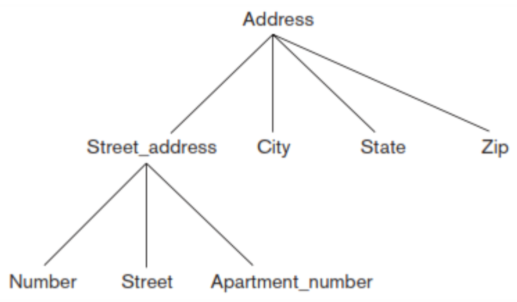


Figure .A hierarchy of composite attributes.

- Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its components. If the composite attribute is referenced only as a whole, there is no need to subdivide it into component attributes. For example, if there is no need to refer to the individual components of an address (Zip Code, street, and so on), then the whole address can be designated as a simple attribute.

ii) <u>Single-Valued versus Multivalued Attributes:</u>

- Most attributes have a single value for a particular entity; such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College_degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; therefore, different people can have different numbers of values for the College_degrees attribute. Such attributes are called **multivalued**. A multivalued attribute may have lower and upper bounds to constrain the number of values allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and three values, if we assume that a car can have three colors at most.

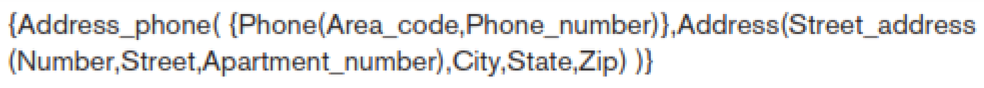
iii) Stored versus Derived Attributes:

- In some cases, two (or more) attribute values are related—for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date. The Age attribute is hence called a **derived attribute** and is said to be **derivable from** the Birth_date attribute, which is called a **stored attribute**. Some attribute values can be derived from related entities; for example, an attribute Number_of_employees of a DEPARTMENT entity can be derived by counting the number of employees related to (working for) that department.

NULL Values:

In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called NULL is created. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees. NULL can also be used if we do not know the value of an attrib-ute for a particular entity—for example, if we do not know the home phone number of 'John Smith'. The meaning of the former type of NULL is not applicable, whereas the meaning of the latter is unknown. The unknown category of NULL can be further classified into two cases. The first case arises when it is known that the attribute value exists but is missing—for instance, if the Height attribute of a person is listed as NULL. The second case arises when it is not known whether the attribute value exists—for example, if the Home_phone attribute of a person is NULL.

Complex Attributes:



{Address_phone({Phone(Area_code,Phone_number)},Address(Street_address
(Number,Street,Apartment_number),City,State,Zip))}

Figure: complex attribute : Address_phone

In general, composite and multivalued attributes can be nested arbitrarily. We can represent arbitrary nesting by grouping components of a composite attribute between parentheses () and separating the components with commas, and by displaying multivalued attributes between braces { }. Such attributes are called **complex attributes**. For example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person can be specified as shown in Figure. Both Phone and Address are themselves composite attributes.

2) Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets.

- A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its own value(s) for each attribute.

- An **entity type** defines a collection (or set) of entities that have the same attributes. Each entity type in the database is described by its name and attributes.

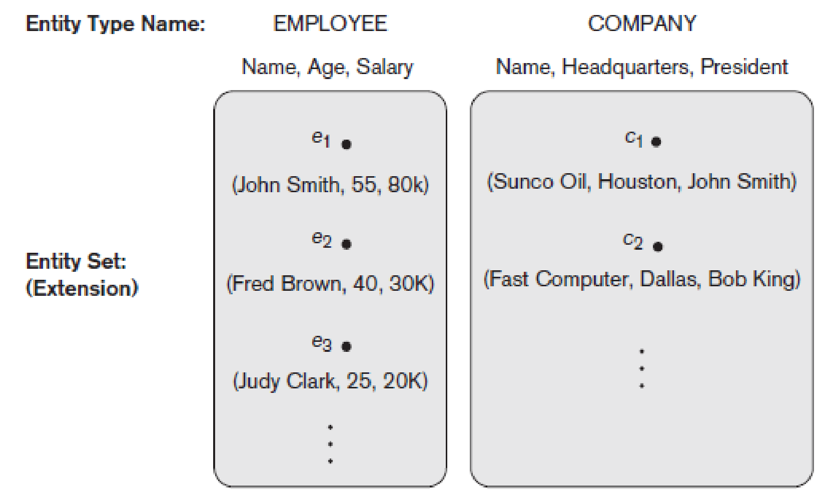


Figure 7.6
Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

- Figure shows two entity types: EMPLOYEE and COMPANY, and a list of some of the attributes for each. A few individual entities of each type are also illustrated, along with the values of their attributes.

- The collection of all entities of a particular entity type in the data-base at any point in time is called an **entity set**; the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a type of entity as well as the current set of all employee entities in the database.

- An entity type is represented in ER diagrams as a rectangular box enclosing the entity type name.

- Attribute names are enclosed in ovals and are attached to their entity type by straight lines.

- Composite attributes are attached to their component attributes by straight lines.

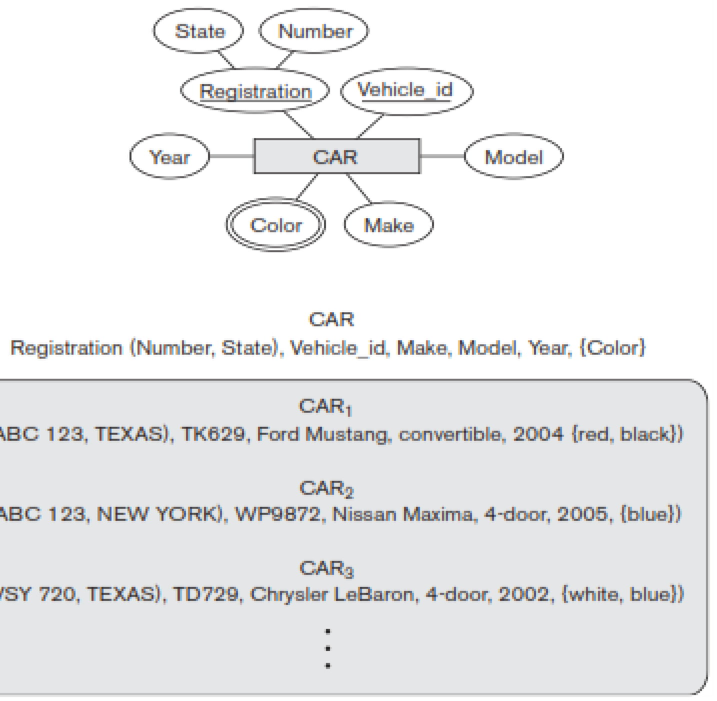
- Multivalued attributes are displayed in double ovals. Figure (a) shows a CAR entity type in this notation.

- An entity type describes the **schema** or **intension** for a set of entities that share the same structure. The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.

Figure 7.7
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.

(a)

(b)



CAR
Registration (Number, State), Vehicle_id, Make, Model, Year, {Color}

CAR_1
((ABC 123, TEXAS), TK629, Ford Mustang, convertible, 2004 {red, black})

CAR_2
((ABC 123, NEW YORK), WP9872, Nissan Maxima, 4-door, 2005, {blue})

CAR_3
((VSY 720, TEXAS), TD729, Chrysler LeBaron, 4-door, 2002, {white, blue})

⋮

Key Attributes of an Entity Type:

- An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely.

- For example, the Name attribute is a key of the COMPANY entity type, because no two companies are allowed to have the same name.

- For the PERSON entity type, a typical key attribute is Ssn (Social Security number). Sometimes several attributes together form a key, meaning that the combination of the attribute values must be distinct for each entity.

- If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a composite attribute and designate it as a key attribute of the entity type. Notice that such a composite key must be minimal; that is, all component attributes must be included in the composite attribute to have the uniqueness property. Superfluous attributes must not be included in a key. In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure (a).

- Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for every entity set of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. It is not the property of a particular entity set; rather, it is a constraint on any entity set of the entity type at any point in time. This key con-straint (and other constraints we discuss later) is derived from the constraints of the miniworld that the database represents.

- Some entity types have more than one key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR (Figure 7.7) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, neither of which is a key on its own. An entity type may also have no key, in which case it is called a weak entity type..

- In our diagrammatic notation, if two attributes are underlined separately, then each is a key on its own. Unlike the relational model, there is no concept of primary key in the ER modele; the primary key will be chosen during mapping to a relational schema.

Value Sets (Domains) of Attributes.

- Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity.

- In Figure, if the range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70. Similarly, we can specify the value set for the Name attribute to be the set of strings of alphabetic characters separated by blank characters, and so on.

- Value sets are not displayed in ER diagrams, and are typically specified using the basic **data types** available in most programming languages, such as integer, string, Boolean, float, enumerated type, subrange, and so on. Additional data types to represent common database types such as date, time, and other concepts are also employed.

- Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function** from E to the power set[6] P(V) of V:

$$A : E \rightarrow P(V)$$

- We refer to the value of attribute A for entity e as A(e). The previous definition cov-ers both single-valued and multivalued attributes, as well as NULLs. A NULL value is represented by the empty set. For single-valued attributes, A(e) is restricted to being a singleton set for each entity e in E, whereas there is no restriction on multivalued attributes.[7] For a composite attribute A, the value set V is the power set of the Cartesian product of $P(V_1)$, $P(V_2)$, ..., $P(V_n)$, where V_1, V_2, ..., V_n are the value sets of the simple component attributes that form A:

$$V = P\ (P(V_1) \times P(V_2) \times ... \times P(V_n))$$

- The value set provides all possible values. Usually only a small number of these val-ues exist in the database at a particular time. Those values represent the data from the current state of the miniworld. They correspond to the data as it actually exists in the miniworld.

Initial Conceptual Design of the COMPANY Database

1. An entity type DEPARTMENT with attributes Name, Number, Locations,Manager, and Manager_start_date. Locations is the only multivalued attribute.We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.

2. An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—First_name, Middle_initial, Last_name—or of Address.
4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date, and Relationship (to the employee).

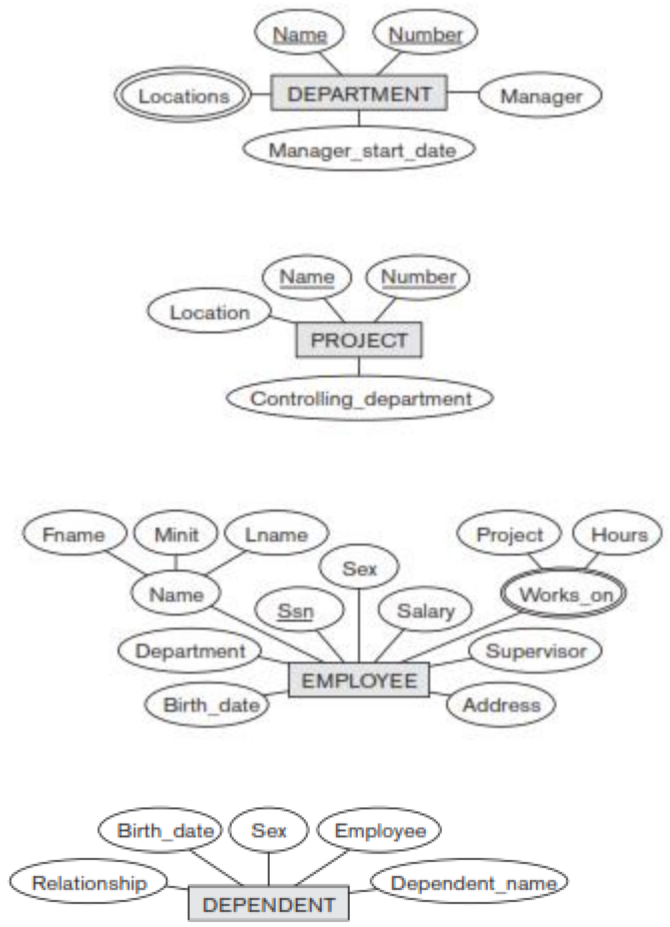


Figure. Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

<u>Relationship types, Roles, Structural Constraints:</u>

Relationship type: R among n Entity types E1, ..., En defines a set of associations among entities from these types. Each association will be denoted as:
(e1, ..., en) where ei belongs to Ei, 1 <= i <= n.
ex. WORKS_FOR relationship

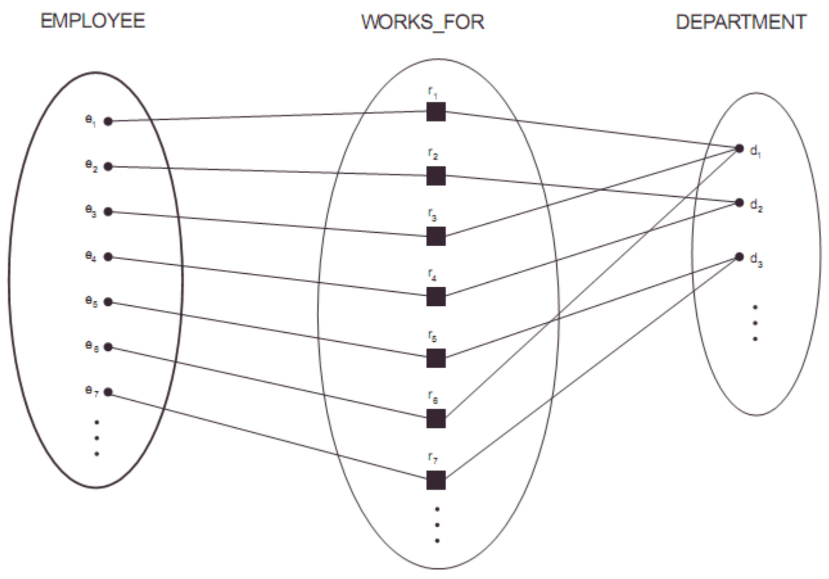


Figure. Some instances of the WORKS_FOR relationship between EMPLOYEE and DEPARTMENT.

Degree of a relationship:

The degree of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called binary, and one of degree three is called ternary. An example of a ternary relationship is SUPPLY Degree of relationship = n (usually n = 2, binary relationship)

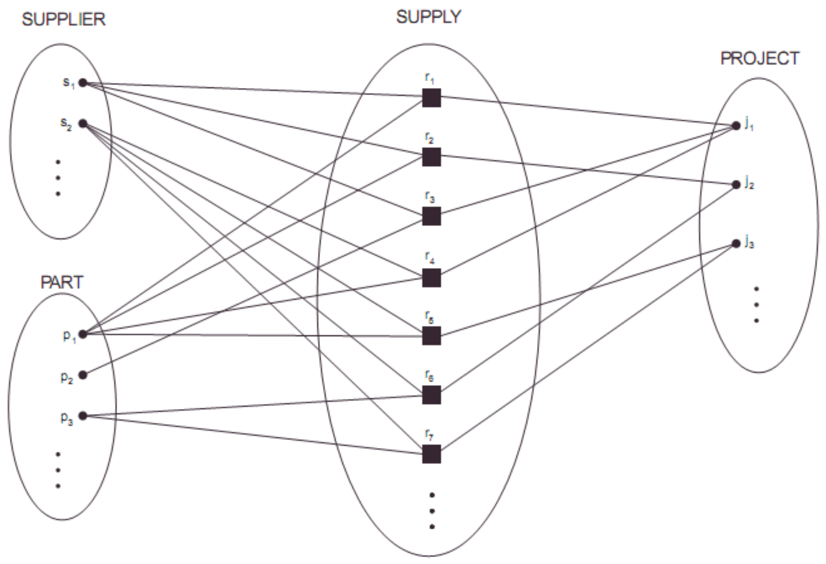


Figure. Some relationship instances of a ternary relationship SUPPLY.

Relationships as Attributes: It is sometimes convenient to think of a relationship type in terms of attributes,

Role names

Each entity participating in a relationship has a ROLE.

E.g. Employee plays the role of worker and Department plays the role of employer in the WORKS_FOR relationship type

Role names are more important in recursive relationships.

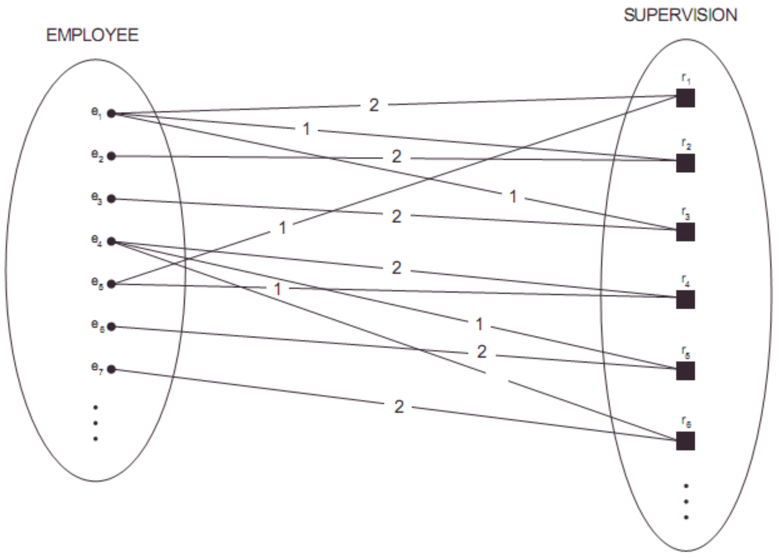


Figure. The recursive relationship SUPERVISION, where the EMPLOYEE entity type plays the two roles of supervisor (1) and supervisee (2).

Structural Constraints on Relationships

Two types:

1) Cardinality Ratio Constraint (1-1, 1-N, M-N)
2) Participation Constraint

* Total participation (existence dependency)

* Partial participation

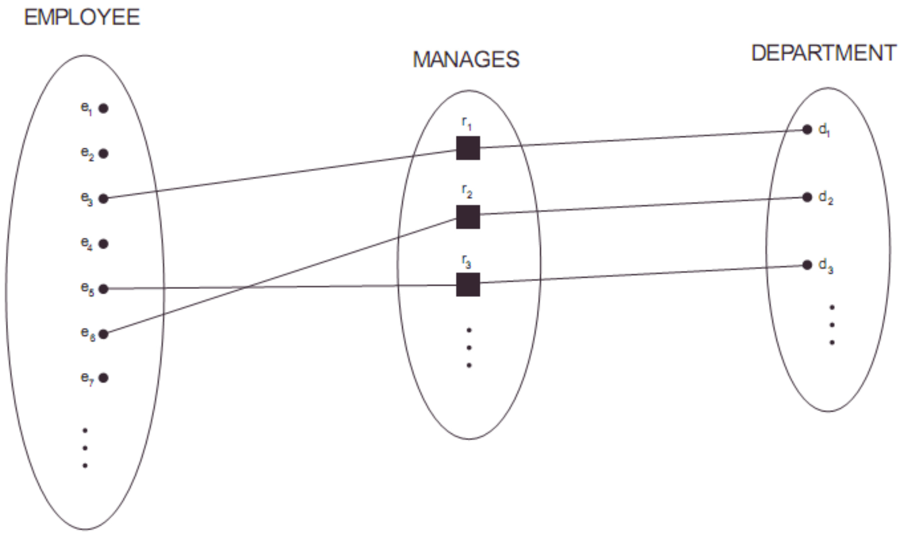


Figure. The 1:1 relationship MANAGES, with partial participation of employee and total participation of DEPARTMENT.

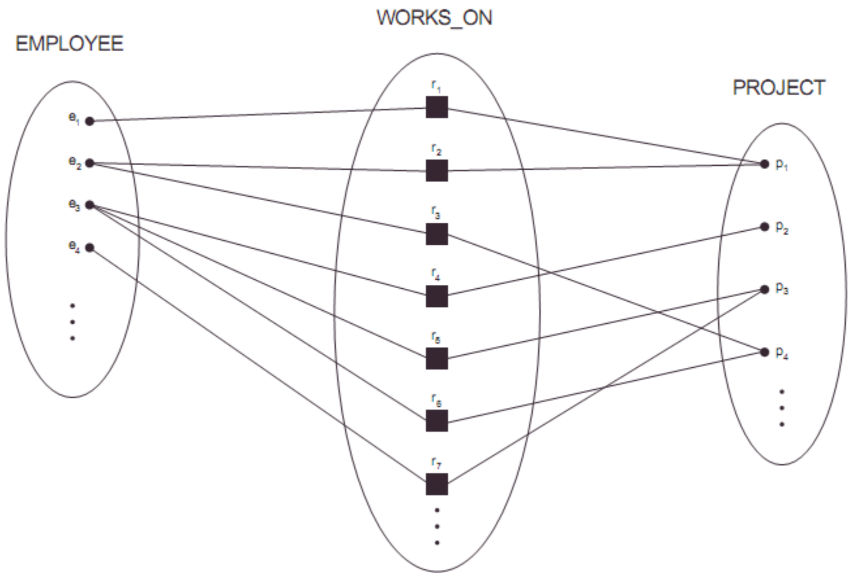


Figure. The M:N relationship WORKS_ON between EMPLOYEE and PROJECT.

In ER Diagrams:
Total participation is denoted by double line and partial participation by single line cardinality ratios are mentioned as labels of edges.

Attributes of relationships:

e.x. Hours attribute for WORKS_ON relationship
If relationship is 1-N or 1-1, these attributes can be migrated to the entity sets involved in the relationship.
1-N: migrate to N side
1-1: migrate to either side

Weak Entity Types

- An entity that does not have a key attribute
- A weak entity must participate in an identifying relationship type with an owner or identifying entity type
- Entities are identified by the combination of:
  - A partial key of the weak entity type
  - The particular entity they are related to in the identifying entity type

Example:

Suppose that a DEPENDENT entity is identified by the dependent's first name and birthdates, and the specific EMPLOYEE that the dependent is related to. DEPENDENT is a weak entity type with EMPLOYEE as its identifying entity type via the identifying relationship type DEPENDENT_OF

Weak Entity Type is: DEPENDENT

Identifying Relationship is: DEPENDENTS_OF

Constraints on Relationships

- Constraints on Relationship Types
  - (Also known as ratio constraints)
  - Maximum Cardinality
    - One-to-one (1:1)
    - One-to-many (1:N) or Many-to-one (N:1)
    - Many-to-many
  - Minimum Cardinality (also called participation constraint or existence dependency constraints)
    - zero (optional participation, not existence-dependent)
    - one or more (mandatory, existence-dependent)

Notation for ER Diagrams:

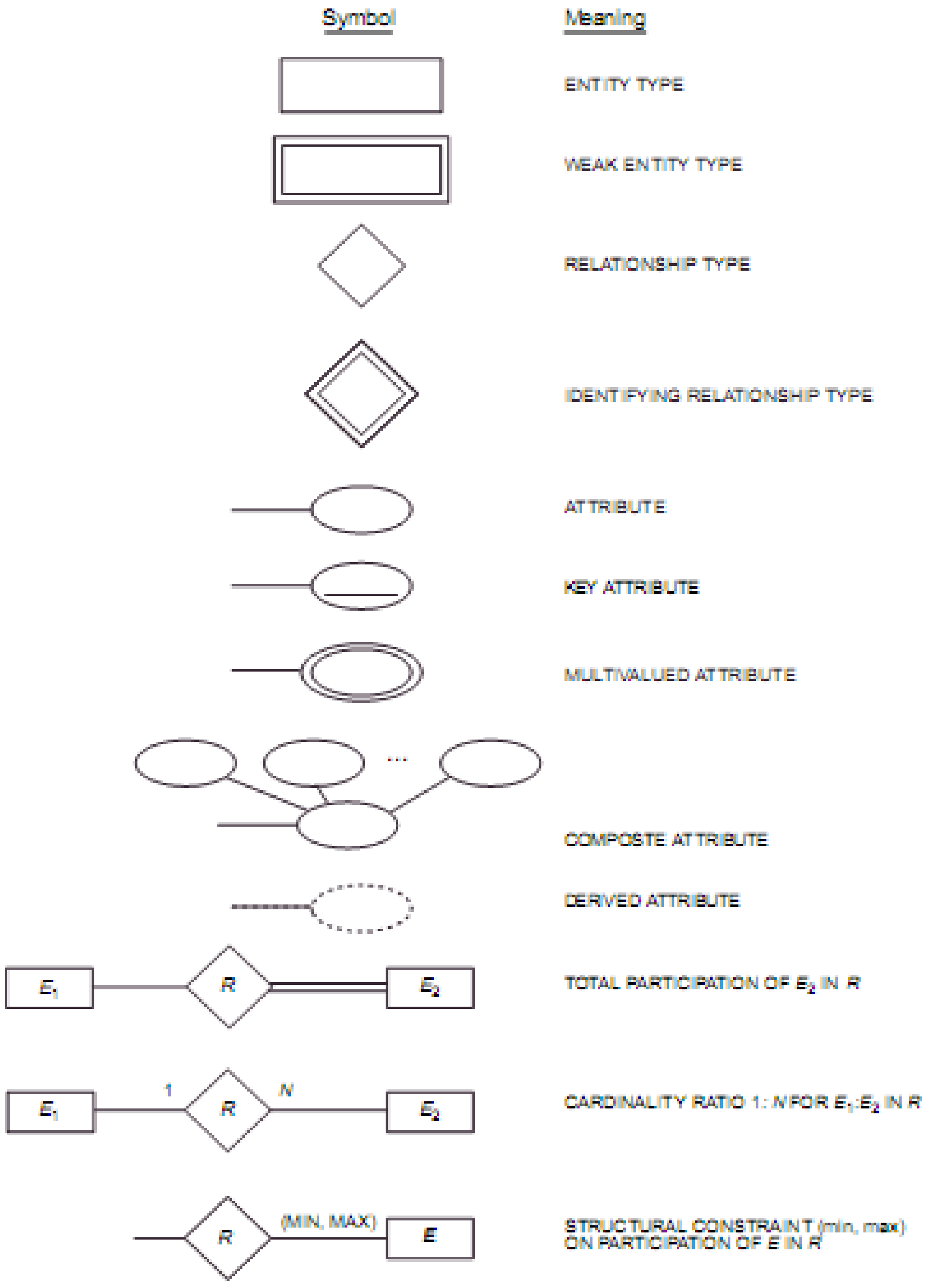


Figure 3.14 Summary of ER diagram notation.

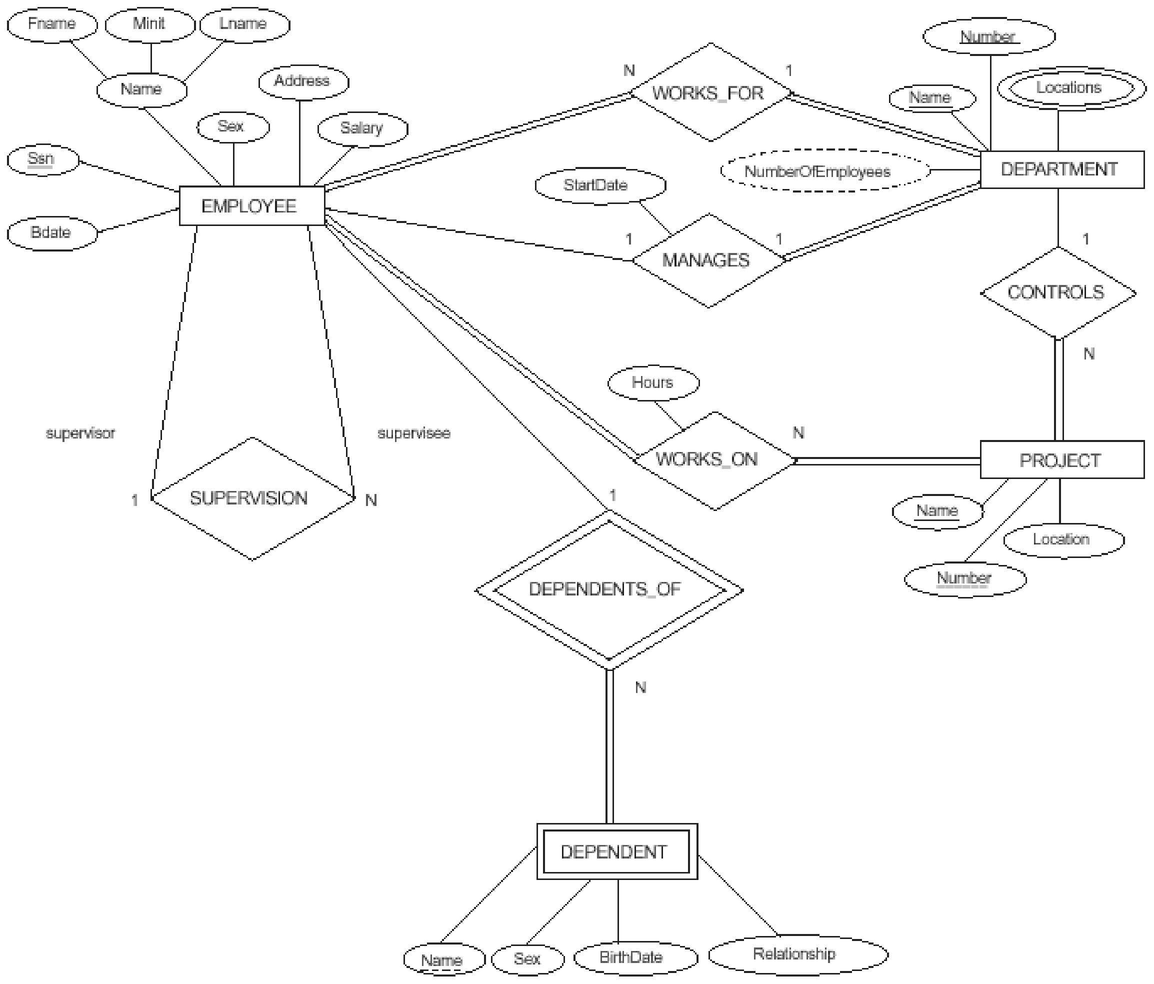


Figure. ER diagram for the COMPANY schema, with all role names included and with structural constraints on relationships specified using the alternate notation (min, max).

2.2. EXTENDED-ER (EER) MODEL :

Extended-ER (EER) Model Concepts

- Includes all modeling concepts of basic ER
- Additional concepts: subclasses/superclasses, specialization/generalization, categories, attribute inheritance
- The resulting model is called the enhanced-ER or Extended ER (E2R or EER) model
- It is used to model applications more completely and accurately if needed
- It includes some object-oriented concepts, such as inheritance

Subclasses and Superclasses:

- An entity type may have additional meaningful subgroupings of its entities
- Example: EMPLOYEE may be further grouped into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE,…

  - Each of these groupings is a subset of EMPLOYEE entities
  - Each is called a subclass of EMPLOYEE
  - EMPLOYEE is the superclass for each of these subclasses
- These are called superclass/subclass relationships.
- Example: EMPLOYEE/SECRETARY, EMPLOYEE/TECHNICIAN

- These are also called IS-A relationships (SECRETARY IS-A EMPLOYEE, TECHNICIAN IS-A EMPLOYEE, …).
- Note: An entity that is member of a subclass represents the same real-world entity as some member of the superclass
  - The Subclass member is the same entity in a distinct specific role
  - An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass
  - A member of the superclass can be optionally included as a member of any number of its subclasses
- Example: A salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE
  - It is not necessary that every entity in a superclass be a member of some subclass

Attribute Inheritance in Superclass / Subclass Relationships

- An entity that is member of a subclass inherits all attributes of the entity as a member of the superclass .
  It also inherits all relationships

<u>Specialization and Generalization:</u>

1)Specialization

- Is the process of defining a set of subclasses of a superclass
- The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass
- Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon job type.
  - May have several specializations of the same superclass
- Example: Another specialization of EMPLOYEE based in method of pay is {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.
  - Superclass/subclass relationships and specialization can be diagrammatically represented in EER diagrams
  - Attributes of a subclass are called specific attributes. For example, TypingSpeed of SECRETARY
  - The subclass can participate in specific relationship types. For example, BELONGS_TO of HOURLY_EMPLOYEE

Example of a Specialization

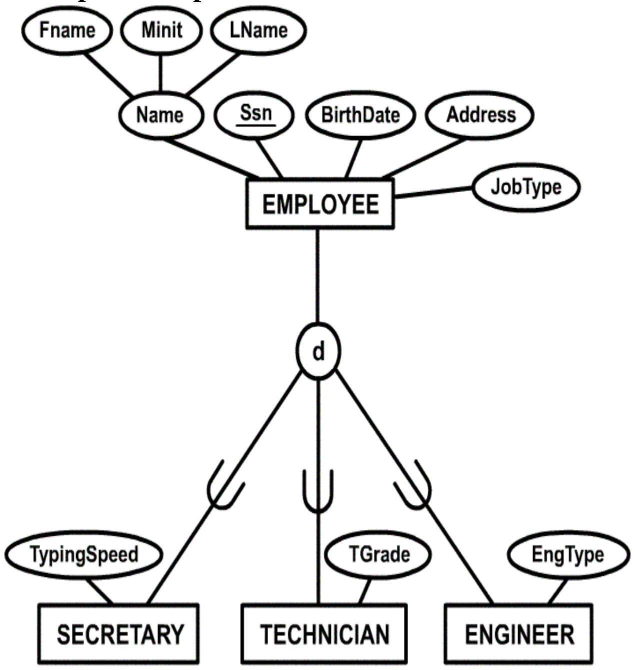
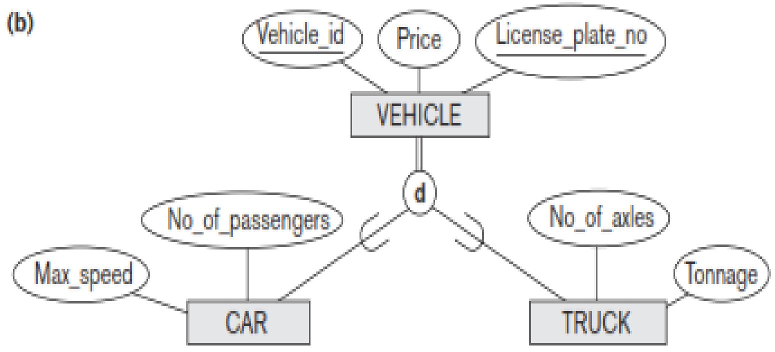


Figure.Specialization

2)Generalization

- The reverse of the specialization process
- Several classes with common features are generalized into a superclass; original classes become its subclasses
- Example: CAR, TRUCK generalized into VEHICLE; both CAR, TRUCK become subclasses of the superclass VEHICLE.
  - We can view {CAR, TRUCK} as a specialization of VEHICLE
  - Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK



3)Generalization and Specialization

- Diagrammatic notation sometimes used to distinguish between generalization and specialization
  - Arrow pointing to the generalized superclass represents a generalization
  - Arrows pointing to the specialized subclasses represent a specialization
  - We do not use this notation because it is often subjective as to which process is more appropriate for a particular situation
  - We advocate not drawing any arrows in these situations

- Data Modeling with Specialization and Generalization
  - A superclass or subclass represents a set of entities
  - Shown in rectangles in EER diagrams (as are entity types)
  - Sometimes, all entity sets are simply called classes, whether they are entity types, superclasses, or subclasses

Constraints on Specialization and Generalization

- If we can determine exactly those entities that will become members of each subclass by a condition, the subclasses are called predicate-defined (or condition-defined) subclasses
  - Condition is a constraint that determines subclass members
  - Display a predicate-defined subclass by writing the predicate condition next to the line attaching the subclass to its superclass
- If all subclasses in a specialization have membership condition on same attribute of the superclass, specialization is called an attribute defined-specialization
  - Attribute is called the defining attribute of the specialization
  - Example: JobType is the defining attribute of the specialization {SECRETARY, TECHNICIAN, ENGINEER} of EMPLOYEE
- If no condition determines membership, the subclass is called user-defined
  - Membership in a subclass is determined by the database users by applying an operation to add an entity to the subclass
  - Membership in the subclass is specified individually for each entity in the superclass by the user
- Two other conditions apply to a specialization/generalization:
- **Disjointness Constraint**:
  - Specifies that the subclasses of the specialization must be disjointed (an entity can be a member of at most one of the subclasses of the specialization)
  - Specified by d in EER diagram
  - If not disjointed, overlap; that is the same entity may be a member of more than one subclass of the specialization
  - Specified by o in EER diagram
- **Completeness Constraint**:
  - Total specifies that every entity in the superclass must be a member of some subclass in the specialization/ generalization
  - Shown in EER diagrams by a double line
  - Partial allows an entity not to belong to any of the subclasses
  - Shown in EER diagrams by a single line
- Hence, we have four types of specialization/generalization:
  - Disjoint, total
  - Disjoint, partial
  - Overlapping, total
  - Overlapping, partial
- Note: Generalization usually is total because the superclass is derived from the subclasses.

  Example of disjoint partial Specialization

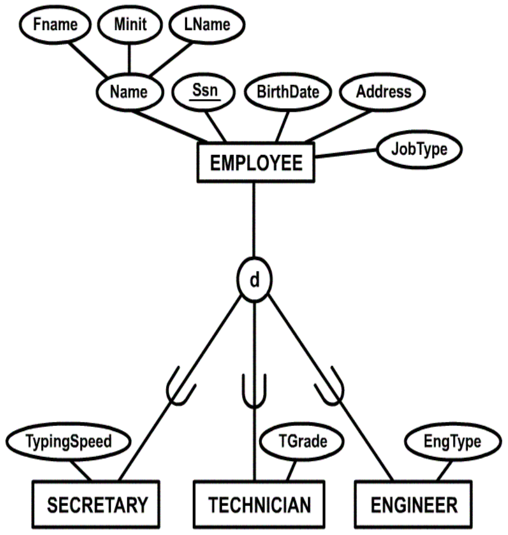


Figure. Disjoint partial Specialization

Specialization / Generalization Hierarchies, Lattices and Shared Subclasses

- A subclass may itself have further subclasses specified on it Forms a hierarchy or a lattice
- Hierarchy has a constraint that every subclass has only one superclass (called single inheritance)
- In a lattice, a subclass can be subclass of more than one superclass (called multiple inheritance)
- In a lattice or hierarchy, a subclass inherits attributes not only of its direct superclass, but also of all its predecessor superclasses
- A subclass with more than one superclass is called a shared subclass
- Can have specialization hierarchies or lattices, or generalization hierarchies or lattices
- In specialization, start with an entity type and then define subclasses of the entity type by successive specialization (top down conceptual refinement process)
- In generalization, start with many entity types and generalize those that have common properties (bottom up conceptual synthesis process)
- In practice, the combination of two processes is employed

Specialization / Generalization Lattice Example (UNIVERSITY)

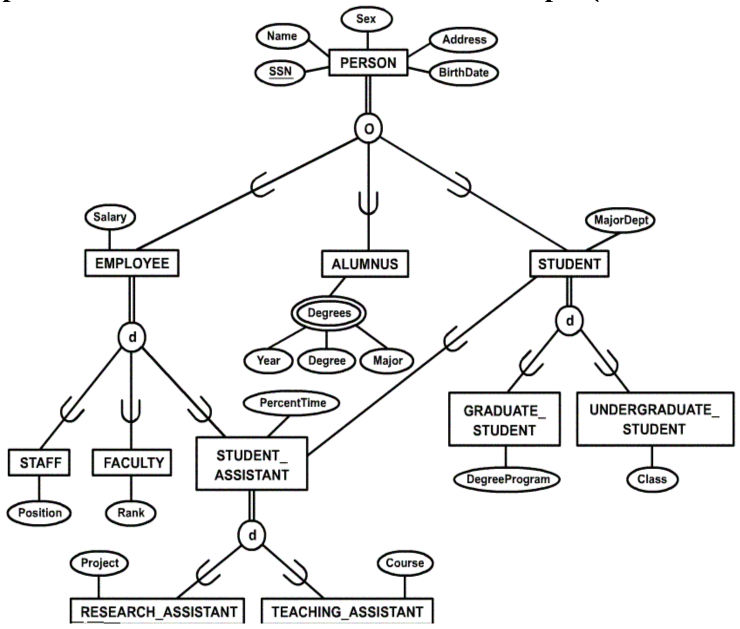


Figure. **Specialization / Generalization**

<u>Modeling of UNION types using categories:</u>

- All of the superclass/subclass relationships we have seen thus far have a single superclass
- A shared subclass is subclass in more than one distinct superclass/subclass relationships, where each relationships has a single superclass (multiple inheritance)
- In some cases, need to model a single superclass/subclass relationship with more than one superclass
- Superclasses represent different entity types
- Such a subclass is called a category or UNION TYPE
- Example: Database for vehicle registration, vehicle owner can be a person, a bank (holding a lien on a vehicle) or a company.
  - Category (subclass) OWNER is a subset of the union of the three superclasses COMPANY, BANK, and PERSON
  - A category member must exist in at least one of its superclasses
- Note: The difference from shared subclass, which is subset of the intersection of its superclasses (shared subclass member must exist in all of its superclasses).

Example of categories (UNION TYPES)

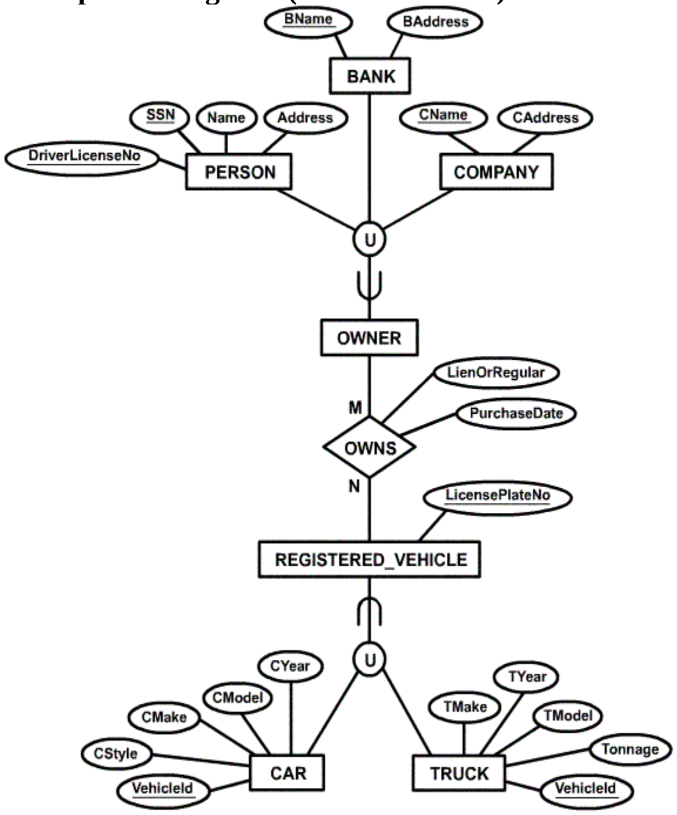
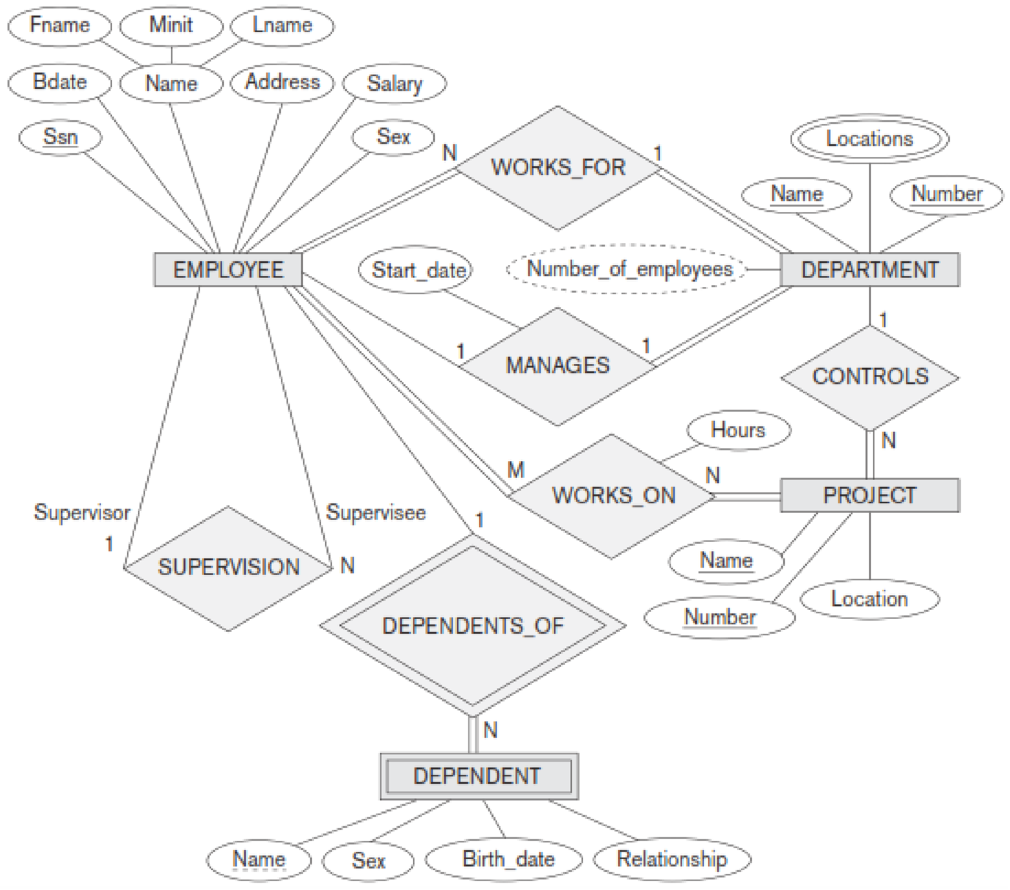


Figure. **categories (UNION TYPES)**

2.3. ER-TO-RELATIONAL MAPPING

Figure . The ER conceptual schema diagram for the COMPANY database.



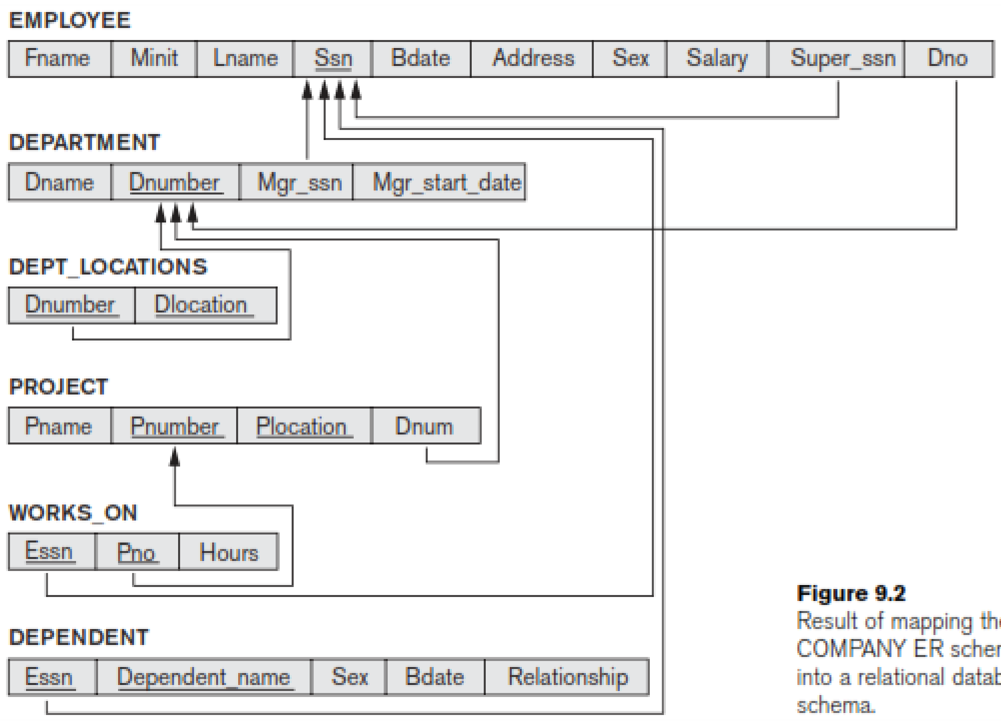


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

Seven-step algorithm to convert the basic ER model constructs into relations :

Step 1: Mapping of Regular Entity Types:

- For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as the primary key for R. If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R.

- If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R. Knowledge about keys is also kept for index-ing purposes and other types of analyses.

- In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT in Figure . The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include the attributes Super_ssn and Dno of EMPLOYEE, Mgr_ssn and Mgr_start_date of DEPARTMENT, and Dnum of PROJECT. In our example, we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively. Knowledge that Dname of DEPARTMENT and Pname of PROJECT are secondary keys is kept for possible use later in the design.

- The relations that are created from the mapping of entity types are sometimes called **entity relations** because each tuple represents an entity instance.

Step 2: Mapping of Weak Entity Types:

- For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R. In addition, include as foreign key attributes of R, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of W. The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

- If there is a weak entity type E_2 whose owner is also a weak entity type E_1, then E_1 should be mapped before E_2 to determine its primary key first.

- In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT . We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it Essn, although this is not necessary.

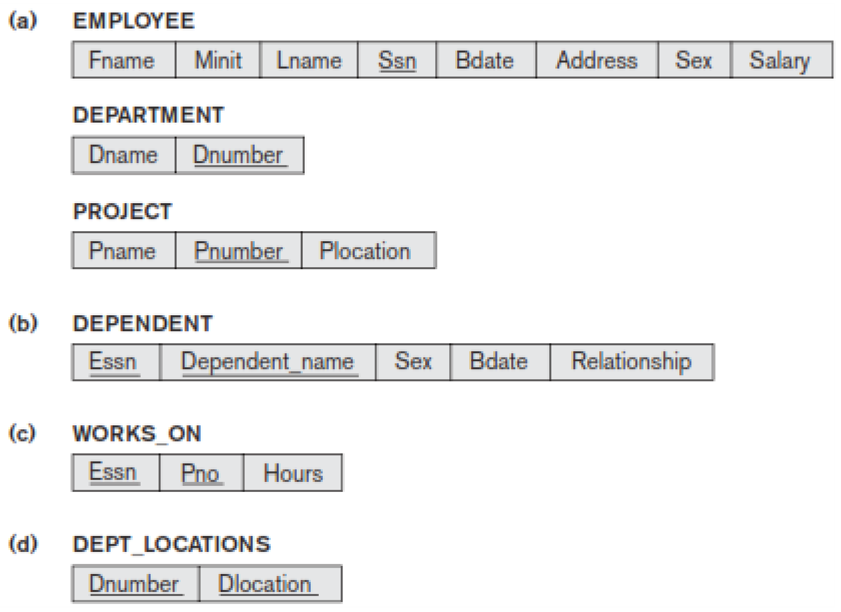
Figure . Illustration of some mapping steps.

(a) Entity relations after step 1.

(b) Additional weak entity relation after step 2.

(c) Relationship relation after step 5.

(d) Relation representing multivalued attribute after step 6.



(a) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary

DEPARTMENT

Dname	Dnumber

PROJECT

Pname	Pnumber	Plocation

(b) DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

(c) WORKS_ON

Essn	Pno	Hours

(d) DEPT_LOCATIONS

Dnumber	Dlocation

The primary key of the DEPENDENT relation is the combination {Essn, Dependent_name}, because Dependent_name is the partial key of DEPENDENT.

It is common to choose the propagate (CASCADE) option for the referential triggered action on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity. This can be used for both ON UPDATE and ON DELETE.

Step 3: Mapping of Binary 1:1 Relationship Types:

For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. There are three possible approaches:

(1) Foreign key approach

(2) Merged relationship approach

(3) Cross-reference or relationship relation approach.

The first approach is the most useful and should be followed unless special conditions exist.

(1) Foreign key approach:
Choose one of the relations—S, say—and include as a foreign key in S the primary key of T. It is better to choose an entity type with total participation in R in the role of S. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.

In our example, we map the 1:1 relationship type MANAGES from Figure by choosing the participating entity type DEPARTMENT to serve in the role of S because its participation in the MANAGES relationship type is total (every department has a manager). We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it Mgr_ssn. We also include the simple attribute Start_date of the MANAGES relationship type in the DEPARTMENT relation and rename it Mgr_start_date.

Note that it is possible to include the primary key of S as a foreign key in T instead. In our example, this amounts to having a foreign key attribute, say Department_managed in the EMPLOYEE relation, but it will have a NULL value for employee tuples who do not manage a department. If only 2 percent of employees manage a department, then 98 percent of the foreign keys would be NULL in this case. Another possibility is to have foreign keys in both relations S and T redundantly, but this creates redundancy and incurs a penalty for consistency maintenance.

(2)Merged relation approach:

An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single rela-tion. This is possible when both participations are total, as this would indicate that the two tables will have the exact same number of tuples at all times.

(3)Cross-reference or relationship relation approach:

The third option is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we will see, this approach is required for binary M:N relationships. The relation R is called a **relationship relation** (or sometimes a **lookup table**), because eachtuple in R represents a relationship instance that relates one tuple from S with one tuple from T. The relation R will include the primary key attributes of S and T as foreign keys to S and T. The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R. The drawback is having an extra relation, and requiring an extra join operation when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types:

For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the N-side of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S.

In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure 9.1. For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn. The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation. .

An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation R whose attributes are the primary keys of S and T, which will also be foreign keys to S and T. The primary key of R is the same as the primary key of S. This option can be used if few tuples in S participate in the relationship to avoid excessive NULL val-ues in the foreign key.

Step 5: Mapping of Binary M:N Relationship Types.

For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any sim-ple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must cre-ate a separate relationship relation S.

In our example, we map the M:N relationship type WORKS_ON from Figure by creating the relation WORKS_ON in Figure 9.2. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively. We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}.

The propagate (CASCADE) option for the referential triggered action should be specified on the foreign keys in the relation corresponding to the relationship R, since each relationship instance has an existence dependency on each of the entities it relates. This can be used for both ON UPDATE and ON DELETE.

Notice that we can always map 1:1 or 1:N relationships in a manner similar to M:N relationships by using the cross-reference (relationship relation) approach, as we discussed earlier. This alternative is particularly useful when few relationship instances exist, in order to avoid NULL values in foreign keys. In this case, the primary key of the relationship relation will be only one of the foreign keys that refer-ence the participating entity relations. For a 1:N relationship, the primary key of the relationship relation will be the foreign key that references the entity relation on the N-side. For a 1:1 relationship, either foreign key can be used as the primary key of the relationship relation.

Step 6: Mapping of Multivalued Attributes:

For each multivalued attribute A, create a new relation R. This relation R will include an attribute corresponding to A, plus the primary key attribute K—as a foreign key in R—of the relation that represents the entity type or relationship type that has A as a multivalued attribute. The primary key of R is the combination of A and K. If the multivalued attribute is com-posite, we include its simple components.

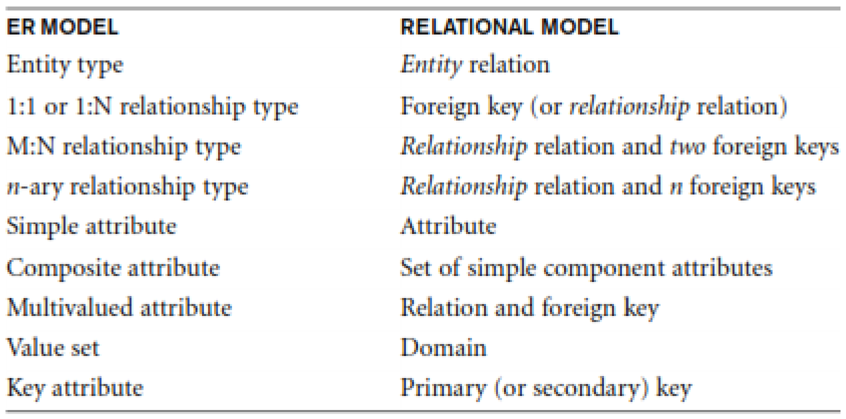
In our example, we create a relation DEPT_LOCATIONS (see Figure 9.3(d)). The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, while Dnumber—as foreign key—represents the primary key of the DEPARTMENT relation. The primary key of DEPT_LOCATIONS is the combination of {Dnumber, Dlocation}. A separate tuple will exist in DEPT_LOCATIONS for each loca-tion that a department has.

The propagate (CASCADE) option for the referential triggered action should be specified on the foreign key in the relation R corresponding to the multivalued attribute for both ON UPDATE and ON DELETE. We should also note that the key of R when mapping a composite, multivalued attribute requires some analysis of the meaning of the component attributes. In some cases, when a multi-valued attribute is composite, only some of the component attributes are required to be part of the key of R; these attributes are similar to a partial key of a weak entity type that corresponds to the multivalued attribute .

Step 7: Mapping of N-ary Relationship Types:

For each n-ary relationship type R, where n > 2, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S. The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E corresponding to E.

Table . Correspondence between ER and Relational Models



ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

<u>2.4. FUNCTIONAL DEPENDENCIES</u>:

- A functional dependency is a constraint between two sets of attributes from the database.
- A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

- This means that the values of the Y component of a tuple in r depend on, or are determined by, the values of the X component; alternatively, the values of the X com-ponent of a tuple uniquely (or **functionally**) determine the values of the Y component. We also say that there is a functional dependency from X to Y, or that Y is **functionally dependent** on X. The abbreviation for functional dependency is **FD** or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.

- Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of r(R) agree on their X-value, they must necessarily agree on their Y-value. Note the following:

- If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance r(R)—that is, X is a **candidate key** of R—this implies that X → Y for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state r(R) will have the same value of X). If X is a candidate key of R, then X → R.

If X → Y in R, this does not say whether or not Y → X in R.

- A functional dependency is a property of the **semantics** or **meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of R—that is, how they relate to one another—to specify the functional dependencies that should hold on all relation states (extensions) r of R. Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint. Relation extensions r(R) that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R.
- Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold at all times. Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, {State, Driver_license_number} → Ssn should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation. It is also possible that certain functional dependencies may cease to exist in the real world if the relationship changes. For example, the FD Zip_code → Area_code used to exist as a relationship between postal codes and telephone num-ber codes in the United States, but with the proliferation of telephone area codes it is no longer true.
- Consider the relation schema EMP_PROJ. From the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

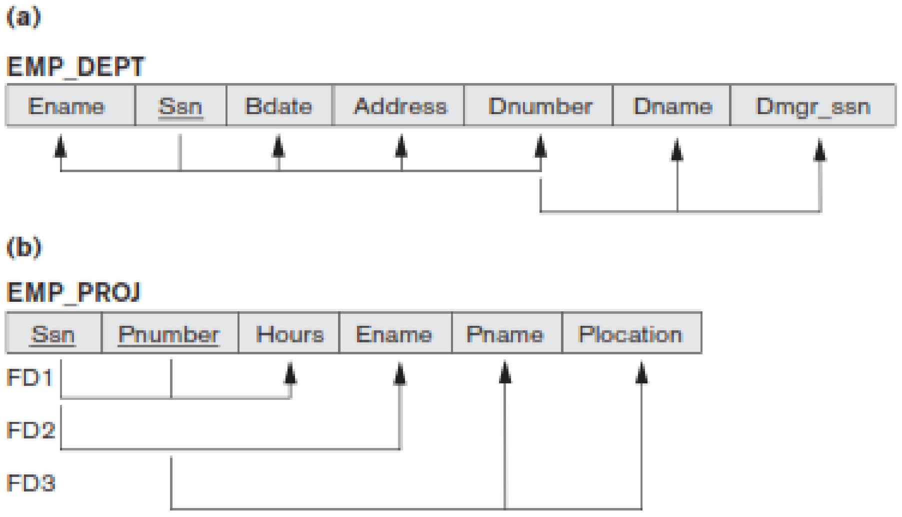
a. Ssn → Ename

b. Pnumber →{Pname, Plocation}

c. {Ssn, Pnumber} → Hours

- These functional dependencies specify that (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename), (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours). Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or given a value of Ssn, we know the value of Ename, and so on.
- A functional dependency is a property of the relation schema R, not of a particular legal relation state r of R. Therefore, an FD cannot be inferred automatically from a given relation extension r but must be defined explicitly by someone who knows the semantics of the attributes of R.

- Given a populated relation, one cannot determine which FDs hold and which do not unless the meaning of and the relationships among the attributes are known. All one can say is that a certain FD may exist if it holds in that particular extension. One cannot guarantee its existence until the meaning of the corresponding attributes is clearly understood. One can, however, emphatically state that a certain FD does not hold if there are tuples that show the violation of such an FD.



- Figure introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, while the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes. We denote by F the set of functional dependencies that are specified on relation schema R.

2.5. NON LOSS DECOMPOSITION (OR) LOSSLESS DECOMPOSITION :

- The decompositio of relation R into R1 and R2 is **lossless** when the join of R1 and R2 yield the same relation as in R.
- A relational table is decomposed (or factored) into two or more smaller tables, in such a way that the designer can capture the precise content of the original table by joining the decomposed parts. This is called lossless-join (or non-additive join) decomposition.
- This is also referred as non-additive decomposition.
- The lossless-join decomposition is always defined with respect to a specific set F of dependencies.

Example:

<EmpInfo>

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables:

<EmpDetails>

Emp_ID	Emp_Name	Emp_Age	Emp_Location
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	Emp_ID	Dept_Name
Dpt1	E001	Operations
Dpt2	E002	HR
Dpt3	E003	Finance

Now, Natural Join is applied on the above two tables:

The result will be:

Emp_ID	Emp_Name	Emp_Age	Emp_Location	Dept_ID	Dept_Name
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Therefore, the above relation had lossless decomposition i.e. no loss of information.

Lossy Decomposition:

- When a relation is decomposed into two or more relational schemas, the loss of information is unavoidable when the original relation is retrieved.

Example:

<EmpInfo>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**	**Dept_ID**	**Dept_Name**
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables:

<EmpDetails>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	**Dept_Name**
Dpt1	Operations
Dpt2	HR
Dpt3	Finance

Now, you won't be able to join the above tables, since **Emp_ID** isn't part of the **DeptDetails** relation. Therefore, the above relation has lossy decomposition.

2.6. NORMAL FORMS:

- A database schema consists of a number of relation schemas. The attributes are grouped to form a relation schema by using the common sense of the database designer or by mapping a database schema design from a conceptual data model such as the ER or Enhanced-ER (EER) data model.
- These models make the designer identify entity types and relationship types and their respective attributes, which leads to a natural and logical grouping of the attributes into relations when the mapping procedures are followed. However, we still need some formal way of analyzing why one grouping of attributes into a relation schema may be better than another.

There are two levels at which we can discuss the goodness of relation schemas.

1) The first is the **logical** (or **conceptual**) **level**—how users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables users to understand clearly the meaning of the data in the relations, and hence to formulate their queries correctly.
2) The second is the **implementation** (or **physical storage**) **level**—how the tuples in a base relation are stored and updated. This level applies only to schemas of base relations—which will be physically stored as files—whereas at the logical level we are interested in schemas of both base relations and views (virtual relations).

Database design may be performed using two approaches:

1) A **bottom-up design methodology** (also called design by synthesis) considers the basic relationships among individual attrib-utes as the starting point and uses those to construct relation schemas. This approach is not very popular in practice because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point. For practical situations, it is next to impossible to capture binary relationships among all such pairs of attributes.

2) In contrast, a **top-down design methodology** (also called design by analysis) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report. The relations are then analyzed individually and collectively, lead-ing to further decomposition until all desirable properties are met. The theory described in this chapter is applicable to both the top-down and bottom-up design approaches, but is more appropriate when used with the top-down approach.

- Relational database design ultimately produces a set of relations. The implicit goals of the design activity are information preservation and minimum redundancy. Information is very hard to quantify—hence we consider information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships.
- Thus, the relational design must preserve all of these concepts, which are originally captured in the conceptual design after the conceptual to logical design mapping. Minimizing redundancy implies minimizing redundant storage of the same information and reducing the need for multiple updates to maintain consistency across multiple copies of the same information in response to real-world events that require making an update.

- Four informal guidelines that may be used as measures to determine the quality of relation schema design:

✓ Making sure that the semantics of the attributes is clear in the schema
✓ Reducing the redundant information in tuples
✓ Reducing the NULL values in tuples
✓ Disallowing the possibility of generating spurious tuples

Normalization of Relations:

- The normalization process is first proposed by Codd (1972)
- The **normalization process** takes a relation schema through a series of tests to certify whether it satisfies a certain **normal form**. The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as relational design by analysis.
- Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively.

- **Normalization of data** can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of

1) minimizing redundancy
2) minimizing the insertion, deletion, and update anomalies

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.

- To overcome these anomalies we need to normalize the data.
- It can be considered as a "filtering" or "purification" process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database design-ers with the following:

a. A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
b. A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been nor-malized.

Normal forms, when considered in isolation from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

1) The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.
2) The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

- The nonadditive join property is extremely critical and **must be achieved at any cost**, whereas the dependency preservation property, although desirable, is some-times sacrificed.

- **Denormalization** is the process of storing the join of higher nor-mal form relations as a base relation, which is in a lower normal form.
- A **superkey** of a relation schema R = {A_1, A_2, ... , A_n} is a set of attributes S ⊆ R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.
- The difference between a key and a superkey is that a key has to be minimal;
- If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is arbitrarily designated to be the **primary key**, and the others are called secondary keys.
- An attribute of relation schema R is called a **prime attribute** of R if it is a member of some candidate key of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

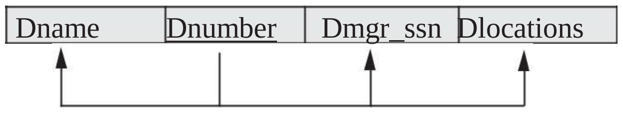
2.7. FIRST NORMAL FORM (1NF)

- First normal form states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute.
- The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.
- Consider the DEPARTMENT relation schema shown in Figure We assume that each department can have a number of locations. This is not in 1NF because Dlocations is not an atomic attribute. There are two ways we can look at the Dlocations attribute:

1) The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.
2) The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber → Dlocations because each set is considered a single member of the attribute domain.

a)

DEPARTMENT



(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bangalore, Chennai, Delhi}
Administration	4	987654321	{ Chennai }
Headquarters	1	888665555	{ Delhi }

(c) DEPARTMENT

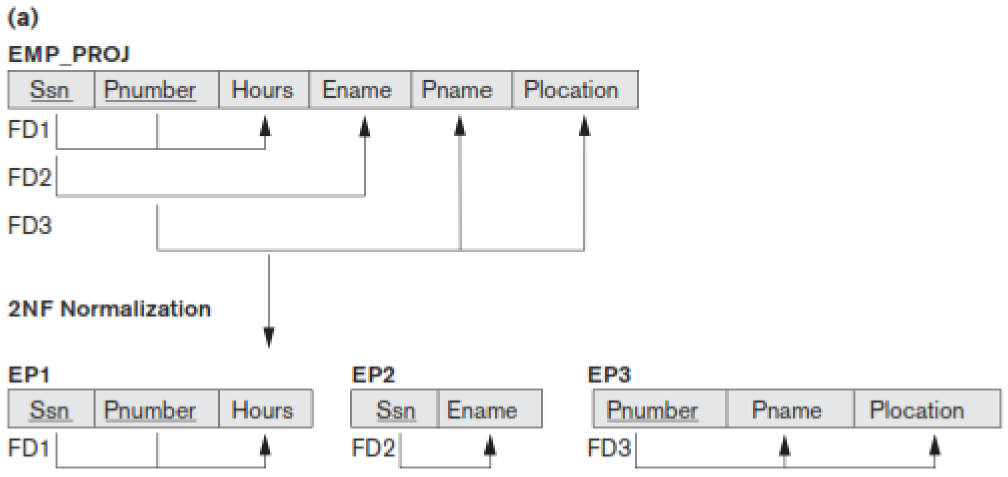
Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bangalore
Research	5	333445555	Chennai
Research	5	333445555	Delhi
Administration	4	987654321	Chennai
Headquarters	1	888665555	Delhi

Figure. Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

2.8. SECOND NORMAL FORM:

- **Second normal form (2NF)** is based on the concept of full functional dependency.
- A functional dependency X → Y is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute A ε X, (X – {A}) does not functionally determine Y.
- A functional dependency X → Y is a **partial dependency** if some attribute A ε X can be removed from X and the dependency still holds; that is, for some A ε X, (X – {A}) → Y.
- In Figure, {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber → Hours holds). However, the dependency {Ssn, Pnumber} → Ename is partial because Ssn → Ename holds.

Definition. A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.



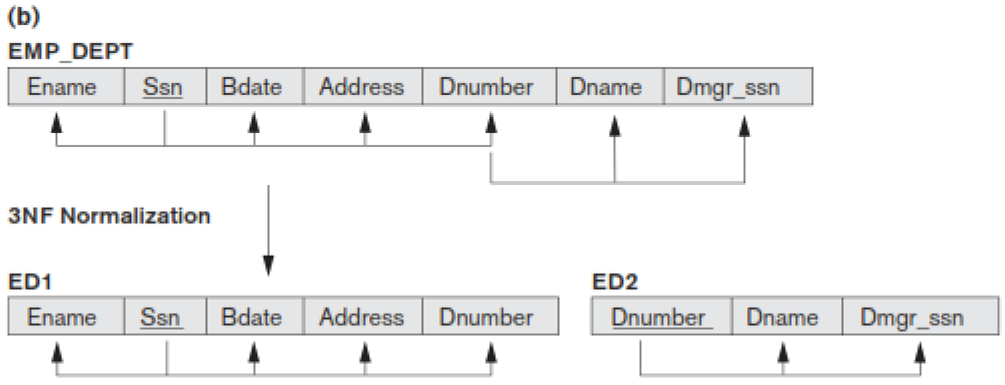
- The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure is in 1NF but is not in 2NF.
- The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.

- If a relation schema is not in 2NF, it can be second normalized or 2NF normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

2.9. THIRD NORMAL FORM:

- **Third normal form (3NF)** is based on the concept of transitive dependency. A functional dependency X → Y in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, and both X → Z and Z → Y hold. The dependency Ssn → Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure 15.3(a), because both the dependencies Ssn → Dnumber and Dnumber → Dmgr_ssn hold and Dnumber is nei-ther a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of Dmgr_ssn on Dnumber is undesirable in since Dnumber is not a key of EMP_DEPT.

Definition. A relation schema R is in **3NF** if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key.



- The relation schema EMP_DEPT in Figure 15.3(a) is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure. Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.
- Intuitively, we can see that any functional dependency in which the left-hand side is part (a proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute, is a problematic FD.

- 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF.

2.10. BOYCE-CODD NORMAL FORM (BCNF):

- **Boyce-Codd normal form (BCNF)** was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

2.11. MULTIVALUED DEPENDENCY AND FOURTH NORMAL FORM

Definition. A multivalued dependency X →→ Y specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote $(R – (X \cup Y))$

$t_3[X] = t_4[X] = t_1[X] = t_2[X]$.

$t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.

$t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

- Whenever X →→ Y holds, we say that X **multidetermines** Y. Because of the symmetry in the definition, whenever X →→ Y holds in R, so does X →→ Z. Hence, X →→ Y implies X →→ Z, and therefore it is sometimes written as X →→ Y|Z.

- An MVD X →→ Y in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) X ∪ Y=R.

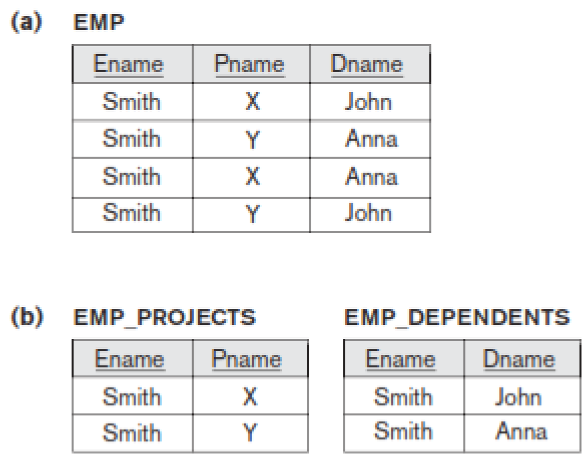
- For example, the relation EMP_PROJECTS in Figure 15.15(b) has the trivial MVD Ename →→ Pname. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. A trivial MVD will hold in any relation state r of R; it is called trivial because it does not specify any significant or meaningful constraint on R.

- If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure(a), the values ‘X’ and ‘Y’ of Pname are repeated with each value of Dname (or, by symmetry, the values ‘John’ and ‘Anna’ of Dname are repeated with each value of Pname). This redundancy is clearly undesirable. However, the EMP schema is in BCNF because no functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP.

- Notice that relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together. Furthermore, it is rare that such all-key relations with a combinatorial occurrence of repeated values would be designed in practice. However, recognition of MVDs as a potential problematic dependency is essential in relational design.

Definition. A relation schema R is in 4**NF** with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every nontrivial multivalued dependency X →→ Y in F, X is a superkey for R.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD.



(a) EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

Figure . (a)The EMP relation with two MVDs: Ename →→ Pname and Ename →→ Dname.

(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

- Consider the EMP relation in Figure(a). EMP is not in 4NF because in the nontrivial MVDs Ename →→ Pname and Ename→→Dname, and Ename is not a superkey of EMP.
- We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure(b). Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs Ename→→Pname in EMP_PROJECTS and Ename →→ Dname in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

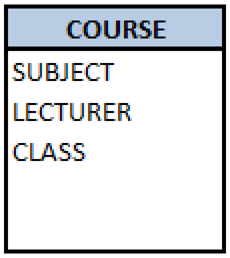
<u>2.12. JOIN DEPENDENCIES AND FIFTH NORMAL FORM</u>

Definition. A join dependency (JD) can be said to exist if the join of R_1 and R_2 over C is equal to relation R. Where, R_1 and R_2 are the decompositions R_1(A, B, C), and R_2 (C,D) of a given relations R (A, B, C, D). Alternatively, R_1 and R_2 is a lossless decomposition of R.

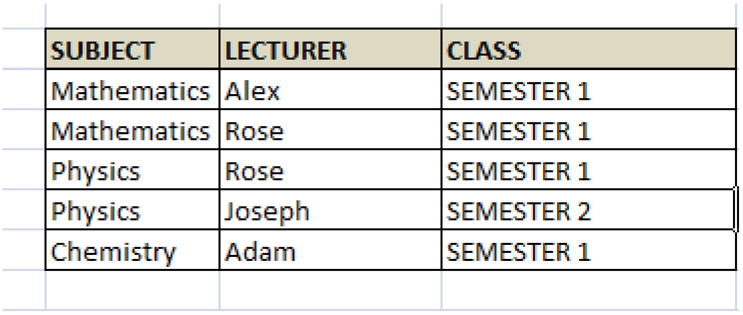
Definition Of fifth normal form, which is also called project-join normal form.

A database is said to be in 5NF, if and only if,

- It's in 4NF
- If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise. In simple words, joining two or more decomposed table should not lose records nor create new records.
- Consider an example of different Subjects taught by different lecturers and the lecturers taking classes for different semesters.
- **Note**: Please consider that Semester 1 has Mathematics, Physics and Chemistry and Semester 2 has only Mathematics in its academic year!!

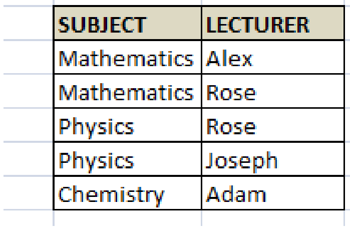


COURSE
SUBJECT
LECTURER
CLASS

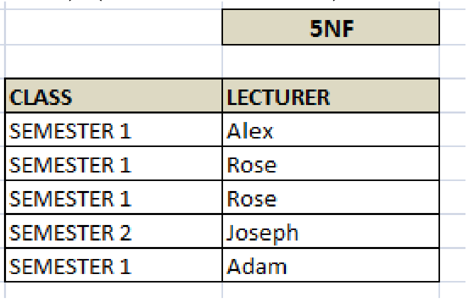


SUBJECT	LECTURER	CLASS
Mathematics	Alex	SEMESTER 1
Mathematics	Rose	SEMESTER 1
Physics	Rose	SEMESTER 1
Physics	Joseph	SEMESTER 2
Chemistry	Adam	SEMESTER 1

- In above table, Rose takes both Mathematics and Physics class for Semester 1, but she does not take Physics class for Semester 2. In this case, combination of all these 3 fields is required to identify a valid data. Imagine we want to add a new class - Semester3 but do not know which Subject and who will be taking that subject. We would be simply inserting a new entry with Class as Semester3 and leaving Lecturer and subject as NULL. As we discussed above, it's not a good to have such entries. Moreover, all the three columns together act as a primary key, we cannot leave other two columns blank!
- Hence we have to decompose the table in such a way that it satisfies all the rules till 4NF and when join them by using keys, it should yield correct record. Here, we can represent each lecturer's Subject area and their classes in a better way. We can divide above table into three - (SUBJECT, LECTURER), (LECTURER, CLASS), (SUBJECT, CLASS)

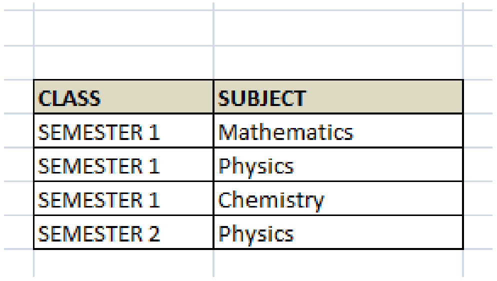


SUBJECT	LECTURER
Mathematics	Alex
Mathematics	Rose
Physics	Rose
Physics	Joseph
Chemistry	Adam



5NF

CLASS	LECTURER
SEMESTER 1	Alex
SEMESTER 1	Rose
SEMESTER 1	Rose
SEMESTER 2	Joseph
SEMESTER 1	Adam



CLASS	SUBJECT
SEMESTER 1	Mathematics
SEMESTER 1	Physics
SEMESTER 1	Chemistry
SEMESTER 2	Physics

- Now, each of combinations is in three different tables. If we need to identify who is teaching which subject to which semester, we need join the keys of each table and get the result.
- For example, who teaches Physics to Semester 1, we would be selecting Physics and Semester1 from table 3 above, join with table1 using Subject to filter out the lecturer names. Then join with table2 using Lecturer to get correct lecturer name. That is we joined key columns of each table to get the correct data. Hence there is no lose or new data - satisfying 5NF condition.

```
SELECT t3.Class, t3.Subject, t1.Lecturer FROM TABLE3 t3, TABLE3 t2, TABLE3 t1,
where t3.Class = 'SEMESTER1' and t3.SUBJECT= 'PHYSICS' AND  t3.Subject = t1.Subject
AND t3.Class = t2.Class  AND t1.Lecturer = t2.Lecturer;
```

UNIT III TRANSACTIONS

Transaction Concepts – ACID Properties – Schedules – Serializability – Concurrency Control – Need for Concurrency – Locking Protocols – Two Phase Locking – Deadlock – Transaction Recovery - Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery.

3.1. TRANSACTION CONCEPTS:

- A **transaction** is a **unit** of program execution that accesses and possibly updates various data items.
- A transaction is delimited by statements of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.
- This collection of steps must appear to the user as a single, indivisible unit. Since a transaction is indivisible, it either executes in its entirety or not at all. Thus, if a transaction begins to execute but fails for whatever reason, any changes to the database that the transaction may have made must be undone.
- This requirement holds regardless of whether the transaction itself failed , the operating system crashed, or the computer itself stopped operating.

3.2. ACID Properties:

Properties of the transactions:

1) Atomicity. Either all operations of the transaction are reflected properly in the database, or none are.

2) Consistency. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the data-base.

3) Isolation. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

4) Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

A Simple Transaction Model:

- Consider a simple bank application consisting of several accounts and a set of transactions that access and update those accounts.
- Transactions access data using two operations:

- read(*X*), which transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory belonging to the transaction that executed the read operation.
- write(*X*), which transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the write to the data item *X* in the database.
- It is important to know if a change to a data item appears only in main memory or if it has been written to the database on disk. In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored elsewhere and executed on the disk later. For now, however, we shall assume that the write operation updates the database immediately.

Let T_i be a transaction that transfers \$50 from account *A* to account *B*. This transaction can be defined as:

```
T_i : read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B).
```

Let us now consider each of the ACID properties.

Consistency: The consistency requirement here is that the sum of *A* and *B* be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints.

Atomicity: Suppose that, just before the execution of transaction T_i , the values of accounts *A* and *B* are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction T_i , a failure occurs that prevents T_i from completing its execution successfully. Further, suppose that the failure happened after the write(*A*) operation but before the write(*B*) operation. In this case, the values of accounts *A* and *B* reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction T_i is executed to completion, there exists a point at which the value of account *A* is \$950 and the value of account *B* is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account *A* is $950, and the value of account *B* is $2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write. This information is written to a file called the *log*. If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed. Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system**.

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of

funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.
We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either:

The updates carried out by the transaction have been written to disk before the transaction completes.

Information about the updates carried out by the transaction and writ-ten to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

The **recovery system** of the database is responsible for ensuring durability, in addition to ensuring atomicity.

Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from *A* to *B* is executing, with the deducted total written to *A* and the increased total yet to be written to *B*. If a second concurrently running transaction reads *A* and *B* at this intermediate point and computes *A*+ *B*, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on *A* and *B* based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concur-rent execution of transactions provides significant performance benefits, as we shall see in Section 14.5. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently.

The isolation property of a transaction ensures that the con-current execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system.**

<u>STATES OF TRANSACTION:</u>

- A transaction may not always complete its execution successfully. Such a transaction is termed **aborted**.
- If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone.
- Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**.
- It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a **log**.
- Each database modification made by a transaction is first recorded in the log. We record the identifier of the transaction performing the modification, the identifier of the data item being modified, and both the old value (prior to modification) and the new value (after modification) of the data item.
- Only then is the database itself modified. Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution.

- A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

- Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account.
- However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system. Chapter 26 includes a discussion of compensating transactions.

- We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model.

- A transaction must be in one of the following states:

1) Active, the initial state; the transaction stays in this state while it is executing.
2) Partially committed, after the final statement has been executed.
3) Failed, after the discovery that normal execution can no longer proceed.
4) Aborted, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
5) Committed, after successful completion.We say that a transaction has committed only if it has entered the committed state.

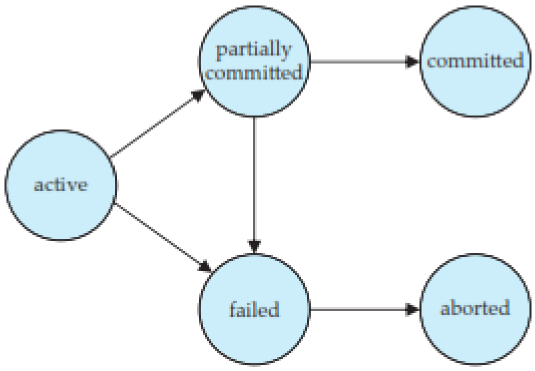


Figure. State diagram of a transaction.

- Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if it has either committed or aborted.

- A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

- The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

- A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

- We must be cautious when dealing with **observable external writes**, such as writes to a user's screen, or sending email. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system.
- Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in a special relation in the database, and to perform the actual writes only after the transaction enters the committed state.
- If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.

- Handling external writes can be more complicated in some situations. For example, suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically).

- It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the user's account, needs to be executed when the system is restarted.

- As another example, consider a user making a booking over the Web. It is possible that the database system or the application server crashes just after the booking transaction commits. It is also possible that the network connection to the user is lost just after the booking transaction commits.

- In either case, even though the transaction has committed, the external write has not taken place. To handle such situations, the application must be designed such that when the user connects to the Web application again, she will be able to see whether her transaction had succeeded or not.

- For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity.

3.3. SCHEDULES:

- Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed.

Two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.
- **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Con-current execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

- The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.

- When several transactions run concurrently, the isolation property may be vi-olated, resulting in database consistency being destroyed despite the correctness of each individual transaction. In this section, we present the concept of sched-ules to help identify those executions that are guaranteed to ensure the isolation property and thus database consistency.

- The database system must control the interaction among the concurrent trans-actions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control schemes**.

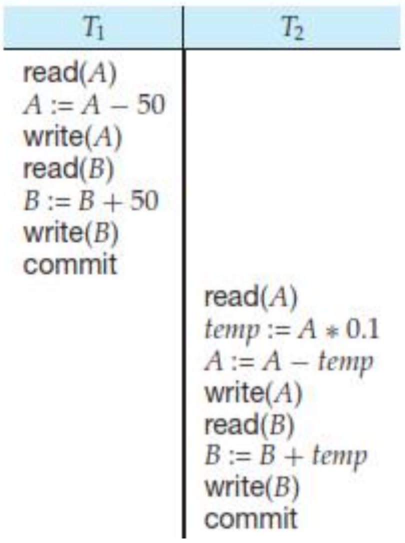
- Consider again the simplified banking system which has several accounts, and a set of transactions that access and update those accounts.
- Let T_1 and T_2 be two transactions that transfer funds from one account to another.
- Transaction T_1 transfers \$50 from account A to account B. It is defined as:

```
T1: read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B).
```

- Transaction T_2 transfers 10 percent of the balance from account A to account B. It is defined as:

```
T2: read(A);
temp := A * 0.1;
A := A − temp;
write(A);
read(B);
B := B + temp;
write(B).
```

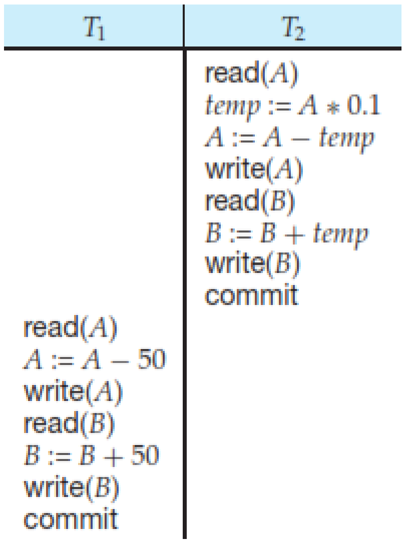
- Suppose the current values of accounts A and B are \$1000 and \$2000, respectively. Suppose also that the two transactions are executed one at a time in the order T_1 followed by T_2. This execution sequence appears in Figure. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_1 appearing in the left column and instructions of T_2 appearing in the right column. The final values of accounts A and B, after the execution in Figure takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in accounts A and B —that is, the sum $A + B$ —is preserved after the execution of both transactions.



T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure. Schedule 1 — a serial schedule in which T_1 is followed by T_2.

Similarly, if the transactions are executed one at a time in the order T_2 followed by T_1, then the corresponding execution sequence is that of Figure 14.3. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively.



T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Figure . Schedule 2 — a serial schedule in which T_2 is followed by T_1.

- The execution sequences just described are called **schedules**.
- Schedules represent the chronological order in which instructions are executed in the system.

- A schedule for a set of transactions must consist of all instructions of those trans-actions, and must preserve the order in which the instructions appear in each individual transaction. For example, in transaction T_1, the instruction write(A) must appear before the instruction read(B), in any valid schedule.
- Note that we include in our schedules the **commit** operation to indicate that the transaction has entered the committed state. In the following discussion, we shall refer to the first execution sequence (T_1 followed by T_2) as schedule 1, and to the second execution sequence (T_2 followed by T_1) as schedule 2.
- These schedules are **serial**:
- Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

- When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

- Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.[1]

- Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure. After this execution takes place, we arrive at the same state as the one in which the transac-tions are executed serially in the order T_1 followed by T_2. The sum $A + B$ is indeed preserved.

- Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are \$950 and \$2100, respectively. This final state is an *inconsistent state,* since we have gained \$50 in the process of the concurrent execution. Indeed, the sum $A + B$ is not preserved by the execution of the two transactions.

- If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure . Schedule 3 — a concurrent schedule equivalent to schedule 1.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Figure. Schedule 4 — a concurrent schedule resulting in an inconsistent state.

3.4. SERIALIZABILITY:

- Serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable.
- Since transactions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact.
- For this reason, we shall not consider the various types of operations that a transaction can perform on a data item, but instead consider only two operations: read and write.
- We assume that, between a read(*Q*) instruction and a write(*Q*) instruction on a data item *Q*, a transaction may perform an arbitrary sequence of operations on the copy of *Q* that is residing in the local buffer of the transaction. In this model, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. Commit operations, though relevant, are not considered. We therefore may show only read and write instructions in schedules, as we do for schedule 3 in Figure..

Different forms of schedule serializablityare

1) **conflict serializability**.
2) View serialzablity

1. CONFLICT SERIALIZABILITY:

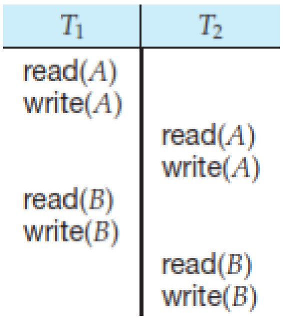
- Let us consider a schedule *S* in which there are two consecutive instructions, and *J*, of transactions T_i and T_j , respectively ($i \;= j$). If *I* and *J* refer to different data items, then we can swap *I* and *J* without affecting the results of any instruction in the schedule. However, if *I* and *J* refer to the same data item *Q*, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

  a. *I* = read(*Q*), *J* = read(*Q*). The order of *I* and *J* does not matter, since the same value of *Q* is read by T_i and T_j , regardless of the order.

  b. *I* = read(*Q*), *J* = write(*Q*). If *I* comes before *J* , then T_i does not read the value of *Q* that is written by T_j in instruction *J* . If *J* comes before *I* , then T_i reads the value of *Q* that is written by T_j . Thus, the order of *I* and *J* matters.

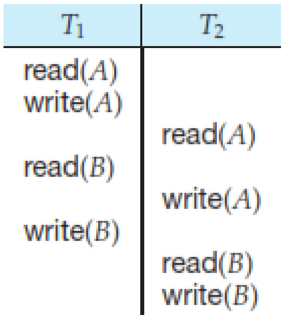
  c. *I* = write(*Q*), *J* = read(*Q*). The order of *I* and *J* matters for reasons similar to those of the previous case.

  d. *I* = write(*Q*), *J* = write(*Q*). Since both instructions are write operations, the order of these instructions does not affect either T_i or T_j . However, the value obtained by the next read(*Q*) instruction of *S* is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(*Q*) instruction after *I* and *J* in *S*, then the order of *I* and J directly affects the final value of Q in the database state that results from schedule S.



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure . Schedule 3 — showing only the read and write instructions.



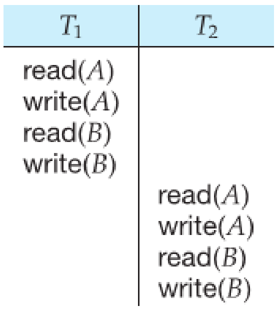
T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Figure . Schedule 5 — schedule 3 after swapping of a pair of instructions.

- Thus, only in the case where both I and J are read instructions does the relative order of their execution not matter.

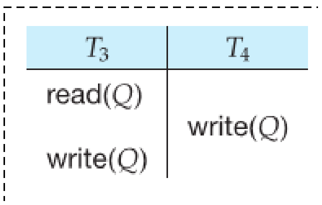
- We say that I and J **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

- To illustrate the concept of conflicting instructions, we consider schedule 3in Figure 14.6. The write(A) instruction of T_1 conflicts with the read(A) instruction of T_2. However, the write(A) instruction of T_2 does not conflict with the read(B) instruction of T_1, because the two instructions access different data items.



T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure . Schedule 6 — a serial schedule that is equivalent to schedule 3.



T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Figure. Schedule 7.

- Let *I* and *J* be consecutive instructions of a schedule *S*. If *I* and *J* are instructions of different transactions and *I* and *J* do not conflict, then we can swap the order of *I* and *J* to produce a new schedule *S* . *S* is equivalent to *S*, since all instructions appear in the same order in both schedules except for *I* and *J* , whose order does not matter.

- Since the write(*A*) instruction of T_2 in schedule 3 does not conflict with the read(*B*) instruction of T_1, we can swap these instructions to generate an equivalent schedule, schedule 5. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

We continue to swap non-conflicting instructions:

- Swap the read(*B*) instruction of T_1 with the read(*A*) instruction of T_2.
- Swap the write(*B*) instruction of T_1 with the write(*A*) instruction of T_2.
- Swap the write(*B*) instruction of T_1 with the read(*A*) instruction of T_2.

The final result of these swaps, schedule 6, is a serial schedule. Note that schedule 6 is exactly the same as schedule 1, but it shows only the read and write instructions. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.

<u>Definition:</u>

- **If a schedule *S* can be transformed into a schedule *S* by a series of swaps of non-conflicting instructions, we say that *S* and *S* are conflict equivalent. The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule *S* is conflict serializable if it is conflict equivalent to a serial schedule.**

- Not all serial schedules are conflict equivalent to each other. For example, schedules 1 and 2 are not conflict equivalent.
- Schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.
- Finally, consider schedule 7; it consists of only the significant operations (that is, the read and write) of transactions T_3 and T_4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule $<T_3,T_4>$ or the serial schedule $<T_4,T_3>$.

PRECEDENCE GRAPH:

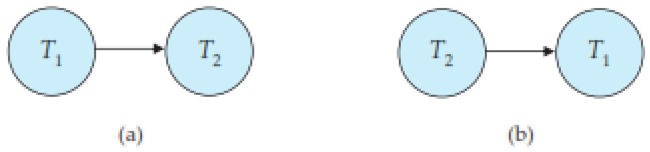


Figure. Precedence graph for (a) schedule 1 and (b) schedule 2.

- Precedence graph is a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule *S*. We construct a directed graph, called a **precedence graph**, from *S*. This graph consists of a pair *G* = (*V*, *E*), where *V* is a set of vertices and *E* is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

  - ✓ T_i executes write(*Q*) before T_j executes read(*Q*).
  - ✓ T_i executes read(*Q*) before T_j executes write(*Q*).
  - ✓ T_i executes write(*Q*) before T_j executes write(*Q*).

- If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule *S* equivalent to *S*, T_i must appear before T_j .

- For example, the precedence graph for schedule 1 in Figure(a) contains the single edge $T_1 \rightarrow T_2$, since all the instructions of T_1 are executed before the first instruction of T_2 is executed. Similarly, Figure(b) shows the precedence graph for schedule 2 with the single edge $T_2 \rightarrow T_1$, since all the instructions of T_2 are executed before the first instruction of T_1 is executed.

- The precedence graph for schedule 4 appears in Figure. It contains the edge $T_1 \rightarrow T_2$, because T_1 executes read(*A*) before T_2 executes write(*A*). It also contains the edge $T_2 \rightarrow T_1$, because T_2 executes read(*B*) before T_1 executes write(*B*).

- **If the precedence graph for *S* has a cycle, then schedule *S* is not conflict serializable. If the graph contains no cycles, then the schedule *S* is conflict serializable.**

- A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**. There are several possible linear orders that can be obtained through a topological sort.
- For example, the graph of Figure(a) has the two acceptable linear orderings shown in Figures(b) and (c).

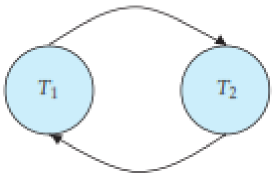


Figure. Precedence graph for schedule 4.

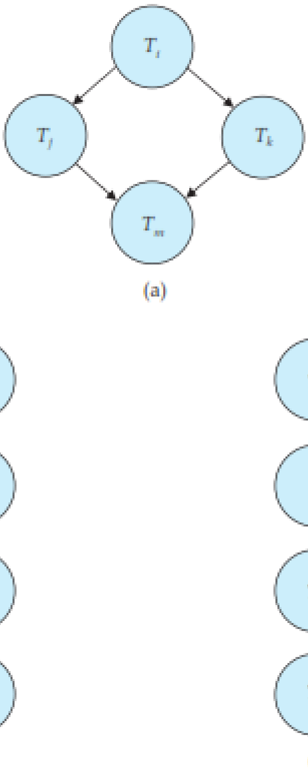


Figure . Illustration of topological sorting.

- Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms.
- Cycle-detection algorithms, such as those based on depth-first search, require on the order of n^2 operations, where n is the number of vertices in the graph (that is, the number of transactions).

- Returning to our previous examples, note that the precedence graphs for schedules 1 and 2, indeed do not contain cycles. The precedence graph for schedule 4, on the other hand, contains a cycle, indicating that this schedule is not conflict serializable.

- It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent. For example, consider transaction T_5, which transfers \$10 from account B to account A.

- Let schedule 8 be as defined in Figure. We claim that schedule 8 is not conflict equivalent to the serial schedule $<T_1,T_5>$, since, in schedule 8, the write(B) instruction of T_5 conflicts with the read(B) instruction of T_1. This creates an edge $T_5 \rightarrow T_1$ in the precedence graph.
- Similarly, we see that the write(A) instruction of T_1 conflicts with the read instruction of T_5 creating an edge $T_1 \rightarrow T_5$. This shows that the precedence graph has a cycle and that schedule 8 is not serializable. However, the final values of accounts A and after the execution of either schedule 8 or the serial schedule $<T_1,T_5>$ are the same —\$960 and \$2040, respectively.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figure. Schedule 8.

We can see from this example that there are less-stringent definitions of schedule equivalence than conflict equivalence. For the system to determine that schedule 8 produces the same outcome as the serial schedule $<T_1,T_5>$, it must analyze the computation performed by T_1 and T_5, rather than just the read and write operations.

<u>Recoverable Schedules:</u>

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

Figure . Schedule 9, a nonrecoverable schedule.

- Consider the partial schedule 9 in Figure, in which T_7 is a transaction that performs only one instruction: read(A). We call this a **partial schedule** because we have not included a **commit** or **abort** operation for T_6. Notice that T_7 commits immediately after executing the read(A) instruction. Thus, T_7 commits while T_6 is still in the active state. Now suppose that T_6 fails before it commits. T_7 has read the value of data item A written by T_6. Therefore, we say that T_7 is **dependent** on T_6. Because of this, we must abort T_7 to ensure atomicity. However, T_7 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T_6.
- Schedule 9 is an example of a *nonrecoverable* schedule.

- **A recoverable schedule is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the commit operation of T_j . For the example of schedule 9 to be recoverable, T_7 would have to delay committing until after T_6 commits.**

Cascadeless Schedules:

- Even if a schedule is recoverable, to recover correctly from the failure of a transaction T_i , we may have to roll back several transactions. Such situations occur if transactions have read data written by T_i .
- **Consider the partial schedule of Figure. Transaction T_8 writes a value of A that is read by transac-tion T_9. Transaction T_9 writes a value of A that is read by transaction T_{10}. Suppose that, at this point, T_8 fails. T_8 must be rolled back. Since T_9 is dependent on T_8, T_9 must be rolled back. Since T_{10} is dependent on T_9, T_{10} must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.**

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Figure. Schedule 10.

- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally, a **cascadeless schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the read operation of T_j . It is easy to verify that every cascadeless schedule is also recoverable.

2. VIEW SERIALIZABILITY:

- There is another form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

- Consider two schedules S and S , where the same set of transactions partici-pates in both schedules. The schedules S and S are said to be **view equivalent** if three conditions are met:

  (1) For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S , also read the initial value of Q.
  (2) For each data item Q, if transaction T_i executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction T_j , then the read(Q) operation of transaction T_i must, in schedule S , also read the value of Q that was produced by the same write(Q) operation of transaction T_j .
  (3) For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S .

- Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

- The concept of view equivalence leads to the concept of view serializability. We say that a schedule S is **view serializable** if it is view equivalent to a serial schedule.

- As an illustration, suppose that we augment schedule 4 with transaction T_{29}, and obtain the following view serializable (schedule 5):

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

  - Indeed, schedule 5 is view equivalent to the serial schedule <T_{27}, T_{28}, T_{29}>, since the one read(Q) instruction reads the initial value of Q in both schedules and T_{29} performs the final write of Q in both schedules.

- Every conflict-serializable schedule is also view serializable, but there are view-serializable schedules that are not conflict serializable. Indeed, schedule 5 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

- Observe that, in schedule 5, transactions T_{28} and T_{29} perform write(Q) operations without having performed a read(Q) operation. Writes of this sort are called **blind writes**. Blind writes appear in any view-serializable schedule that is not conflict serializable.

3.5. NEED FOR CONCURRENCY CONTROL:

Transactions running concurrently may interfere with each other, causing various problems (lost updates etc.) Concurrency control is the process of managing simultaneous operations on the database without having them interfere with each other.

3.6. LOCK-BASED PROTOCOLS

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item.

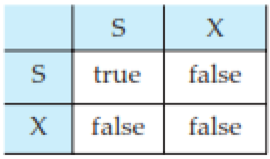
Locks:

- There are various modes in which a data item may be locked.

Two modes are:

1) Shared. If a transaction T_i has obtained a **shared-mode lock** (denoted by S) on item Q, then T_i can read, but cannot write, Q.

2) Exclusive. If a transaction T_i has obtained an **exclusive-mode lock** (denoted by X) on item Q, then T_i can both read and write Q.



	S	X
S	true	false
X	false	false

Figure. Lock-compatibility matrix comp.

- We require that every transaction **request** a lock in an appropriate mode on data item *Q*, depending on the types of operations that it will perform on *Q*. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

To state this more generally, given a set of lock modes, we can define a **compatibility function** on them as follows: Let *A* and *B* represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode *A* on item *Q* on which transaction T_j (T_i =T_j) currently holds a lock of mode *B*. If transaction T_i can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is **compatible** with mode *B*. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure. An element comp(*A*, *B*) of the matrix has the value *true* if and only if mode is compatible with mode *B*.

- Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

- A transaction requests a shared lock on data item *Q* by executing the lock-S(*Q*) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction. A transaction can unlock a data item *Q* by the unlock(*Q*) instruction.

- To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency-control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, T_i is made to **wait** until all incompatible locks held by other transactions have been released.

- Transaction T_i may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

- Consider again the banking example. Let *A* and *B* be two accounts that are accessed by transactions T_1 and T_2.

```
T1: lock-X(B);
read(B);
B := B − 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(A).
```

Figure. Transaction T_1.

- Transaction T_1 transfers $50 from account *B* to account *A*.
- Transaction T_2 displays the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*..
- Suppose that the values of accounts *A* and *B* are $100 and $200, respectively. If these two transactions are executed serially, either in the order T_1, T_2 or the order T_2, T_1, then transaction T_2 will display the value $300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 15.4, is possible. In this case, transaction T_2 displays $250, which is incorrect. The reason for this mistake is that the transaction T_1 unlocked data item *B* too early, as a result of which T_2 saw an inconsistent state.
- The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks. The transac-tion making a lock request cannot execute its next action until the concurrency-control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the trans-action. Exactly when within this interval the lock is granted is not important; we can safely assume that the lock is granted just before the following action of the transaction. We shall therefore drop the column depicting the actions of the concurrency-control manager from all schedules depicted in the rest of the chapter. We let you infer when locks are granted.

```
T2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display(A + B).
```

Figure. Transaction T_2.

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Figure. Schedule 1.

- Suppose now that unlocking is delayed to the end of the transaction. Trans-action T_3 corresponds to T_1 with unlocking delayed. Transaction T_4 corresponds to T_2 with unlocking delayed.

```
T3: lock-X(B);
read(B);
B := B − 50;
write(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(B);
unlock(A).
```

Figure. Transaction T_3 (transaction T_1 with unlocking delayed).

```
T4: lock-S(A);
read(A);
lock-S(B);
read(B);
display(A + B);
unlock(A);
unlock(B).
```

Figure. Transaction T_4 (transaction T_2 with unlocking delayed).

- You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of $250 being displayed, is no longer possible with T_3 nd T_4. Other schedules are possible. T_4 will not print out an inconsistent result in any of them;
- Unfortunately, locking can lead to an undesirable situation.

- **Consider the partial schedule of Figure for T_3 and T_4. Since T_3 is holding an exclusive-mode lock on *B* and T_4 is requesting a shared-mode lock on *B, T_4* is waiting for T_3 to unlock *B*. Similarly, since T_4 is holding a shared-mode lock on *A* and T_3 is requesting an exclusive-mode lock on *A, T_3* is waiting for T_4 to unlock *A*. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock.**

- When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. There are ways to avoid deadlock in some situations.
- However, in general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure. Schedule 2

- We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules.

GRANTING OF LOCKS:

- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.
- However, care must be taken to avoid the following scenario. Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item.
- Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock.
- At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the

data item, and each transaction releases the lock a short while after it is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be **starved**.

- We can avoid starvation of transactions by granting locks in the following manner: When a transaction T_i requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:
- There is no other transaction holding a lock on Q in a mode that conflicts with M.
- There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i .
- Thus, a lock request will never get blocked by a lock request that is made later.

3.7. THE TWO-PHASE LOCKING PROTOCOL:

- One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1) Growing phase. A transaction may obtain locks, but may not release any lock.

2) Shrinking phase. A transaction may release locks, but may not obtain any new locks.

- Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
- The two-phase locking protocol ensures conflict serializability.
- Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction.
- Now, transactions can be ordered according to their lock points— this ordering is, in fact, a serializability ordering for the transactions. We leave the proof as an exercise for you to do.
- Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions T_3 and T_4 are two phase, but, in schedule 2 , they are deadlocked.
- Cascading rollback may occur under two-phase locking.
- Consider the partial schedule of Figure 15.8. Each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure .Partial schedule under two-phase locking.

STRICT TWO-PHASE LOCKING PROTOCOL:

- Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

RIGOROUS TWO-PHASE LOCKING PROTOCOL:

- Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits.

- Consider the following two transactions, for which we have shown only some of the significant read and write operations:

T_8: read(a_1);
read(a_2);
. . .
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display(a_1 + a_2).

- If we employ the two-phase locking protocol, then T_8 must lock a_1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that T_8 needs an exclusive lock on a_1 only at the end of its execution, when it writes a_1. Thus, if T_8 could initially lock a_1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since T_8 and T_9 could access a_1 and a_2 simultaneously.

- This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed.

- We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**.

- Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

T_8	T_9
lock-s(a_1)	
	lock-s(a_1)
lock-s(a_2)	
	lock-s(a_2)
lock-s(a_3)	
lock-s(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-s(a_n)	
upgrade(a_1)	

Figure. Incomplete schedule with a lock conversion.

- Returning to our example, transactions T_8 and T_9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure, where only some of the locking instructions are shown.
- Note that a transaction attempting to upgrade a lock on an item Q may be forced to wait. This enforced wait occurs if Q is currently locked by *another* transaction in shared mode.
- Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

- For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. We shall see examples when we consider other locking protocols later in this chapter.

- Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

- A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a read(Q) operation, the system issues a lock-S(Q) instruction followed by the read(Q) instruction.

- When T_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an up-grade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the write(Q) instruction.

- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

<u>IMPLEMENTATION OF LOCKING:</u>

- A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply.
- The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks).
- Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

- The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**.

- Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.
- Figure shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

- Although the figure does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction.
- The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

- It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.

- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction , it releases all locks held by the aborted transaction.

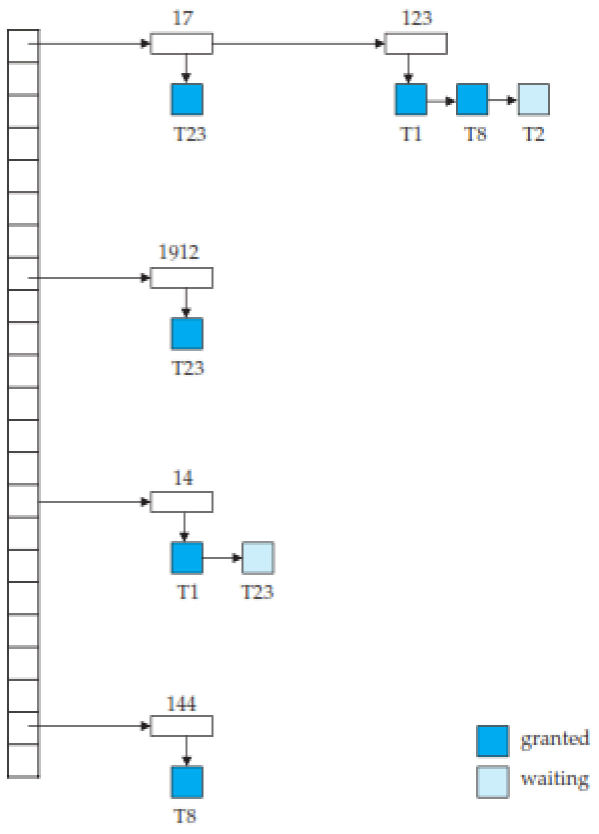


Figure. Lock table.

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted.

GRAPH-BASED PROTOCOLS: (or) Tree Protocol

- The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

- To acquire such prior knowledge, we impose a partial ordering → on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j . This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

- The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**. In this section, for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We shall present a simple protocol, called the *tree protocol*, which is restricted to employ only *exclusive* locks. References to other, more complex, graph-based locking protocols are in the bibliographical notes.

- In the **tree protocol**, the only lock instruction allowed is lock-X. Each trans-action T_i can lock a data item at most once, and must observe the following rules:

  - The first lock by T_i may be on any data item.
  - Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i .
  - Data items may be unlocked at any time.
  - A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i .

- All schedules that are legal under the tree protocol are conflict serializable.

- To illustrate this protocol, consider the database graph of Figure. The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).

T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).

T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).

T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

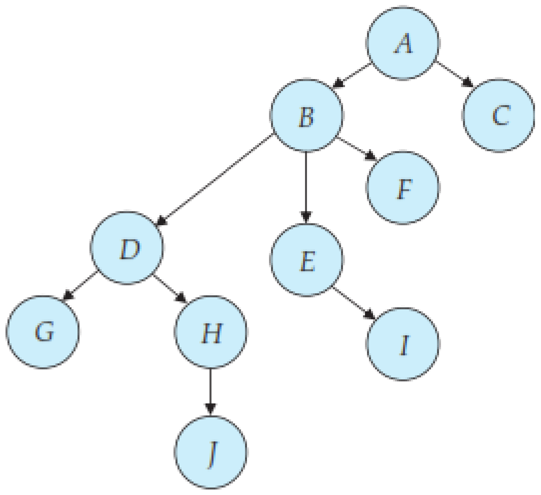


Figure. Tree-structured database graph.

- One possible schedule in which these four transactions participated appears in Figure. Note that, during its execution, transaction T_{10} holds locks on two *disjoint* subtrees.
- Observe that the schedule of Figure is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.
- The tree protocol does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction.
- Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write, we record which transaction performed the last write to the data item.
- Whenever a transaction T_i performs a read of an uncommitted data item, we record a **commit dependency** of T_i on the transaction that performed the last write to the data item. Transaction T_i is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, T_i must also be aborted.

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(*B*)			
	lock-X(*D*)		
	lock-X(*H*)		
	unlock(*D*)		
lock-X(*E*)			
lock-X(*D*)			
unlock(*B*)			
unlock(*E*)			
		lock-X(*B*)	
		lock-X(*E*)	
	unlock(*H*)		
lock-X(*G*)			
unlock(*D*)			
			lock-X(*D*)
			lock-X(*H*)
			unlock(*D*)
			unlock(*H*)
		unlock(*E*)	
		unlock(*B*)	
unlock(*G*)			

Figure .Serializable schedule under the tree protocol.

- The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

- However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items *A* and *J* in the database graph of Figure must lock not only *A* and *J*, but also data items *B*, *D*, and *H*.

- This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

- For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that are not possible under the tree protocol, and vice versa.

3.8. DEADLOCK:

- A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and . . . , and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

- The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

- There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state.

- Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme. Both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

- Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

DEADLOCK PREVENTION:

There are five approaches to deadlock prevention.

1) The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked.
   Two main disadvantages to this protocol:
   a) It is often hard to predict, before the transaction begins, what data items need to be locked;
   b) Data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

2) Another approach for preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed is to ensure that locks are requested in the right order.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction T_j requests a lock that transaction T_i holds, the lock granted to T_i may be **preempted** by rolling back of T_i , and granting of the lock to T_j . To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted.

Two different deadlock-prevention schemes using timestamps have been proposed:

3) Wait –die scheme:

- Consider a transaction T_i requests to lock a resource (data item), which is already held with a conflicting lock by another transaction T_j.
- (TS- Time Stamp)
- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock, is older than T_j − then T_i is allowed to wait until the data-item is available. Else T_i dies.
- T_i is restarted later with a random delay but with the same timestamp.
- The **wait–die** scheme is a non-preemptive technique.

4) Wound–wait scheme:

- It is a preemptive technique.
- Consider a transaction T_i requests to lock a resource (data item), which is already held with a conflicting lock by another transaction T_j.
- (TS- Time Stamp)

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- Else T_i is forced to wait until the resource is available.
- The major problem with both of these schemes is that unnecessary rollbacks may occur.

5) Time out based approach:

- In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.
- If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.

- The timeout scheme is particularly easy to implement, and works well if transactions are short and if long waits are likely to be due to deadlocks. However, in general it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources. Starvation is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

DEADLOCK DETECTION:

- If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

  - ➢ Maintain information about the current allocation of data items to transac-tions, as well as any outstanding data item requests.
  - ➢ Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
  - ➢ Recover from the deadlock when the detection algorithm determines that a deadlock exists.

- Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges.
- The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair $T_i \rightarrow T_j$. If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j , implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.

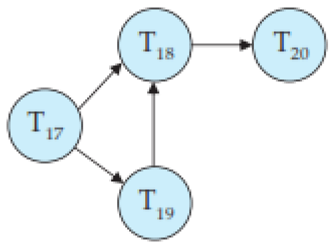


Figure. Wait-for graph with no cycle.

- When transaction T_i requests a data item currently being held by transaction T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

- To illustrate these concepts, consider the wait-for graph in Figure, which depicts the following situation:

    - ➢ Transaction T_{17} is waiting for transactions T_{18} and T_{19}.
    - ➢ Transaction T_{19} is waiting for transaction T_{18}.
    - ➢ Transaction T_{18} is waiting for transaction T_{20}.

- Since the graph has no cycle, the system is not in a deadlock state.

- Suppose now that transaction T_{20} is requesting an item held by T_{19}. The edge $T_{20} \rightarrow T_{19}$ is added to the wait-for graph, resulting in the new system state in Figure 15.14. This time, the graph contains the cycle:

${}^{T}18 \rightarrow {}^{T}20 \rightarrow {}^{T}19 \rightarrow {}^{T}18$
implying that transactions T_{18}, T_{19}, and T_{20} are all deadlocked.

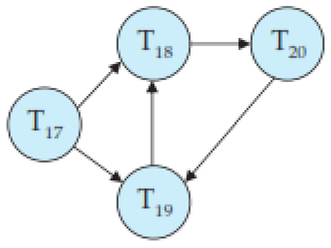


Figure. Wait-for graph with a cycle.

- If deadlocks occur frequently, then the detection algorithm should be in-voked more frequently. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

RECOVERY FROM DEADLOCK:

- When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock.
- Three actions need to be taken:

1) **Selection of a victim**. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including:

a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
b. How many data items the transaction has used.
c. How many more data items the transaction needs for it to complete.
d. How many transactions will be involved in the rollback.

2) **Rollback**. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.

- The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded.
- The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the bibliographical notes for relevant references.

3) **Starvation**. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

UNIT IV IMPLEMENTATION TECHNIQUES

RAID – File Organization – Organization of Records in Files – Indexing and Hashing – Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for SELECT and JOIN operations – Query optimization using Heuristics and Cost Estimation.

4.1. RAID

- The data-storage requirements of some applications (in particular Web, database, and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.
- Having a large number of disks in a system presents opportunities for im-proving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.
- A variety of disk-organization techniques, collectively called **redundant arrays of independent disks** (**RAID)**, have been proposed to achieve improved performance and reliability.
- In the past, system designers viewed storage systems composed of several small, cheap disks as a cost-effective alternative to using large, expensive disks; the cost per megabyte of the smaller disks was less than that of larger disks. In fact, the I in RAID, which now stands for *independent*, originally stood for *inexpensive*. Today, however, all disks are physically small, and larger-capacity disks actually have a lower cost per megabyte.
- RAID systems are used for their higher reliability and higher performance rate, rather than for economic reasons.
- Another key justification for RAID use is easier management and operations.

Improvement of Reliability via Redundancy:

- If we store only one copy of the data, then each disk failure will result in loss of a significant amount of datak. Such a high frequency of data loss is unacceptable.
- The solution to the problem of reliability is to introduce **redundancy**; that is, we store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost, so the effective mean time to failure is increased, provided that we count only failures that lead to loss of data or to non availability of data.

- The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or, sometimes, *shadow-ing*). A logical disk then consists of two physical disks, and every write is carriedout on both disks. If one of the disks fails, the data can be read from the other.

- Data will be lost only if the second disk fails before the first failed disk is repaired.

- The mean time to failure (where failure is the loss of data) of a mirrored disk depends on the mean time to failure of the individual disks, as well as on the **mean time to repair**, which is the time it takes (on an average) to replace a failed disk and to restore the data on it.

- Power failures, and natural disasters such as earthquakes, fires, and floods, may result in damage to both disks at the same time. As disks age, the probability of failure increases, increasing the chance that a second disk will fail while the first is being repaired. In spite of all these considerations, however, mirrored-disk systems offer much higher reliability than do single-disk systems.

- Mirrored-disk systems with mean time to data loss of about 500,000 to 1,000,000 hours, or 55 to 110 years, are available today.

<u>Improvement in Performance via Parallelism:</u>

- Now let us consider the benefit of parallel access to multiple disks. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

- With multiple disks, we can improve the transfer rate as well (or instead) by **striping data** across multiple disks.

Bit-level striping :

- Data striping consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.

- For example, if we have an array of eight disks, we write bit *i* of each byte to disk *i* . The array of eight disks can be treated as a single disk with sectors that are eight times the normal size, and, more important, that has eight times the transfer rate. In such an organization, every disk participates in every access (read or write), so the number of accesses that can be processed per second is about the same as on a single disk, but each access can read eight times as many data in the same time as on a single disk. Bit-level striping can be generalized to a number of disks that either is a multiple of 8 or a factor of 8. For example, if we use an array of four disks, bits *i* and 4 + *i* of each byte go to disk *i* .

<u>Block-level striping :</u>

- **Block-level striping** stripes blocks across multiple disks. It treats the array of disks as a single large disk, and it gives blocks logical numbers; we assume the block numbers start from 0. With an array of n disks, block-level striping assigns logical block i of the disk array to disk (i mod n) + 1; it uses the i /n th physical block of the disk to store logical block i .

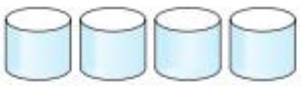
- For example, with 8 disks, logical block 0 is stored in physical block 0 of disk 1, while logical block 11 is stored in physical block 1 of disk 4. When reading a large file, block-level striping fetches n blocks at a time in parallel from the n disks, giving a high data-transfer rate for large reads. When a single block is read, the data-transfer rate is the same as on one disk, but the remaining $n - 1$ disks are free to perform other actions.

- Block-level striping is the most commonly used form of data striping. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible.

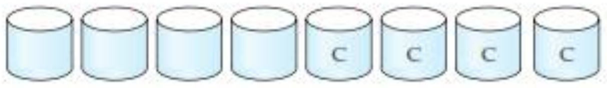
There are two main goals of parallelism in a disk system:

1) Load-balance multiple small accesses (block accesses), so that the throughput of such accesses increases.
2) Parallelize large accesses so that the response time of large accesses is reduced.

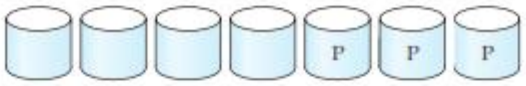
<u>RAID Levels:</u>



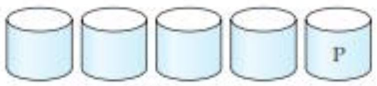
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



(c) RAID 2: memory-style error-correcting codes



(d) RAID 3: bit-interleaved parity

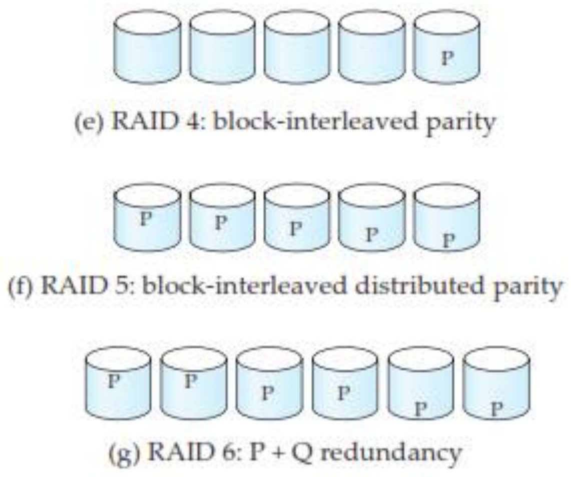

P
(e) RAID 4: block-interleaved parity
P P P P P
(f) RAID 5: block-interleaved distributed parity
P P P P P P
(g) RAID 6: P + Q redundancy

Figure. RAID levels.

- Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability. Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with “parity” bits. These schemes have different cost – performance trade-offs. The schemes are classified into **RAID levels**.

- In the figure, P indicates error-correcting bits, and C indicates a second copy of the data.

- For all levels, the figure depicts four disks’ worth of data, and the extra disks depicted are used to store redundant information for failure recovery.

<u>**RAID level 0:**</u>

- **RAID level 0 r**efers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits). Figure(a) shows an array of size 4.

RAID level 1:

- **RAID level 1 r**efers to disk mirroring with block striping. Figure(b) shows a mirrored organization that holds four disks’ worth of data.
- Note that some vendors use the term **RAID level 1+0** or **RAID level 10** to refer to mirroring with striping, and use the term RAID level 1 to refer to mirroring without striping. Mirroring without striping can also be used with arrays of disks, to give the appearance of a single large, reliable disk: if each disk has M blocks, logical blocks 0 to $M - 1$ are stored on disk 0, M to $2M - 1$ on disk 1(the second disk), and so on, and each disk is mirrored.

RAID level 2:

- **RAID level 2** known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for error detection and correction. Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte that are set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte gets damaged (either a

1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity. Thus, all 1-bit errors will be detected by the memory system. Error-correcting schemes store 2 or more extra bits, and can reconstruct the data if a single bit gets damaged.

- The idea of error-correcting codes can be used directly in disk arrays by striping bytes across disks. For example, the first bit of each byte could be stored in disk 0, the second bit in disk 1, and so on until the eighth bit is stored in disk 7, and the error-correction bits are stored in further disks.

- Figure (c) shows the level 2 scheme. The disks labeled *P* store the error-correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks, and can be used to reconstruct the damaged data. Figure 10.3c shows an array of size 4; note RAID level 2 requires only three disks' overhead for four disks of data, unlike RAID level 1, which required four disks' overhead.

RAID level 3:

- **RAID level 3** bit-interleaved parity organization, improves on level 2 by exploiting the fact that disk controllers, unlike memory systems, can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows: If one of the sectors gets damaged, the system knows exactly which sector it is, and, for each bit in the sector, the system can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.
- RAID level 3 is as good as level 2, but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice. Figure (d) shows the level 3 scheme.

- RAID level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas level 1 needs one mirror disk for every disk, and thus level 3 reduces the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with *N*-way striping of data, the transfer rate for reading or writing a single block is *N* times faster than a RAID level 1 organization using *N*-way striping. On the other hand, RAID level 3 supports a lower number of I/O operations per second, since every disk has to participate in every I/O request.

RAID level 4:

- **RAID level 4** block-interleaved parity organization, uses block-level striping, like RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from *N* other disks. This scheme is shown pictorially in Figure (e). If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

- A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

- Small independent writes, on the other hand, cannot be performed in parallel. A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated. Moreover, both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed. Thus, a single write requires four disk accesses: two to read the two old blocks, and two to write the two blocks.

RAID level 5:

- **RAID level 5** block-interleaved distributed parity, improves on level 4 by partitioning data and parity among all *N* + 1 disks, instead of storing data in *N* disks and parity in one disk. In level 5, all disks can participate in satisfying read requests, unlike RAID level 4, where the parity disk cannot participate, so level 5 increases the total number of requests that can be met in a given amount of time. For each set of *N* logical blocks, one of the disks stores the parity, and the other *N* disks store the blocks.
- Figure shows the setup. The P's are distributed across all the disks. For example, with an array of 5 disks, the parity block, labeled Pk, for logical blocks 4*k*, 4*k* + 1, 4*k* + 2, 4*k* + 3 is stored in disk *k* mod 5; the corresponding blocks of the other four disks store the 4 data blocks 4*k* to 4*k* + 3. The following table indicates how the first 20 blocks, numbered 0 to 19, and their parity blocks are laid out. The pattern shown gets repeated on further blocks.

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

- Note that a parity block cannot store parity for blocks in the same disk, since then a disk failure would result in loss of data as well as of parity, and hence would not be recoverable. Level 5 subsumes level 4, since it offers better read – write performance at the same cost, so level 4 is not used in practice.

RAID level 6:

- **RAID level 6 is** the P + Q redundancy scheme, is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures. Instead of using parity, level 6 uses error-correcting codes such as the Reed – Solomon codes (see the bibliographical notes). In the scheme in Figure (g), 2 bits of redundant data are stored for every 4 bits of data — unlike 1 parity bit in level 5 — and the system can tolerate two disk failures.

<u>Choice of RAID Level:</u>

The factors to be taken into account in choosing a RAID level are:

- Monetary cost of extra disk-storage requirements.
- Performance requirements in terms of number of I/O operations.
- Performance when a disk has failed.
- Performance during rebuild (that is, while the data in a failed disk are being rebuilt on a new disk).

- The time to rebuild the data of a failed disk can be significant, and it varies with the RAID level that is used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk. The **rebuild performance** of a RAID system may be an important factor if continuous availability of data is required, as it is in high-performance database systems. Furthermore, since rebuild time can form a significant part of the repair time, rebuild performance also influences the mean time to data loss.

- RAID level 0 is used in high-performance applications where data safety is not critical. Since RAID levels 2 and 4 are subsumed by RAID levels 3 and 5, the choice of RAID levels is restricted to the remaining levels. Bit striping (level 3) is inferior to block striping (level 5), since block striping gives as good data-transfer rates for large transfers, while using fewer disks for small transfers.

- For small transfers, the disk access time dominates anyway, so the benefit of parallel reads diminishes. In fact, level 3 may perform worse than level 5 for a small transfer, since the transfer completes only when corresponding sectors on all disks have been fetched; the average latency for the disk array thus becomes very close to the worst-case latency for a single disk, negating the benefits of higher transfer rates. Level 6 is not supported currently by many RAID implementations, but it offers better reliability than level 5 and can be used in applications where data safety is very important.

- The choice between RAID level 1 and level 5 is harder to make. RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance. RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes. For applications where data are read frequently, and written rarely, level 5 is the preferred choice.

- Disk-storage capacities have been growing at a rate of over 50 percent per year for many years, and the cost per byte has been falling at the same rate. As a result, for many existing database applications with moderate storage requirements, the monetary cost of the extra disk storage needed for mirroring has become relatively small (the extra monetary cost, however, remains a significant issue for storage-intensive applications such as video data storage). Access speeds have improved at a much slower rate (around a factor of 3 over 10 years), while the number of I/O operations required per second has increased tremendously, particularly for Web application servers.

- RAID level 5, which increases the number of I/O operations needed to write a single logical block, pays a significant time penalty in terms of write performance. RAID level 1 is therefore the RAID level of choice for many applications with moderate storage requirements and high I/O requirements.

- RAID system designers have to make several other decisions as well. For example, how many disks should there be in an array? How many bits should be protected by each parity bit? If there are more disks in an array, data-transfer rates are higher, but the system will be more expensive. If there are more bits protected by a parity bit, the space overhead due to parity bits is lower, but there is an increased chance that a second disk will fail before the first failed disk is repaired, and that will result in data loss.

<u>Hardware Issues:</u>

- Another issue in the choice of RAID implementations is at the level of hardware. RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called **software RAID**. However, there are significant benefits to be had by building special-purpose hardware to support RAID, which we outline below; systems with special hardware support are called **hardware RAID** systems.

- Hardware RAID implementations can use nonvolatile RAM to record writes before they are performed. In case of power failure, when the system comes back up, it retrieves information about any incomplete writes from nonvolatile RAM and then completes the writes. Without such hardware support, extra work needs to be done to detect blocks that may have been partially written before power failure.

- Even if all writes are completed properly, there is a small chance of a sector in a disk becoming unreadable at some point, even though it was successfully written earlier. Reasons for loss of data on individual sectors could range from manufacturing defects, to data corruption on a track when an adjacent track is written repeatedly. Such loss of data that were successfully written earlier is sometimes referred to as a *latent failure*, or as *bit rot*. When such a failure happens, if it is detected early the data can be recovered from the remaining disks in the RAID organization. However, if such a failure remains undetected, a single disk failure could lead to data loss if a sector in one of the other disks has a latent failure.

- To minimize the chance of such data loss, good RAID controllers perform **scrubbing**; that is, during periods when disks are idle, every sector of every disk is read, and if any sector is found to be unreadable, the data are recovered from the remaining disks in the RAID organization, and the sector is written back. (If the physical sector is damaged, the disk controller would remap the logical sector address to a different physical sector on disk.)

- Some hardware RAID implementations permit **hot swapping**; that is, faulty disks can be removed and replaced by new ones without turning power off. Hot swapping reduces the mean time to repair, since replacement of a disk does not have to wait until a time when the system can be shut down. In fact many critical systems today run on a 24 × 7 schedule; that is, they run 24 hours a day, 7 days a week, providing no time for shutting down and replacing a failed disk. Further, many RAID implementations assign a spare disk for each array (or for a set of disk arrays). If a disk fails, the spare disk is immediately used as a replacement. As a result, the mean time to repair is reduced greatly, minimizing the chance of any data loss. The failed disk can be replaced at leisure.

- The power supply, or the disk controller, or even the system interconnection in a RAID system could become a single point of failure that could stop functioning of the RAID system. To avoid this possibility, good RAID implementations have multiple redundant power supplies (with battery backups so they continue to function even if power fails). Such RAID systems have multiple disk interfaces, and multiple interconnections to connect the RAID system to the computer system (or to a network of computer systems). Thus, failure of any single component will not stop the functioning of the RAID system.

Other RAID Applications:

- The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, the RAID structures are able to recover data even if one of the tapes in an array of tapes is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units.

4.2. FILE ORGANIZATION:

- A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks.
- A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files.
- Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer. Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created. Larger block sizes can be useful in some database applications.
- A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used. We shall assume that *no record is larger than a block*. This assumption is realistic for most data-processing applications, such as our university example. There are certainly several kinds of large data items, such as images, that can be significantly larger than a block. We briefly discuss how to handle such large data items later, in Section 10.5.2, by storing large data items separately, and storing a pointer to the data item in the record.
- In addition, we shall require that *each record is entirely contained in a single block*; that is, no record is contained partly in one block, and partly in another. This restriction simplifies and speeds up access to data items.
- In a relational database, tuples of distinct relations are generally of different sizes. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records; however, files of fixed-length records are easier to implement than are files of variable-length records. Many of the techniques used for the former can be applied to the variable-length case. Thus, we begin by considering a file of fixed-length records, and consider storage of variable-length records later.

1) Fixed-Length Records:

- Consider a file of *instructor* records for university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record
ID varchar (5);
name varchar(20);
dept name varchar (20);
salary numeric (8,2);
end
```

- Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes. Suppose that instead of allocating a variable amount of bytes for the attributes *ID*, *name*, and *dept name*, we allocate the maximum number of bytes that each attribute can hold. Then, the *instructor* record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on.

Two problems with this simple approach:

1) Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

2) It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	CaliÞeri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure. File containing *instructor* records.

- To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

- When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead. Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	CaliÞeri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure . File with record 3 deleted and all records moved.

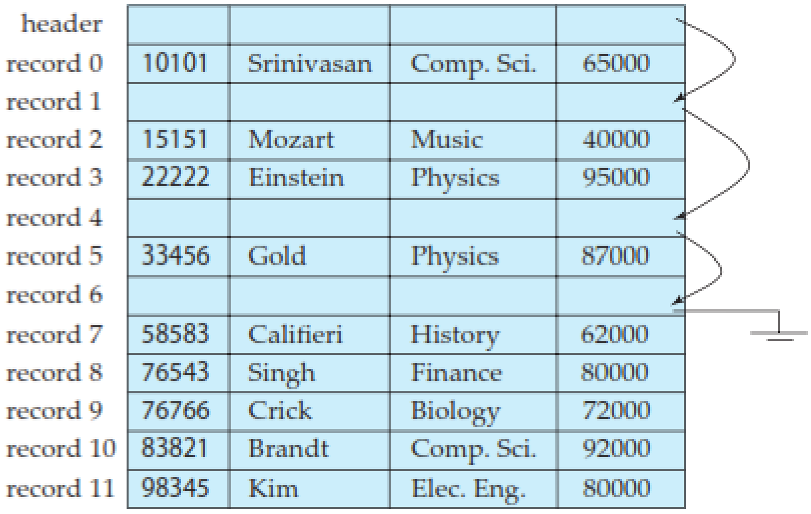
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	CaliÞeri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Figure. File with record 3 deleted and final record moved.

- It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space. A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

- At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list**. Figure 10.7 shows the file of Figure 10.4, with the free list, after records 1, 4, and 6 have been deleted.

- On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure. File with free list after deletion of records 1, 4, and 6.

- Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record. If we allow records of variable length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.

2) <u>Variable-Length Records:</u>

Variable-length records arise in database systems in several ways:

- ➢ Storage of multiple record types in a file.
- ➢ Record types that allow variable lengths for one or more fields.

- Record types that allow repeating fields, such as arrays or multisets.

- Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:

- How to represent a single record in such a way that individual attributes can be extracted easily.
- How to store variable-length records within a block, such that records in a block can be extracted easily.
- The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable-length attributes. Fixed-length attributes, such as numeric values, dates, or fixed-length character strings are allocated as many bytes as required to store their value. Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (*offset*, *length*), where *offset* denotes where the data for that attribute begins within the record, and *length* is the length in bytes of the variable-sized attribute. The values for these attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length.

- An example of such a record representation is shown in Figure 10.8. The figure shows an *instructor* record, whose first three attributes *ID*, *name*, and *dept name* are variable-length strings, and whose fourth attribute *salary* is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The *salary* attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

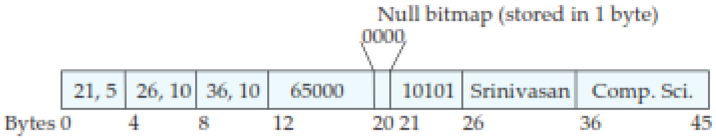


Figure. Representation of variable-length record.

- The figure also illustrates the use of a **null bitmap**, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the *salary* value stored in bytes 12 through 19 would be ignored. Since the record has four attributes, the null bitmap for this record fits in 1 byte, although more bytes may be required with more attributes. In some representations, the null bitmap is stored at the beginning of the record, and for attributes that are null, no data (value, or offset/length) are stored at all. Such a representation would save some storage space, at the cost of extra work to extract attributes of the record. This representation is particularly useful for certain applications where records have a large number of fields, most of which are null.

- We next address the problem of storing variable-length records in a block. The **slotted-page structure** is commonly used for organizing records within a block.

There is a header at the beginning of each block, containing the following information:

- ➢ The number of record entries in the header.
- ➢ The end of free space in the block.

- An array whose entries contain the location and size of each record.
- The actual records are allocated *contiguously* in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

- If a record is deleted, the space that it occupies is freed, and its entry is set to *deleted* (its size is set to −1, for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header array and the first record.

- The end-of-free-space pointer in the header is appropriately updated as well. Records can be grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records is not too high, since the size of a block is limited: typical values are around 4 to 8 kilobytes.

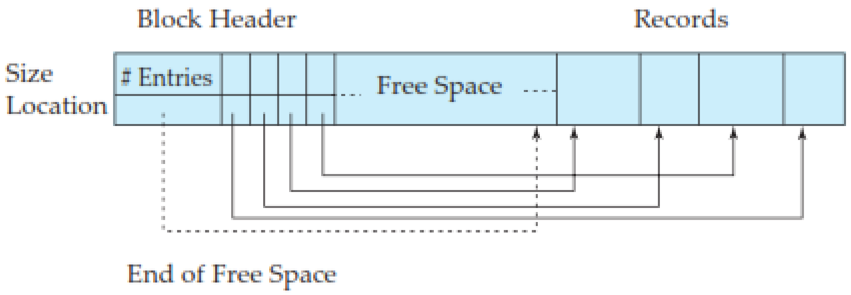


Figure. Slotted-page structure.

- The slotted-page structure requires that there be no pointers that point directly to records. Instead, pointers must point to the entry in the header that contains the actual location of the record. This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record.

- Databases often store data that can be much larger than a disk block. For instance, an image or an audio recording may be multiple megabytes in size, while a video object may be multiple gigabytes in size. Recall that SQL supports the types **blob** and **clob**, which store binary and character large objects.

- Most relational databases restrict the size of a record to be no larger than the size of a block, to simplify buffer management and free-space management. Large objects are often stored in a special file (or collection of files) instead of being stored with the other (short) attributes of records in which they occur. A (logical) pointer to the object is then stored in the record containing the large object. Large objects are often represented using B$^+$-tree file organizations. B$^+$-tree file organizations permit us to read an entire object, or specified byte ranges in the object, as well as to insert and delete parts of the object.

4.3. ORGANIZATION OF RECORDS IN FILES:

- A relation is a set of records.

Possible ways of organizing records in files are:

1) Heap file organization. Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.

2) Sequential file organization. Records are stored in sequential order, according to the value of a "search key" of each record.

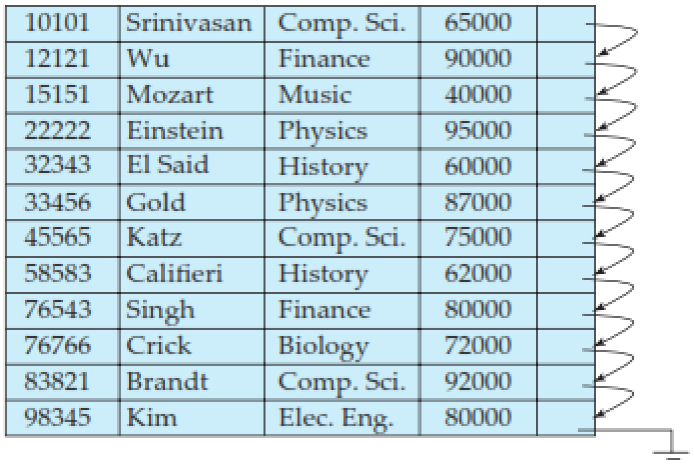
3) Hashing file organization. A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed. Generally, a separate file is used to store the records of each relation.

4) In a **multitable clustering file organization**, records of several different relations are stored in the same file; further, related records of the different relations are stored on the same block, so that one I/O operation fetches related records from all the relations. For example, records of the two relations can be considered to be related if they would match in a join of the two relations.

1) Sequential File Organization:

- A **sequential file** is designed for efficient processing of records in sorted order based on some search key. A **search key** is any attribute or set of attributes; it need not be the primary key, or even a super key.

- To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure. Sequential file for *instructor* records.

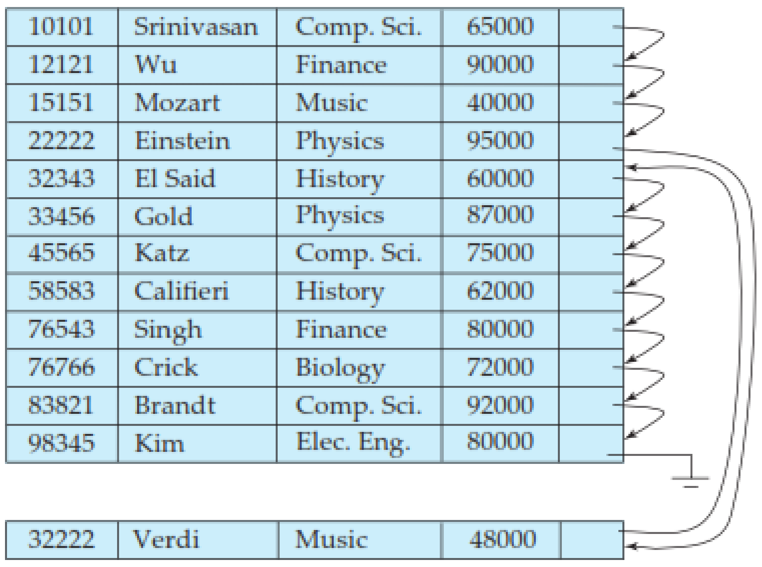
- Figure shows a sequential file of *instructor* records taken from our university example. In that example, the records are stored in search-key order, using *ID* as the search key.

- The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms.

- It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:

i) Locate the record in the file that comes before the record to be inserted in search-key order.

ii) If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
32222	Verdi	Music	48000

Figure. Sequential file after an insertion.

- Figure shows the file after the insertion of the record (32222, Verdi, Music, 48000). The structure in Figure allows fast insertion of new records, but forces sequential file-processing applications to process records in an order that does not match the physical order of the records.

- If relatively few records need to be stored in overflow blocks, this approach works well. Eventually, however, the correspondence between search-key order and physical order may be totally lost over a period of time, in which case sequential processing will become much less efficient. At this point, the file should be **reorganized** so that it is once again physically in sequential order.

- Such reorganizations are costly, and must be done during times when the system load is low. The frequency with which reorganizations are needed depends on the frequency of insertion of new records. In the extreme case in which insertions rarely occur, it is possible always to keep the file in physically sorted order. In such a case, the pointer field is not needed.

2) <u>Multitable Clustering File Organization:</u>

- Many relational database systems store each relation in a separate file, so that they can take full advantage of the file system that the operating system provides. Usually, tuples of a relation can be represented as fixed-length records. Thus, relations can be mapped to a simple file structure. This simple implementation of a relational database system is well suited to low-cost database implementations as in, for example, embedded systems or portable devices. In such systems, the size of the database is small, so little is gained from a sophisticated file structure. Furthermore, in such environments, it is essential that the overall size of the object code for the database system be small. A simple file structure reduces the amount of code needed to implement the system.

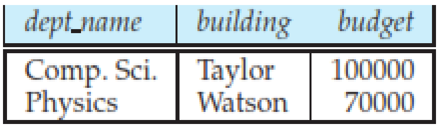
- This simple approach to relational database implementation becomes less satisfactory as the size of the database increases. We have seen that there are performance advantages to be gained from careful assignment of records to blocks, and from careful organization of the blocks themselves. Clearly, a more complicated file structure may be beneficial, even if we retain the strategy of storing each relation in a separate file.

- However, many large-scale database systems do not rely directly on the underlying operating system for file management. Instead, one large operating-system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself.

- Even if multiple relations are stored in a single file, by default most databases store records of only one relation in a given block. This simplifies data management. However, in some cases it can be useful to store records of more than one relation in a single block. To see the advantage of storing records of multiple relations in one block, consider the following SQL query for the university database:

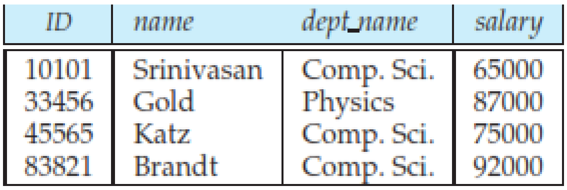
select *dept name, building, budget, ID, name, salary* **from** *department*
natural join *instructor*;

- This query computes a join of the *department* and *instructor* relations. Thus, for each tuple of *department*, the system must locate the *instructor* tuples with the same value for *dept name*. Ideally, these records will be located with the help of *indices*, which we shall discuss in Chapter 11. Regardless of how these records are located, however, they need to be transferred from disk into main memory. In the worst case, each record will reside on a different block, forcing us to do one block read for each record required by the query.



dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

Figure. The *department* relation.



ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

Figure. The *instructor* relation.

- As a concrete example, consider the *department* and *instructor* relations. In Figure, we show a file structure designed for efficient execution of queries involving the natural join of *department* and *instructor*. The *instructor* tuples for each *ID* are stored near the *department* tuple for the corresponding *dept name*. This structure mixes together tuples of two relations, but allows for efficient processing of the join.

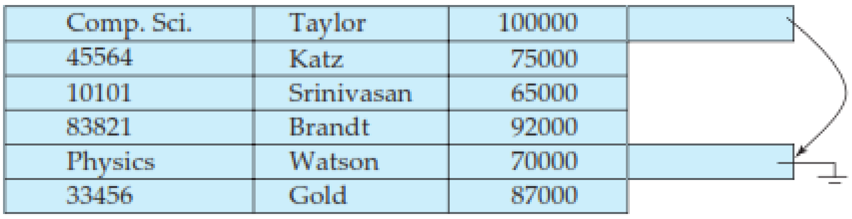
- When a tuple of the *department* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding *instructor* tuples are stored on the disk near the *department* tuple, the block containing the *department* tuple contains tuples of the *instructor* relation needed to process the query. If a department has so many instructors that the *instructor* records do not fit in one block, the remaining records appear on nearby blocks.

- A **multitable clustering file organization** is a file organization, such as that illustrated in Figure, that stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently.

- In the representation shown in Figure, the *dept name* attribute is omitted from *instructor* records since it can be inferred from the associated *department* record; the attribute may be retained in some implementations, to simplify access to the attributes. We assume that each record contains the identifier of the relation to which it belongs, although this is not shown in Figure.

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Figure. Multitable clustering file structure.



Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

Figure. Multitable clustering file structure with pointer chains.

select * **from** *department*;

- Our use of clustering of multiple tables into a single file has enhanced processing of a particular join (that of *department* and *instructor*), but it results in slowing processing of other types of queries. For example, requires more block accesses than it did in the scheme under which we stored each relation in a separate file, since each block now contains significantly fewer *department* records. To locate efficiently all tuples of the *department* relation in the structure of Figure, we can chain together all the records of that relation using pointers.
- When multitable clustering is to be used depends on the types of queries that the database designer believes to be most frequent. Careful use of multitable clustering can produce significant performance gains in query processing.

4.4. INDEXING AND HASHING:

- Many queries reference only a small proportion of the records in a file. For ex-ample, a query like "Find all instructors in the Physics department" or "Find the total number of credits earned by the student with *ID* 22201" references only a fraction of the student records.
- It is inefficient for the system to read every tuple in the *instructor* relation to check if the *dept name* value is "Physics". Likewise, it is inefficient to read the entire *student* relation just to find the one tuple for the *ID* "32556,". Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

Basic Concepts:

- An index for a file in a database system works in much the same way as the index in the textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed.
- Database-system indices play the same role as book indices in libraries. For example, to retrieve a *student* record given an *ID*, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate *student* record.
- Keeping a sorted list of students' *ID* would not work well on very large databases with thousands of students, since the index would itself be very big; further, even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used.

There are two basic kinds of indices:

1) Ordered indices: Based on a sorted ordering of the values.
2) Hash indices: Based on a uniform distribution of values across a range of buckets.
The bucket to which a value is assigned is determined by a function, called a *hash function.*

- We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications.

Each technique must be evaluated on the basis of these factors:

1) Access types: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.

2) Access time: The time it takes to find a particular data item, or set of items, using the technique in question.

3) Insertion time: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.

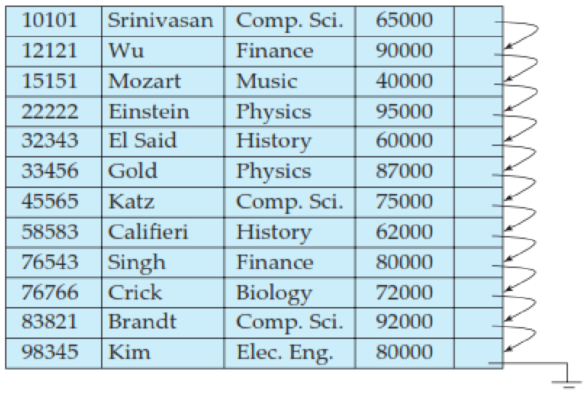
4) Deletion time: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.

5) Space overhead: The additional space occupied by an index structure. Pro-vided that the amount of additional space is moderate, it is usually worth-while to sacrifice the space to achieve improved performance.

- We often want to have more than one index for a file. For example, we may wish to search for a book by author, by subject, or by title.
- An attribute or set of attributes used to look up records in a file is called a **search key**.

<u>4.5. ORDERED INDICES:</u>

- To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key.
- An ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.
- The records in the indexed file may themselves be stored in some sorted order. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file.
- Clustering indices are also called **primary indices**; the term primary index may appear to denote an index on a primary key, but such indices can in fact be built on any search key.
- The search key of a clustering index is often the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary** indices. The terms “clustered” and “nonclustered” are often used in place of “clustering” and “nonclustering.”
- All files are ordered sequentially on some search key. Such files, with a clustering index on the search key, are called **index-sequential files**. They represent one of the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records.
- Figure shows a sequential file of *instructor* records taken from our university example. In the example of Figure , the records are stored in sorted order of instructor *ID*, which is used as the search key.



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Figure. Sequential file for *instructor* records.

<u>DENSE AND SPARSE INDICES :</u>

- An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

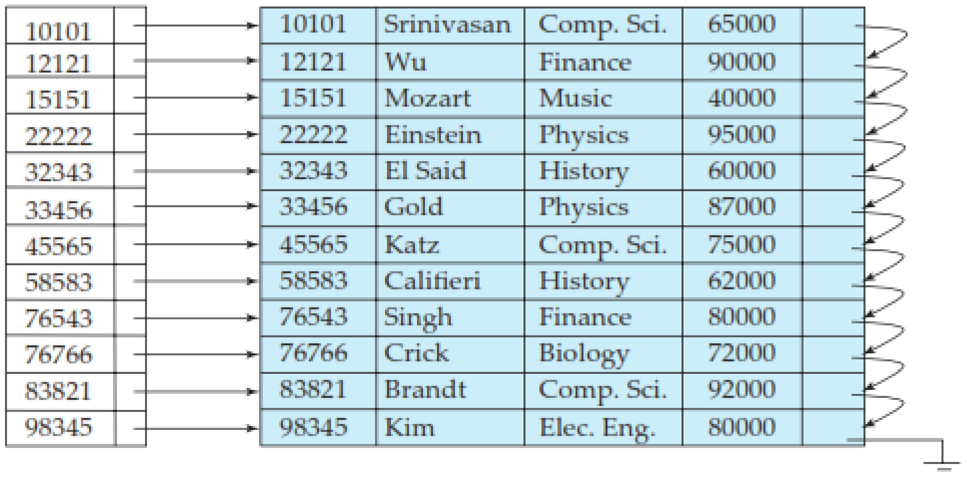


Figure. Dense index.

There are two types of ordered indices that we can use:

1) Dense index: In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key.

In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.

2) Sparse index: In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

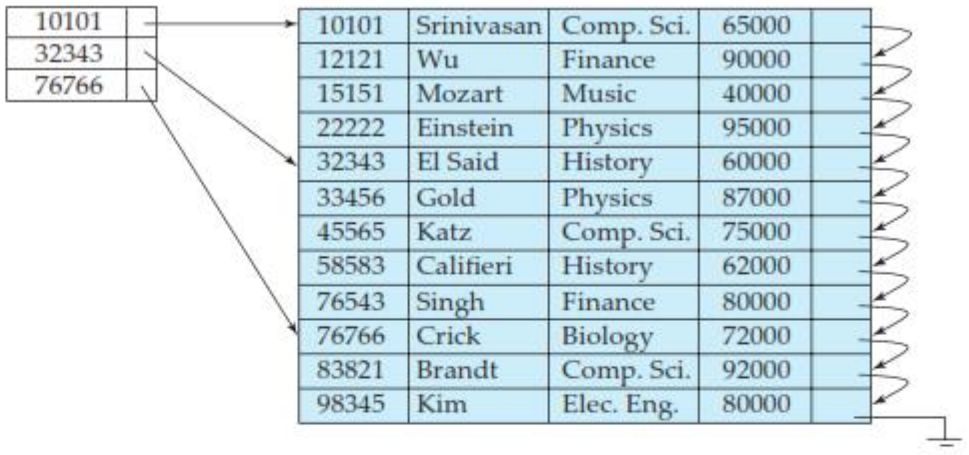
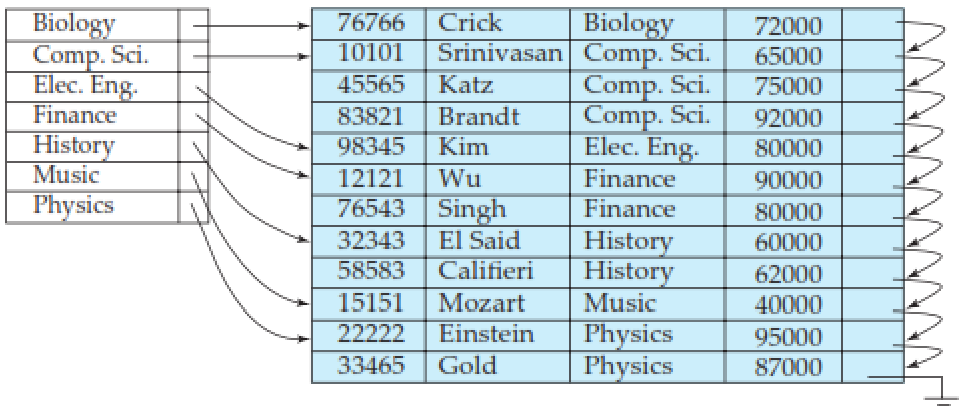


Figure. Sparse index.



76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

Figure. Dense index with search key *dept name*.

- Figures show dense and sparse indices, respectively, for the *instructor* file. Suppose that we are looking up the record of instructor with *ID* "22222". Using the dense index of Figure, we follow the pointer directly to the desired record. Since *ID* is a primary key, there exists only one such record and the search is complete. If we are using the sparse index, we do not find an index entry for "22222". Since the last entry (in numerical order) before "22222" is "10101", we follow that pointer. We then read the *instructor* file in sequential order until we find the desired record.

- Consider a (printed) dictionary. The header of each page lists the first word alphabetically on that page. The words at the top of each page of the book index together form a sparse index on the contents of the dictionary pages.

- As another example, suppose that the search-key value is not not a primary key. Figure shows a dense clustering index for the *instructor* file with the search key being *dept name*. Observe that in this case the *instructor* file is sorted on the search key *dept name*, instead of *ID*, otherwise the index on *dept name* would be a nonclustering index. Suppose that we are looking up records for the History department. Using the dense index of Figure, we follow the pointer directly to the first History record. We process this record, and follow the pointer in that record to locate the next record in search-key (*dept name*) order. We continue processing records until we encounter a record for a department other than History.
- It is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

- There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block. The reason this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block, we minimize block accesses while keeping the size of the index (and thus our space overhead) as small as possible.

- For the preceding technique to be fully general, we must consider the case where records for one search-key value occupy several blocks. It is easy to modify our scheme to handle this situation.

MULTILEVEL INDICES:

- Suppose we build a dense index on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4 kilobyte block. Thus, our index occupies 10,000 blocks. If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space. Such large indices are stored as sequential files on disk.
- If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.

- Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index would occupy *b* blocks, binary search requires as many as $\log_2(b)$ blocks to be read. (x denotes the least integer that is greater than or equal to x; that is, we round upward.) For a 10,000-block index, binary search requires 14 block reads.

- On a disk system where a block read takes on average 10 milliseconds, the index search will take 140 milliseconds. This may not seem much, but we would be able to carry out only seven index searches a second, whereas a more efficient search mechanism would let us carry out far more searches per second, as we shall see shortly. Note that, if overflow blocks have been used, binary search is not possible. In that case, a sequential search is typically used, and that requires *b* block reads, which will take even longer. Thus, the process of searching a large index may be costly.

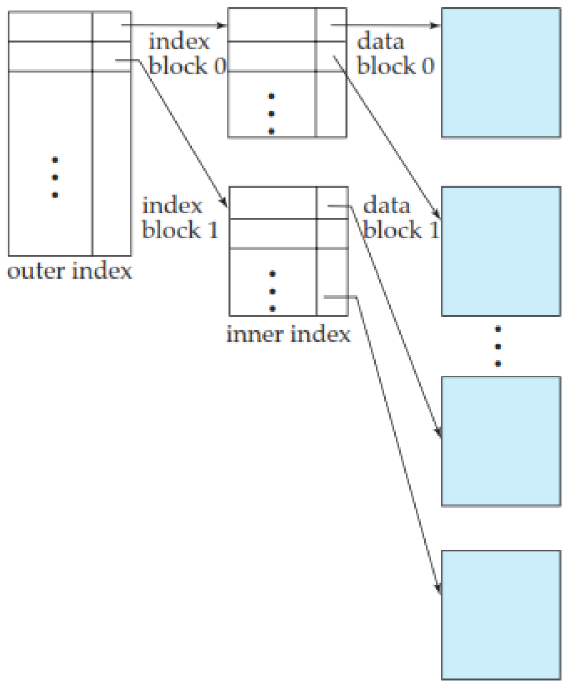


Figure. Two-level sparse index.

- To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse outer index on the original index, which we now call the inner index, as shown in Figure 11.5. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

- In our example, an inner index with 10,000 blocks would require 10,000 entries in the outer index, which would occupy just 100 blocks. If we assume that the outer index is already in main memory, we would read only one index block for a search using a multilevel index, rather than the 14 blocks we read with binary search. As a result, we can perform 14 times as many index searches per second.
- If our file is extremely large, even the outer index may grow too large to fit in main memory. With a 100,000,000 tuple relation, the inner index would occupy 1,000,000 blocks, and the outer index occupies 10,000 blocks, or 40 megabytes. Since there are many demands on main memory, it may not be possible to reserve that much main memory just for this particular outer index. In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel** indices. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search.

<u>Index Update:</u>

- Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file. Further, in case a record in the file is updated, any index whose search-key attribute is affected by the update must also be updated; for example, if the department of an instructor is changed, an index on the *dept name* attribute of *instructor* must be updated correspondingly. Such a record update can be modeled as a deletion of the old record, followed by an insertion of the new value of the record, which results in an index deletion followed by an index insertion. As a result we only need to consider insertion and deletion on an index, and do not need to consider updates explicitly.

1) Insertion. First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:

<u>Dense indices:</u>

- If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.

Otherwise the following actions are taken:

- If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry.
- Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.

<u>Sparse indices:</u>

- We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

2) Deletion. To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:

<u>Dense indices:</u>

- If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.

Otherwise the following actions are taken:

- If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index entry.
- Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.

<u>Sparse indices:</u>

- If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.

Otherwise the system takes the following actions:

- If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index rec-ord for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
- Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.
- Insertion and deletion algorithms for multilevel indices are a simple extension of the scheme just described. On deletion or insertion, the system updates the lowest-level index as described. As far as the second level is concerned, the lowest-level index is merely a file containing records—thus, if there is any change in the lowest-level index, the system updates the second-level index as described. The same technique applies to further levels of the index, if there are any.

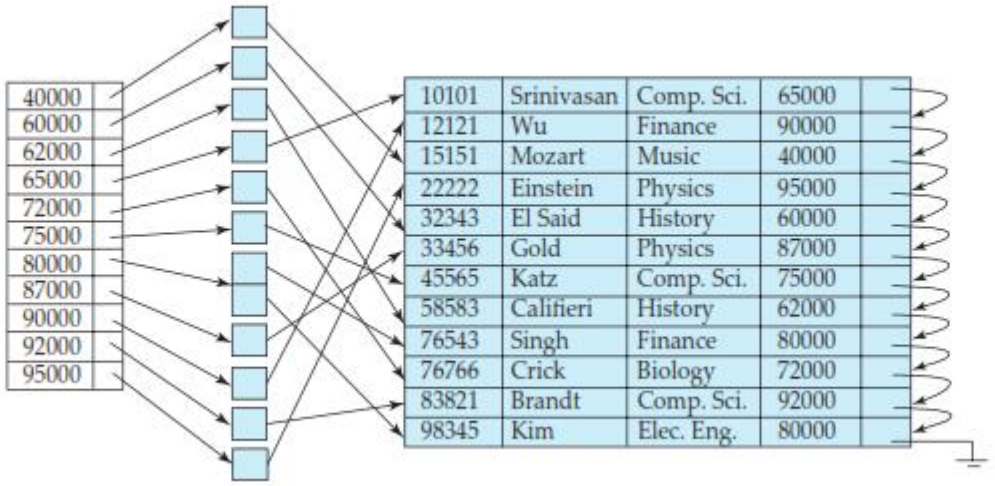
<u>SECONDARY INDICES:</u>

- Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.
- A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search key of a clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

- In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

- We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file. Figure 11.6 shows the structure of a secondary index that uses an extra level of indirection on the *instructor* file, on the search key *salary*.

- A sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered by both the search key of the clustering index and the search key of a secondary index.



40000
60000
62000
65000
72000
75000
80000
87000
90000
92000
95000
10101 Srinivasan Comp. Sci. 65000
12121 Wu Finance 90000
15151 Mozart Music 40000
22222 Einstein Physics 95000
32343 El Said History 60000
33456 Gold Physics 87000
45565 Katz Comp. Sci. 75000
58583 Califieri History 62000
76543 Singh Finance 80000
76766 Crick Biology 72000
83821 Brandt Comp. Sci. 92000
98345 Kim Elec. Eng. 80000

Figure . Secondary index on *instructor* file, on noncandidate key *salary*.

- Because secondary-key order and physical-key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.
- The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, *every* index must be updated.
- Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

<u>Indices on Multiple Keys:</u>

- Although the examples we have seen so far have had a single attribute in a search key, in general a search key can have more than one attribute. A search key containing more than one attribute is referred to as a **composite search key**. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form ($a_1, \ldots, a_n$), where the indexed attributes are $A_1, \ldots, A_n$. The ordering of search-key values is the *lexicographic ordering*. For example, for the case of two attribute search keys, $(a_1, a_2) < (b_1, b_2)$ if either $a_1 < b_1$ or $a_1 = b_1$ and $a_2 < b_2$. Lexicographic ordering is basically the same as alphabetic ordering of words.

- As an example, consider an index on the *takes* relation, on the composite search key (*course id*, *semester*, *year*). Such an index would be useful to find all students who have registered for a particular course in a particular semester/year. An ordered index on a composite key can also be used to answer several other kinds of queries efficiently.

4.6. B$^+$-TREE INDEX FILES:

- The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

- The **B$^+$ -tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B$^+$-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between $n/2$ and n children, where n is fixed for a particular tree.

- We shall see that the B$^+$-tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B$^+$-tree structure.

Structure of a B$^+$-Tree:

- A B$^+$-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure shows a typical node of a B$^+$-tree. It contains up to $n - 1$ search-key values $K_1, K_2, \ldots, K_{n-1}$, and n pointers $P_1, P_2, \ldots, P_n$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$.

- We consider first the structure of the **leaf nodes**. For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i . Pointer P_n has a special purpose that we shall discuss shortly.

- Figure shows one leaf node of a B$^+$-tree for the *instructor* file, in which we have chosen n to be 4, and the search key is *name*.
- Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $(n - 1)/2$ values. With $n = 4$ in our example B$^+$-tree, each leaf must contain at least 2 values, and at most 3 values.

- The ranges of values in each leaf do not overlap, except if there are duplicate search-key values, in which case a value may be present in more than one leaf. Specifically, if L_i and L_j are leaf nodes and $i < j$, then every search-key value in L_i is less than or equal to every search-key value in L_j . If the B$^+$-tree index is used as a dense index (as is usually the case) every search-key value must appear in some leaf node.

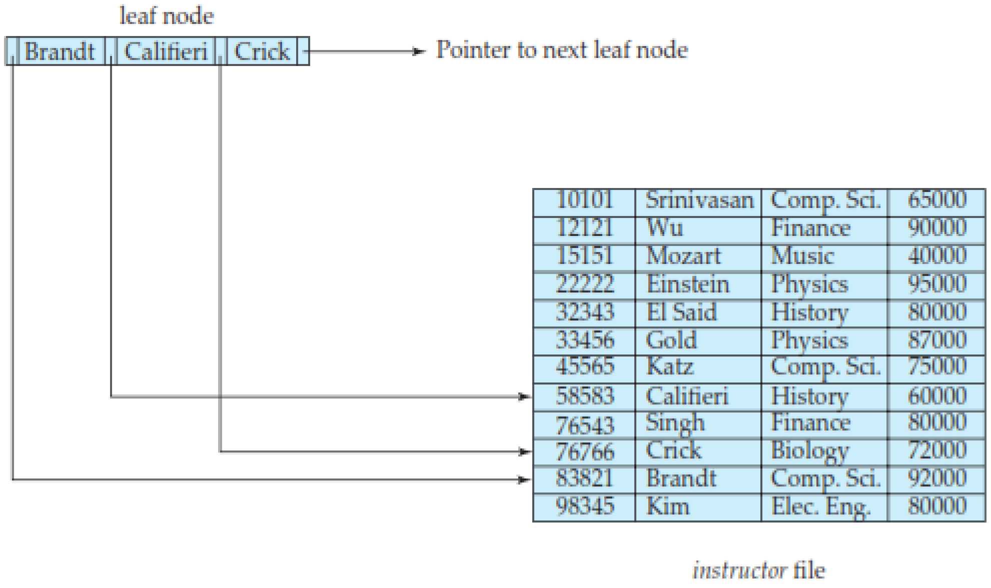
P_1	K_1	P_2	É	P_{n-1}	K_{n-1}	P_n

Figure. Typical node of a B$^+$ -tree.

- Now we can explain the use of the pointer P_n. Since there is a linear order on the leaves based on the search-key values that they contain, we use P_n to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

- The **nonleaf nodes** of the B$^+$-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and *must* hold at least $n/2$ pointers. The number of pointers in a node is called the *fanout* of the node. Nonleaf nodes are also referred to as **internal nodes**.

- Let us consider a node containing m pointers ($m \leq n$). For $i = 2, 3, \ldots, m - 1$, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1}, and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1.

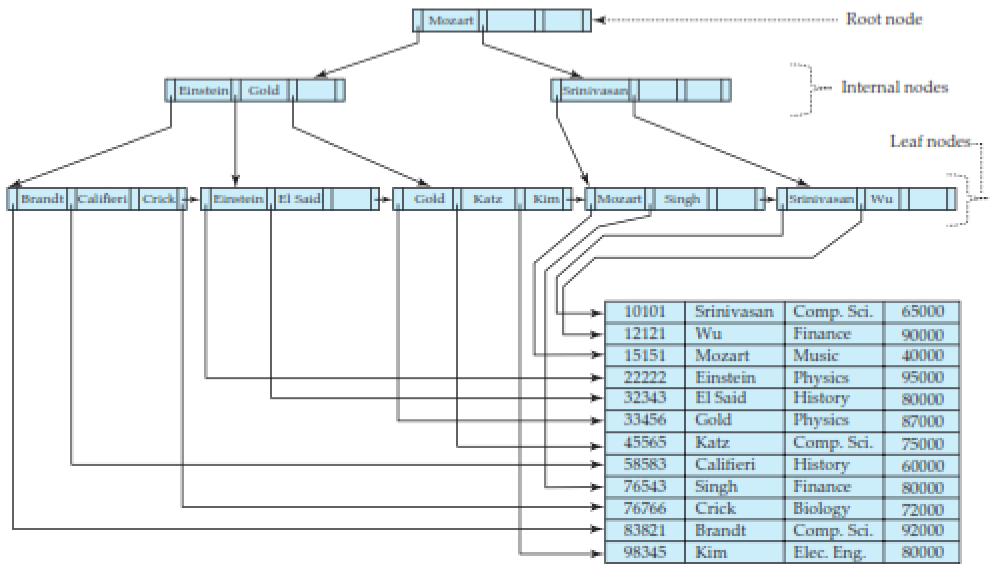


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure. A leaf node for *instructor* B^+ -tree index ($n = 4$).

- Unlike other nonleaf nodes, the root node can hold fewer than $n/2$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B^+-tree, for any n, that satisfies the preceding requirements.

- Figure shows a complete B^+-tree for the *instructor* file (with $n = 4$). We have shown instructor names abbreviated to 3 characters in order to depict the tree clearly; in reality, the tree nodes would contain the full names. We have also omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure . B^+ -tree for *instructor* file (n =4).

- Figure shows another B^+-tree for the *instructor* file, this time with n = 6. As before, we have abbreviated instructor names only for clarity of presentation.

- Observe that the height of this tree is less than that of the previous tree, which had $n = 4$.
- These examples of B^+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a B^+-tree. Indeed, the "B" in B^+ -tree stands for "balanced." It is the balance property of B^+-trees that ensures good performance for lookup, insertion, and deletion.

<u>Queries on B^+-Trees:</u>

- Let us consider how we process queries on a B^+-tree. Suppose that we wish to find records with a search-key value of V. Figure 11.11 presents pseudocode for a function find() to carry out this task.

- Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest i such that search-key value K_i is greater than or equal to V. Suppose such a value is found; then, if K_i is equal to V, the current node is set to the node pointed to by P_{i+1}, otherwise $K_i > V$, and the current node is set to the node pointed to by P_i . If no such value K_i is found, then clearly $V > K_{m-1}$, where P_m is the last nonnull pointer in the node. In this case the current node is set to that pointed to by P_m. The above procedure is repeated, traversing down the tree until a leaf node is reached.

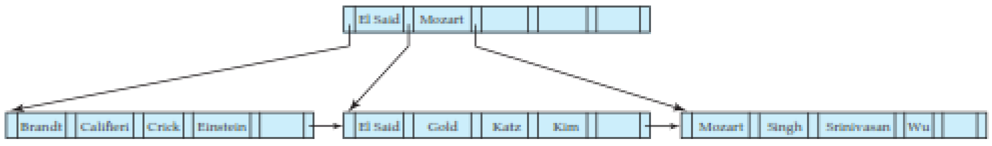


Figure. B^+ -tree for *instructor* file with n = 6.

```
function find(value $V$)
/* Returns leaf node $C$ and index $i$ such that $C.P_i$ points to first record
    with search key value $V$ */
Set $C$ = root node
while ($C$ is not a leaf node) begin
    Let $i$ = smallest number such that $V \leq C.K_i$
    if there is no such number $i$ then begin
        Let $P_m$ = last non-null pointer in the node
        Set $C = C.P_m$
    end
    else if ($V = C.K_i$ )
        then Set $C = C.P_{i+1}$
    else $C = C.P_i$ /* $V < C.K_i$ */
end
/* $C$ is a leaf node */
Let $i$ be the least value such that $K_i = V$
if there is such a value $i$
    then return ($C$, $i$)
    else return null ; /* No record with key value $V$ exists*/


procedure printAll(value $V$)

/* prints all records with search key value $V$ */
Set done = false;
Set ($L$, $i$) = find($V$);
if (($L$ , $i$) is null) return
repeat
    repeat
        Print record pointed to by $L.P_i$
        Set $i = i + 1$
    until ($i$ > number of keys in $L$ or $L.K_i > V$)
    if ($i$ > number of keys in $L$)
        then $L = L.P_n$
        else Set done = true;
until (done or $L$ is null)
```

Figure. Querying a B^+ -tree.

- At the leaf node, if there is a search-key value equal to V, let K_i be the first such value; pointer P_i directs us to a record with search-key value K_i . The function
- then returns the leaf node L and the index i. If no search-key with value V is found in the leaf node, no record with key value V exists in the relation, and function find returns null, to indicate failure.

- If there is at most one record with a search key value V (for example, if the index is on a primary key) the procedure that calls the find function simply uses the pointer $L.P_i$ to retrieve the record and is done. However, in case there may be more than one matching record, the remaining records also need to be fetched.

- Procedure printAll shown in Figur shows how to fetch all records with a specified search key *V*. The procedure first steps through the remaining keys in the node *L*, to find other records with search-key value *V*. If node *L* contains at least one search-key value greater than *V*, then there are no more records matching *V*. Otherwise, the next leaf, pointed to by P_n may contain further entries for *V*. The node pointed to by P_n must then be searched to find further records with search-key value *V*. If the highest search-key value in the node pointed to by P_n is also *V*, further leaves may have to be traversed, in order to find all matching records. The **repeat** loop in printAll carries out the task of traversing leaf nodes until all matching records have been found.

- A real implementation would provide a version of find supporting an iterator interface similar to that provided by the JDBC ResultSet. Such an iterator interface would provide a method next(), which can be called repeatedly to fetch successive records with the specified search-key. The next() method would step through the entries at the leaf level, in a manner similar to printAll, but each call takes only one step, and records where it left off, so that successive calls next step through successive records. We omit details for simplicity, and leave the pseudocode for the iterator interface as an exercise for the interested reader.

- B^+-trees can also be used to find all records with search key values in a specified range (*L, U*). For example, with a B^+-tree on attribute *salary* of *instruc-tor*, we can find all *instructor* records with salary in a specified range such as (50000, 100000) (in other words, all salaries between 50000 and 100000). Such queries are called **range queries**. To execute such queries, we can create a proce-dure printRange(*L, U*), whose body is the same as printAll except for these dif-ferences: printRange calls find(*L*), instead of find(*V*), and then steps through records as in procedure printAll, but with the stopping condition being that $L.K_i > U$, instead of $L.K_i > V$.

- In processing a query, we traverse a path in the tree from the root to some leaf node. If there are *N* records in the file, the path is no longer than $\log_{n/2}(N)$.

- In practice, only a few nodes need to be accessed. Typically, a node is made to be the same size as a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of 8 bytes, *n* is around 200. Even with a more conservative estimate of 32 bytes for the search-key size, *n* is around 100. With $n = 100$, if we have 1 million search-key values in the file, a lookup requires only $\log_{50}(1{,}000{,}000) = 4$ nodes to be accessed. Thus, at most four blocks need to be read from disk for the lookup. The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.
- An important difference between B^+-tree structures and in-memory tree struc-tures, such as binary trees, is the size of a node, and as a result, the height of the tree. In a binary tree, each node is small, and has at most two pointers. In a B^+-tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus, B^+-trees tend to be fat and short, unlike thin and tall binary trees. In a balanced binary tree, the path for a lookup can be of length $\log_2(N)$, where *N* is the number of records in the file being indexed. With $N = 1{,}000{,}000$ as in the previous example, a balanced binary tree requires around 20 node accesses. If each node were on a different disk block, 20 block reads would be required to process a lookup, in contrast to the four block reads for the B^+-tree. The difference is significant, since each block read could require a disk arm seek, and a block read together with the disk arm seek takes about 10 milliseconds on a typical disk.

Updates on B^+-Trees:

- When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly. Recall that updates to a record can be modeled as a deletion of the old record followed by insertion of the updated record. Hence we only consider the case of insertion and deletion.

- Insertion and deletion are more complicated than lookup, since it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce** nodes (that is, combine nodes) if a node becomes too small (fewer than *n*/2 pointers). Furthermore, when a node is split or a pair of nodes is combined, we must ensure that balance is preserved. To introduce the idea behind insertion and deletion in a B^+-tree, we shall assume temporarily that nodes never become too large or too small. Under this assumption, insertion and deletion are performed as defined next.

Insertion. Using the same technique as for lookup from the find() function (Figure 11.11), we first find the leaf node in which the search-key value would appear. We then insert an entry (that is, a search-key value and record pointer pair) in the leaf node, positioning it such that the search keys are still in order.

Deletion. Using the same technique as for lookup, we find the leaf node containing the entry to be deleted, by performing a lookup on the search-key value of the deleted record; if there are multiple entries with the same search-key value, we search across all entries with the same search-key value until we find the entry that points to the record being deleted. We then remove the entry from the leaf node. All entries in the leaf node that are to the right of the deleted entry are shifted left by one position, so that there are no gaps in the entries after the entry is deleted.

We now consider the general case of insertion and deletion, dealing with node splitting and node coalescing.

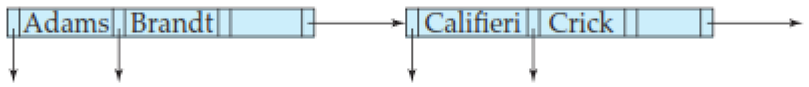


Figure. Split of leaf node on insertion of "Adams"

<u>Insertion:</u>

- We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for "Adams" into the B^+ -tree of Figure 11.9. Using the algorithm for lookup, we find that "Adams" should appear in the leaf node containing "Brandt", "Califieri", and "Crick." There is no room in this leaf to insert the search-key value "Adams." Therefore, the node is *split* into two nodes. Figure shows the two leaf nodes that result from the split of the leaf node on inserting "Adams". The search-key values "Adams" and "Brandt" are in one leaf, and "Califieri" and "Crick" are in the other. In general, we take the *n* search-key values (the $n - 1$ values in the leaf node plus the value being inserted), and put the first *n*/2 in the existing node and the remaining values in a newly created node.

- Having split a leaf node, we must insert the new leaf node into the B^+-tree structure. In our example, the new node has "Califieri" as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The B^+-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

- Splitting of a nonleaf node is a little different from splitting of a leaf node. Figure shows the result of inserting a record with search key "Lamport" into the tree shown in Figure. The leaf node in which "Lamport" is to be inserted already has entries "Gold", "Katz", and "Kim", and as a result the leaf node has to be split. The new right-hand-side node resulting from the split contains the search-key values "Kim" and "Lamport". An entry (Kim, *n*1) must then be added to the parent node, where *n*1 is a pointer to the new node, However, there is no space in the parent node to add a new entry, and the parent node has to be split. To do so, the parent node is conceptually expanded temporarily, the entry added, and the overfull node is then immediately split.

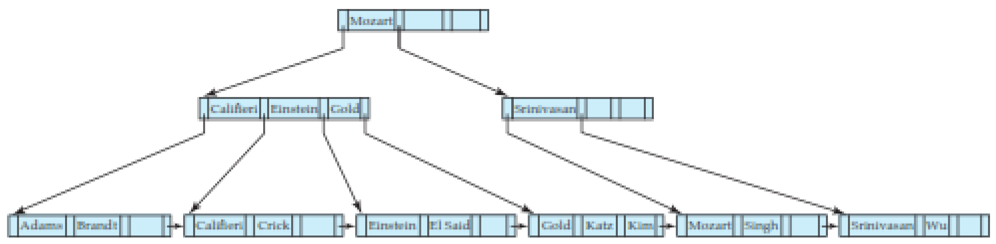


Figure. Insertion of “Adams” into the B$^+$ -tree of Figure 11.9.

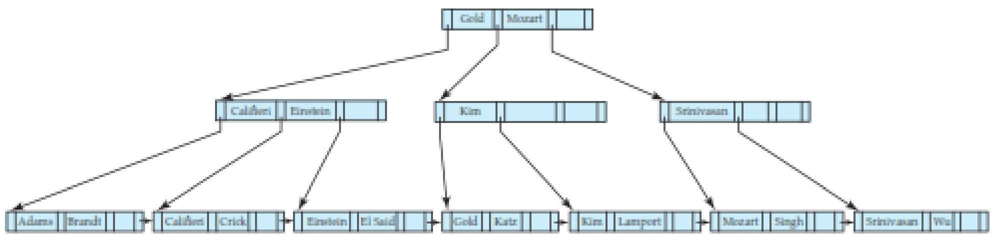


Figure. Insertion of “Lamport” into the B$^+$ -tree of Figure 11.13.

- When an overfull nonleaf node is split, the child pointers are divided among the original and the newly created nodes; in our example, the original node is left with the first three pointers, and the newly created node to the right gets the remaining two pointers. The search key values are, however, handled a little differently. The search key values that lie between the pointers moved to the right node (in our example, the value “Kim”) are moved along with the pointers, while those that lie between the pointers that stay on the left (in our example, “Califieri” and “Einstein”) remain undisturbed.

- However, the search key value that lies between the pointers that stay on the left, and the pointers that move to the right node is treated differently. In our example, the search key value “Gold” lies between the three pointers that went to the left node, and the two pointers that went to the right node. The value “Gold” is not added to either of the split nodes. Instead, an entry (Gold, *n*2) is added to the parent node, where *n*2 is a pointer to the newly created node that resulted from the split. In this case, the parent node is the root, and it has enough space for the new entry.

- The general technique for insertion into a B$^+$-tree is to determine the leaf node *l* into which insertion must occur. If a split results, insert the new node into the parent of node *l*. If this insertion causes a split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.

- Figure 11.15 outlines the insertion algorithm in pseudocode. The procedure insert inserts a key-value pointer pair into the index, using two subsidiary procedures insert in leaf and insert in parent. In the pseudocode, *L* , *N*, *P* and *T* denote pointers to nodes, with *L* being used to denote a leaf node. $L.K_i$ and $L.P_i$ denote the *i*th value and the *i*th pointer in node *L*, respectively; $T.K_i$ and $T.P_i$ are used similarly. The pseudocode also makes use of the function parent(*N*) to find the parent of a node *N*. We can compute a list of nodes in the path from the root to the leaf while initially finding the leaf node, and can use it later to find the parent of any node in the path efficiently.

- The procedure insert in parent takes as parameters *N*, *K* , *N* , where node *N* was split into *N* and *N* , with *K* being the least value in *N* . The procedure modifies the parent of *N* to record the split. The procedures insert into index and insert in parent use a temporary area of memory *T* to store the contents of a node being split. The procedures can be modified to copy data from the node being split directly to the newly created node, reducing the time required for copying data. However, the use of the temporary space *T* simplifies the procedures.

```
**procedure** insert(*value K , poi nter P*)

**if** (tree is empty) create an empty leaf node $L$, which is also the root **else** Find the leaf node $L$
that should contain key value $K$

**if** ($L$ has less than $n - 1$ key values)
**then** insert in leaf ($L$ , $K$ , $P$)     – –

**else begin** /* $L$ has $n - 1$ key values already, split it */ Create node $L$

Copy $L .P_1 . . . L .K_{n-1}$ to a block of memory $T$ that can hold $n$ (pointer, key-value) pairs
insert in leaf ($T$, $K$ , $P$)          – –

Set $L .P_n = L .P_n$; Set $L .P_n = L$
Erase $L .P_1$ through $L .K_{n-1}$ from $L$
Copy $T.P_1$ through $T.K_{n/2}$ from $T$ into $L$ starting at $L.P_1$
Copy $T.P_{n/2+1}$ through $T.K_n$ from $T$ into $L$ starting at $L .P_1$
Let $K$ be the smallest key-value in $L$
insert in parent($L$, $K$ , $L$ )        – –

**end**

**procedure** insert in leaf (*node L, value K , poi nter P*)

**if** ($K < L .K_1$)
**then** insert $P$, $K$ into $L$ just before $L .P_1$
**else begin**
Let $K_i$ be the highest value in $L$ that is less than $K$ Insert $P$, $K$ into $L$ just after $T.K_i$
**end**

**procedure** insert in parent(*node N, value K , node N* )

**if** ($N$ is the root of the tree)
**then begin**
Create a new node $R$ containing $N$, $K$ , $N$ /* $N$ and $N$ are pointers */ Make $R$ the root of the tree
**return**
**end**
Let $P$ = *parent* ($N$)
**if** ($P$ has less than $n$ pointers)
**then** insert ($K$ , $N$ ) in $P$ just after $N$
**else begin** /* Split $P$ */

Copy $P$ to a block of memory $T$ that can hold $P$ and ($K$ , $N$ )
Insert ($K$ , $N$ ) into $T$ just after $N$
```

```
Erase all entries from $P$; Create node $P$
Copy $T.P_1 \ldots T.P_{n/2}$ into $P$
Let $K$ = $T.K_{n/2}$
Copy $T.P_{n/2+1} \ldots T.P_{n+1}$ into $P$
insert in parent($P$, $K$ , $P$ )

end
```

Figure. Insertion of entry in a B$^+$ -tree.

Deletion:

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete "Srinivasan" from the B$^+$-tree of Figure 11.13. The resulting B$^+$-tree appears in Figure. We now consider how the deletion is performed. We first locate the entry for "Srinivasan" by using our lookup algorithm. When we delete the entry for "Srinivasan" from its leaf node, the node is left with only one entry, "Wu". Since, in our example, $n = 4$ and $1 < (n - 1)/2$, we must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for "Wu" can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

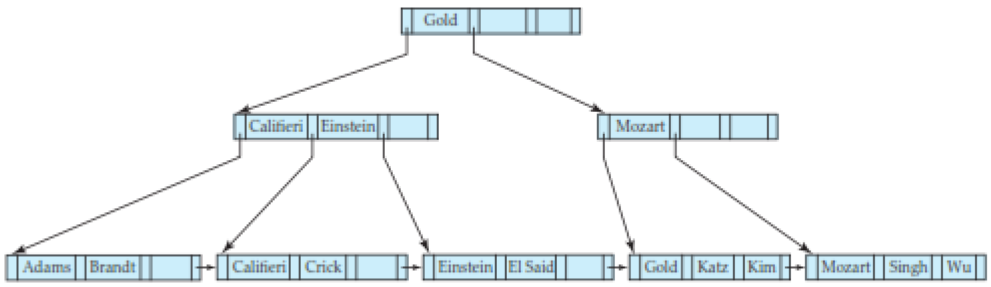


Figure. Deletion of "Srinivasan" from the B$^+$ -tree of Figure 11.13.

- In our example, the entry to be deleted is (Srinivasan, $n3$), where $n3$ is a pointer to the leaf containing "Srinivasan". (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.)
- After deleting the above entry, the parent node, which had a search key value "Srinivasan" and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since $1 < n/2$ for $n = 4$, the parent node is underfull. (For larger n, a node that becomes underfull would still have some values as well as pointers.)

- In this case, we look at a sibling node; in our example, the only sibling is the nonleaf node containing the search keys "Califieri", "Einstein", and "Gold". If possible, we try to coalesce the node with its sibling. In this case, coalescing is not possible, since the node and its sibling together have five pointers, against a maximum of four. The solution in this case is to **redistribute** the pointers between the node and its sibling, such that each has at least $n/2 = 2$ child pointers. To do so, we move the rightmost pointer from the left sibling (the one pointing to the leaf node containing "Mozart") to the underfull right sibling. However, the underfull right sibling would now have two pointers, namely its leftmost pointer, and the newly moved pointer, with no value separating them.

- In fact, the value separating them is not present in either of the nodes, but is present in the parent node, between the pointers from the parent to the node and its sibling. In our example, the value “Mozart” separates the two pointers, and is present in the right sibling after the redistribution. Redistribution of the pointers also means that the value “Mozart” in the parent no longer correctly separates search-key values in the two siblings. In fact, the value that now correctly separates search-key values in the two sibling nodes is the value “Gold”, which was in the left sibling before redistribution.

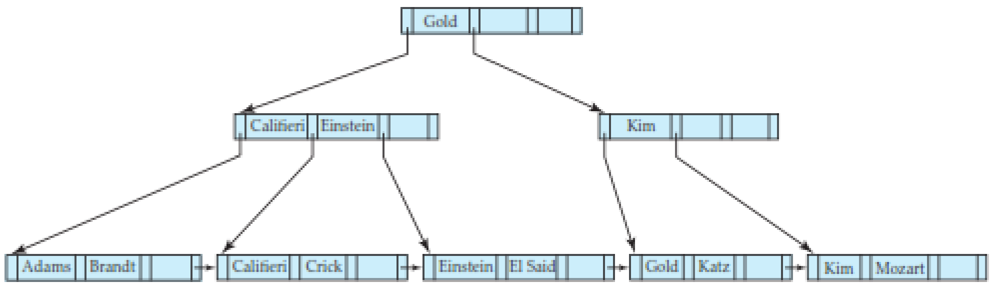


Figure. Deletion of “Singh” and “Wu” from the B$^+$ -tree.

- As a result, as can be seen in the B$^+$-tree in Figure 11.16, after redistribution of pointers between siblings, the value “Gold” has moved up into the parent, while the value that was there earlier, “Mozart”, has moved down into the right sibling.
- We next delete the search-key values “Singh” and “Wu” from the B$^+$-tree of Figure. The deletion of the first of these values does not make the leaf node underfull, but the deletion of the second value does. It is not possible to merge the underfull node with its sibling, so a redistribution of values is carried out, moving the search-key value “Kim” into the node containing “Mozart”, resulting in the tree shown in Figure. The value separating the two siblings has been updated in the parent, from “Mozart” to “Kim”.
- Now we delete “Gold” from the above tree; the result is shown in Figure. This results in an underfull leaf, which can now be merged with its sibling. The resultant deletion of an entry from the parent node (the nonleaf node containing “Kim”) makes the parent underfull (it is left with just one pointer). This time around, the parent node can be merged with its sibling. This merge results in the search-key value “Gold” moving down from the parent into the merged node. As a result of this merge, an entry is deleted from its parent, which happens to be the root of the tree. And as a result of that deletion, the root is left with only one child pointer and no search-key value, violating the condition that the root have at least two children. As a result, the root node is deleted and its sole child becomes the root, and the depth of the B$^+$-tree has been decreased by 1.

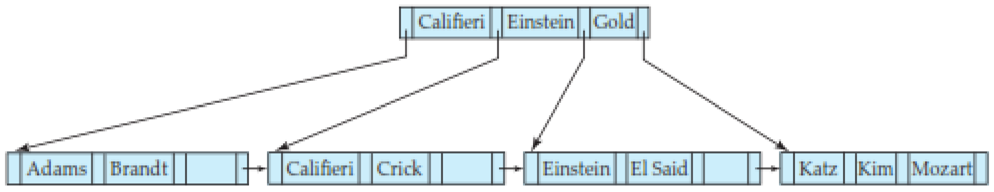


Figure. Deletion of “Gold” from the B$^+$ -tree of Figure 11.17.

- It is worth noting that, as a result of deletion, a key value that is present in a nonleaf node of the B$^+$-tree may not be present at any leaf of the tree. For example, in Figure, the value “Gold” has been deleted from the leaf level, but is still present in a nonleaf node.

- In general, to delete a value in a B$^+$-tree, we perform a lookup on the value and delete it. If the node is too small, we delete it from its parent. This deletion results in recursive application of the deletion algorithm until the root is reached, a parent remains adequately full after deletion, or redistribution is applied.

- Figure outlines the pseudocode for deletion from a B$^+$-tree. The procedure swap variables(*N*, *N*) merely swaps the values of the (pointer) variables *N* and *N* ; this swap has no effect on the tree itself. The pseudocode uses the condition "too few pointers/values." For nonleaf nodes, this criterion means less than *n/*2 pointers; for leaf nodes, it means less than (*n* − 1)*/*2 values. The pseudocode redistributes entries by borrowing a single entry from an adjacent node. We can also redistribute entries by repartitioning entries equally between the two nodes. The pseudocode refers to deleting an entry (*K* , *P*) from a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the pointer *P* precedes the key value *K* . For nonleaf nodes, *P* follows the key value *K* .

```
procedure delete(value K , poi nter P)

find the leaf node L that contains (K , P)
delete entry(L , K , P)

procedure delete entry(node N, value K , poi nter P)
delete (K , P) from N
if (N is the root and N has only one remaining child)
then make the child of N the new root of the tree and delete N else if (N has too few
values/pointers) then begin
Let N be the previous or next child of parent (N)
Let K be the value between pointers N and N in parent (N)
if (entries in N and N can fit in a single node)
then begin /* Coalesce nodes */
if (N is a predecessor of N ) then swap variables(N, N )
if (N is not a leaf)
then append K and all pointers and values in N to N
else append all ($K_i$ , $P_i$ ) pairs in N to N ; set $N.P_n = N.P_n$ delete entry(parent (N), K , N); delete node N
end
else begin /* Redistribution: borrow an entry from N */ if (N is a predecessor of N) then begin
                              if (N is a nonleaf node) then begin
let m be such that $N.P_m$ is the last pointer in N remove ($N.K_{m-1}$, $N.P_m$) from N
insert ($N.P_m$, K ) as the first pointer and value in N, by shifting other pointers and values right
replace K in parent (N) by $N.K_{m-1}$
end
else begin
let m be such that ($N.P_m$, $N.K_m$) is the last pointer/value pair in N
remove ($N.P_m$, $N.K_m$) from N
insert ($N.P_m$, $N.K_m$) as the first pointer and value in N, by shifting other pointers and values right
replace K in parent (N) by $N.K_m$
end
end
else . . . symmetric to the then case . . .
end
end
```

Figure. Deletion of entry from a B$^+$ -tree.

<u>Nonunique Search Keys:</u>

- If a relation can have more than one record containing the same search key value (that is, two or more records can have the same values for the indexed attributes), the search key is said to be a **nonunique search key**.
- One problem with nonunique search keys is in the efficiency of record dele-tion. Suppose a particular search-key value occurs a large number of times, and one of the records with that search key is to be deleted. The deletion may have to search through a number of entries, potentially across multiple leaf nodes, to find the entry corresponding to the particular record being deleted.
- A simple solution to this problem, used by most database systems, is to make search keys unique by creating a composite search key containing the original search key and another attribute, which together are unique across all records.
- The extra attribute can be a record-id, which is a pointer to the record, or any other attribute whose value is unique among all records with the same search-key value. The extra attribute is called a **uniquifier** attribute.
- When a record is to be deleted, the composite search-key value is computed from the record, and then used to look up the index. Since the value is unique, the corresponding leaf- level entry can be found with a single traversal from root to leaf, with no further accesses at the leaf level. As a result, record deletion can be done efficiently.
- A search with the original search-key attribute simply ignores the value of the uniquifier attribute when comparing search-key values.
- With nonunique search keys, our B^+-tree structure stores each key value as many times as there are records containing that value. An alternative is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle nonunique search keys. This approach is more space efficient since it stores the key value only once; however, it creates several complications when B^+-trees are implemented.
- If the buckets are kept in the leaf node, extra code is needed to deal with variable-size buckets, and to deal with buckets that grow larger than the size of the leaf node. If the buckets are stored in separate blocks, an extra I/O operation may be required to fetch records. In addition to these problems, the bucket approach also has the problem of inefficiency for record deletion if a search-key value occurs a large number of times.

<u>Complexity of B^+-Tree Updates:</u>

- Although insertion and deletion operations on B^+-trees are complicated, they require relatively few I/O operations, which is an important benefit since I/O operations are expensive. It can be shown that the number of I/O operations needed in the worst case for an insertion is proportional to $\log_{n/2}(N)$, where n is the maximum number of pointers in a node, and N is the number of records in the file being indexed.

- The worst-case complexity of the deletion procedure is also proportional to $\log_{n/2}(N)$, provided there are no duplicate values for the search key. If there are duplicate values, deletion may have to search across multiple records with the same search-key value to find the correct entry to be deleted, which can be inefficient. However, making the search key unique by adding a uniquifier attribute, ensures the worst-case complexity of deletion is the same even if the original search key is nonunique.

- In other words, the cost of insertion and deletion operations in terms of I/O operations is proportional to the height of the B^+-tree, and is therefore low. It is the speed of operation on B^+-trees that makes them a frequently used index structure in database implementations.

- In practice, operations on B^+-trees result in fewer I/O operations than the worst-case bounds. With fanout of 100, and assuming accesses to leaf nodes are uniformly distributed, the parent of a leaf node is 100 times more likely to get accessed than the leaf node. Conversely, with the same fanout, the total number of nonleaf nodes in a B^+-tree would be just a little more than 1/100th of the number of leaf nodes. As a result, with memory sizes of several gigabytes being common today, for B^+-trees that are used frequently, even if the relation is very large it is quite likely that most of the nonleaf nodes are already in the database buffer when they are accessed. Thus, typically only one or two I/O operations are required to perform a lookup.

4.7. B-TREE INDEX FILES:

- **B-tree indices** are similar to B^+-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. In the B^+-tree of Figure, the search keys "Califieri", "Einstein", "Gold", "Mozart", and "Srinivasan" appear in nonleaf nodes, in addition to appearing in the leaf nodes. Every search-key value appears in some leaf node; several are repeated in nonleaf nodes.

- A B-tree allows search-key values to appear only once (if they are unique), unlike a B^+-tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node. Figure shows a B-tree that represents the same search keys as the B^+-tree of Figure. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding B^+-tree index. However, since search keys that appear in nonleaf nodes appear nowhere else in the B-tree, we are forced to include an additional pointer field for each search key in a nonleaf node. These additional pointers point to either file records or buckets for the associated search key.

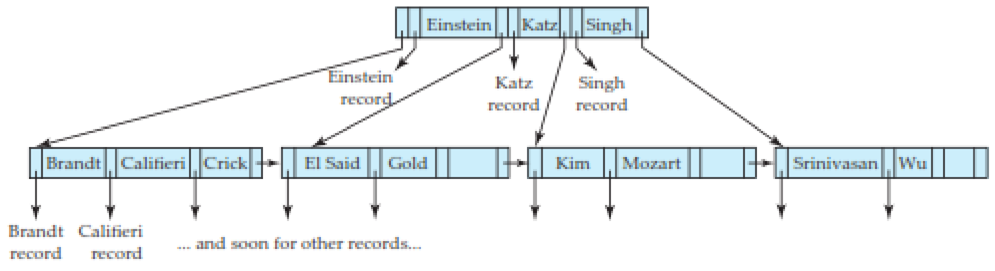


Figure. B-tree equivalent of B^+ -tree

- It is worth noting that many database system manuals, articles in industry literature, and industry professionals use the term B-tree to refer to the data structure that we call the B^+-tree. In fact, it would be fair to say that in current usage, the term B-tree is assumed to be synonymous with B^+-tree. However, in this book we use the terms B-tree and B^+-tree as they were originally defined, to avoid confusion between the two data structures.

- A generalized B-tree leaf node appears in Figure (a); a nonleaf node appears in Figure (b). Leaf nodes are the same as in B^+-trees. In nonleaf nodes, the pointers P_i are the tree pointers that we used also for B^+-trees, while the pointers B_i are bucket or file-record pointers. In the generalized B-tree in the figure, there are $n - 1$ keys in the leaf node, but there are $m - 1$ keys in the nonleaf node. This discrepancy occurs because nonleaf nodes must include pointers B_i , thus reducing the number of search keys that can be held in these nodes. Clearly, $m < n$, but the exact relationship between m and n depends on the relative size of search keys and pointers.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is located. A lookup on a B$^+$-tree requires traversal of a path from the root of the tree to some leaf node. In contrast, it is sometimes possible to find the desired value in a B-tree before reaching a leaf node. However, roughly *n* times as many keys are stored in the leaf level of a B-tree as in the nonleaf levels, and, since *n* is typically large, the benefit of finding certain values early is relatively small. Moreover, the fact that fewer search keys appear in a nonleaf B-tree node, compared to B$^+$-trees, implies that a B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding B$^+$-tree. Thus, lookup in a B-tree is faster for some search keys but slower for others, although, in general, lookup time is still proportional to the logarithm of the number of search keys.

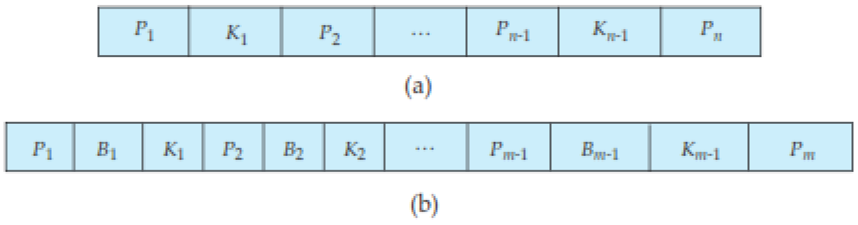


Figure . Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

- Deletion in a B-tree is more complicated. In a B$^+$-tree, the deleted entry always appears in a leaf. In a B-tree, the deleted entry may appear in a nonleaf node. The proper value must be selected as a replacement from the subtree of the node containing the deleted entry. Specifically, if search key K_i is deleted, the smallest search key appearing in the subtree of pointer P_{i+1} must be moved to the field formerly occupied by K_i . Further actions need to be taken if the leaf node now has too few entries. In contrast, insertion in a B-tree is only slightly more complicated than is insertion in a B$^+$-tree.

- The space advantages of B-trees are marginal for large indices, and usually do not outweigh the disadvantages that we have noted. Thus, pretty much all database-system implementations use the B$^+$-tree data structure, even if (as we discussed earlier) they refer to the data structure as a B-tree.

4.8 . STATIC HASHING:

- One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of **hashing** allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices. .

- The term **bucket is used** to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

- Let *K* denote the set of all search-key values, and let *B* denote the set of all bucket addresses. A **hash function** *h* is a function from *K* to *B*. Let *h* denote a hash function.

- To insert a record with search key K_i , we compute $h(K_i)$, which gives the address of the bucket for that record. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket.

- To perform a lookup on a search-key value K_i , we simply compute $h(K_i)$, then search the bucket with that address. Suppose that two search keys, K_5 and K_7, have the same hash value; that is, $h(K_5) = h(K_7)$. If we perform a lookup on K_5, the bucket $h(K_5)$ contains records with search-key values K_5 and records with search-key values K_7. Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.

- Deletion is equally straightforward. If the search-key value of the record to be deleted is K_i , we compute $h(K_i)$, then search the corresponding bucket for that record, and delete the record from the bucket.

- Hashing can be used for two different purposes. In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record. In a **hash index organization** we organize the search keys, with their associated pointers, into a hash file structure.

Hash Functions:

- The worst possible hash function maps all search-key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired. An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.
- Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key values to buckets in such a way that the distribution has these qualities:

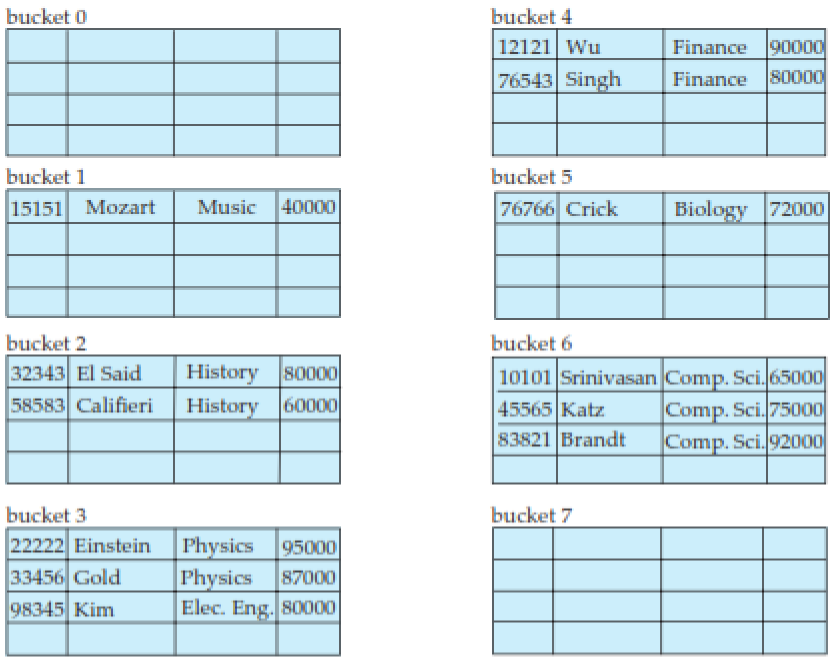
- The distribution is *uniform*. That is, the hash function assigns each bucket the same number of search-key values from the set of *all* possible search-key values.

- The distribution is *random*. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.

- As an illustration of these principles, let us choose a hash function for the *instructor* file using the search key *dept name*. The hash function that we choose must have the desirable properties not only on the example *instructor* file that we have been using, but also on an *instructor* file of realistic size for a large university with many departments.

- Assume that we decide to have 26 buckets, and we define a hash function that maps names beginning with the *i*th letter of the alphabet to the *i*th bucket. This hash function has the virtue of simplicity, but it fails to provide a uniform distribution, since we expect more names to begin with such letters as B and R than Q and X, for example.
- Now suppose that we want a hash function on the search key *salary*. Suppose that the minimum salary is \$30,000 and the maximum salary is \$130,000, and we use a hash function that divides the values into 10 ranges, \$30,000 – \$40,000, \$40,001 – \$50,000 and so on. The distribution of search-key values is uniform (since each bucket has the same number of different *salary* values), but is not random. Records with salaries between \$60,001 and \$70,000 are far more common than are records with salaries between \$30,001 and \$40,000.

- As a result, the distribution of records is not uniform — some buckets receive more records than others do. If the function has a random distribution, even if there are such correlations in the search keys, the randomness of the distribution will make it very likely that all buckets will have roughly the same number of records, as long as each search key occurs in only a small fraction of the records. (If a single search key occurs in a large fraction of the records, the bucket containing it is likely to have more records than other buckets, regardless of the hash function used.)



bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Figure. Hash organization of *instructor* file, with *dept name* as the key.

- Typical hash functions perform computation on the internal binary machine representation of characters in the search key. A simple hash function of this type first computes the sum of the binary representations of the characters of a key, then returns the sum modulo the number of buckets.

- Figure shows the application of such a scheme, with eight buckets, to the *instructor* file, under the assumption that the *i*th letter in the alphabet is represented by the integer *i*.

- The following hash function is a better alternative for hashing strings. Let *s* be a string of length *n*, and let *s*[*i*] denote the *i*th byte of the string. The hash function is defined as:

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdot \cdot \cdot + s[n-1]$$

- The function can be implemented efficiently by setting the hash result initially to 0, and iterating from the first to the last character of the string, at each step multiplying the hash value by 31 and then adding the next character (treated as an integer). The above expression would appear to result in a very large number, but it is actually computed with fixed-size positive integers; the result of each multiplication and addition is thus automatically computed modulo the largest possible integer value plus 1. The result of the above function modulo the number of buckets can then be used for indexing.

- Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file.

Handling of Bucket Overflows:

- So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur. Bucket overflow can occur for several reasons:
- **Insufficient buckets**. The number of buckets, which we denote n_B , must be chosen such that $n_B > n_r / f_r$, where n_r denotes the total number of records that will be stored and f_r denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.
- **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:

  1) Multiple records may have the same search key.

  2) The chosen hash function may result in nonuniform distribution of search keys.

- So that the probability of bucket overflow is reduced, the number of buckets is chosen to be (n_r / f_r) *(1 + *d*), where *d* is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

- Despite allocation of a few more buckets than required, bucket overflow can still occur. We handle bucket overflow by using **overflow buckets**. If a record must be inserted into a bucket *b*, and *b* is already full, the system provides an overflow bucket for *b*, and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list. Overflow handling using such a linked list is called **overflow chaining**.

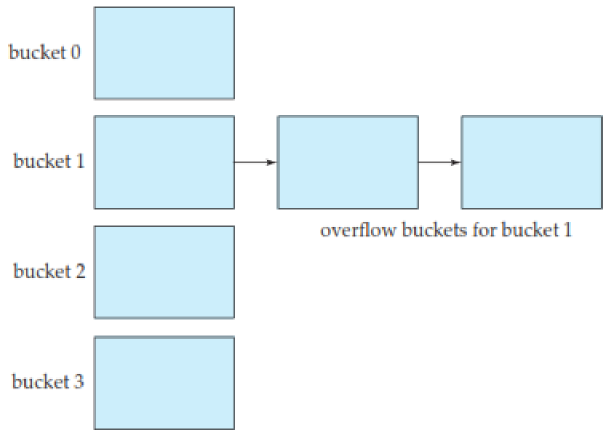


Figure . Overflow chaining in a hash structure.

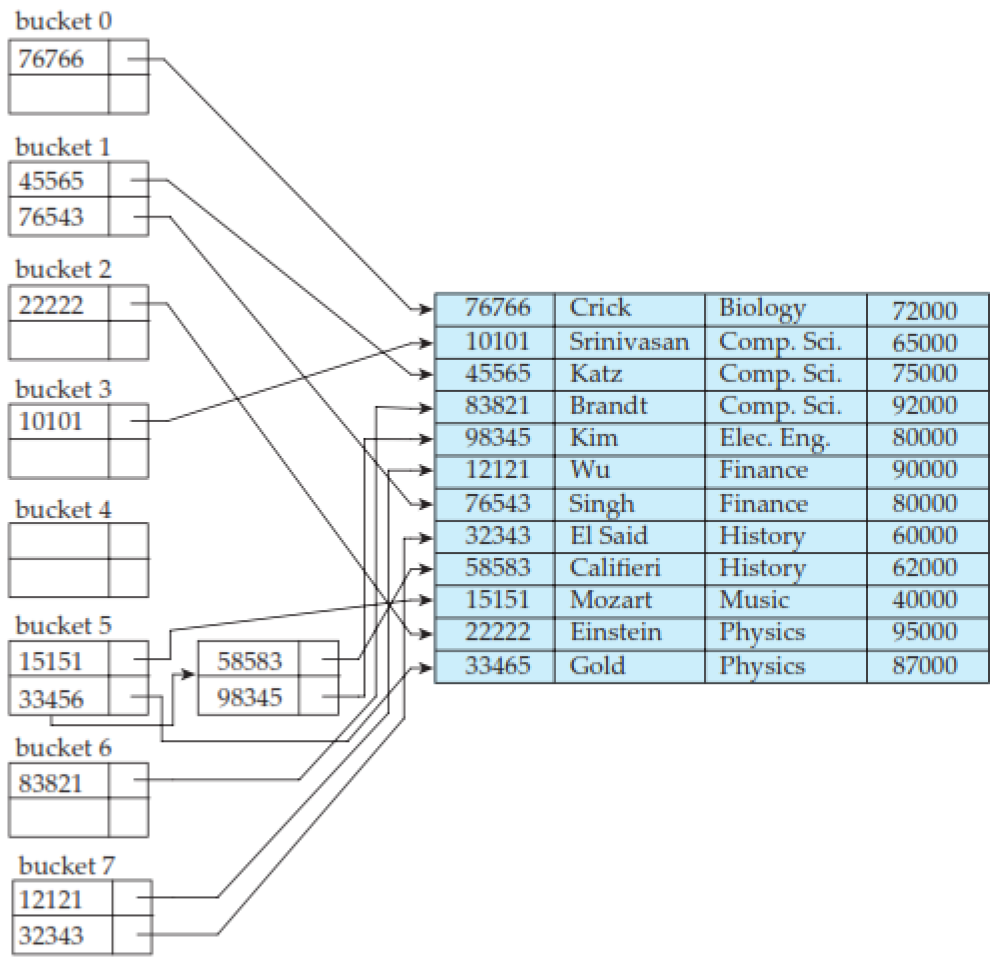
- We must change the lookup algorithm slightly to handle overflow chaining. As before, the system uses the hash function on the search key to identify a bucket *b*. The system must examine all the records in bucket *b* to see whether they match the search key, as before. In addition, if bucket *b* has overflow buckets, the system must examine the records in all the overflow buckets also.

- The form of hash structure that we have just described is sometimes referred to as **closed hashing**.
- Under an alternative approach, called **open hashing**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets *B*. One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*. Other policies, such as computing further hash functions, are also used.
- Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.

- An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function *h* maps search-key values to a fixed set *B* of bucket addresses, we waste space if *B* is made large to handle future growth of the file. If *B* is too small, the buckets contain records of many different search-key values, and bucket overflows can occur. As the file grows, performance suffers.

HASH INDICES:

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). Figure shows a secondary hash index on the *instructor* file, for the search key *ID*. The hash function in the figure computes the sum of the digits of the *ID* modulo 8.
- The hash index has eight buckets, each of size 2 (realistic indices would, of course, have much larger bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *ID* is a primary key for *instructor*, so each search key has only one associated pointer. In general, multiple pointers can be associated with each key.



76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

Figure. Hash index on search key *ID* of *instructor* file.

- We use the term *hash index* to denote hash file structures as well as secondary hash indices. Strictly speaking, hash indices are only secondary index structures. A hash index is never needed as a clustering index structure, since, if a file itself is organized by hashing, there is no need for a separate hash index structure on it. However, since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a clustering hash index on it.

4.9. DYNAMIC HASHING:

- The need to fix the set *B* of bucket addresses presents a serious problem with the static hashing technique. Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:
- Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
- Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
- Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating

new bucket assignments. This reorganization is a massive, time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

- **Dynamic hashing** techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database.

Data Structure:

- Extendable hashing copes with changes in database size by splitting and join together buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since the reorganization is performed on only one bucket at a time, the resulting performance overhead is acceptably low.

- With extendable hashing, we choose a hash function *h* with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range — namely, *b*-bit binary integers. A typical value for *b* is 32.

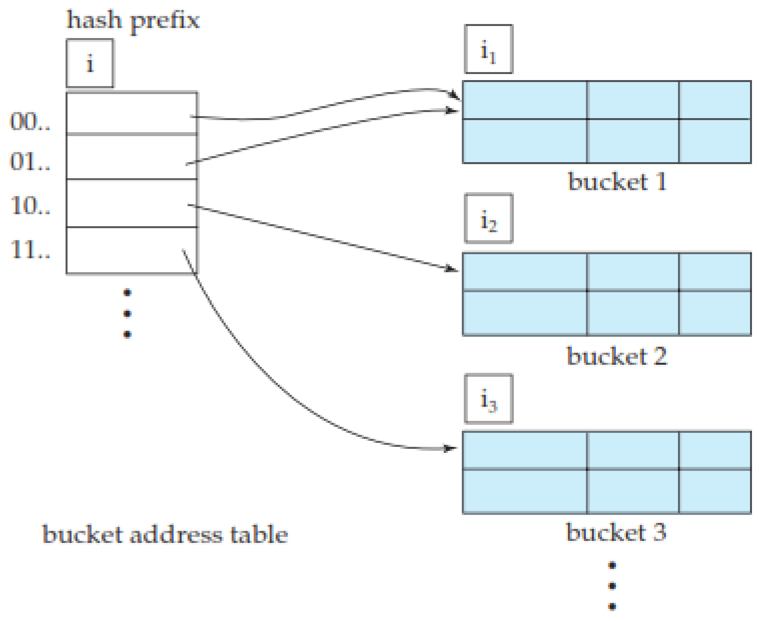


Figure. General extendable hash structure.

- We do not create a bucket for each hash value. Indeed, 2^{32} is over 4 billion, and that many buckets is unreasonable for all but the largest databases. Instead, we create buckets on demand, as records are inserted into the file. We do not use the entire *b* bits of the hash value initially. At any point, we use *i* bits, where
- $0 \leq i \leq b$. These *i* bits are used as an offset into an additional table of bucket addresses. The value of *i* grows and shrinks with the size of the database.
- Figure shows a general extendable hash structure. The *i* appearing above the bucket address table in the figure indicates that *i* bits of the hash value *h*(*K*) are required to determine the correct bucket for *K*. This number will change as the file grows. Although *i* bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket.
- All such entries will have a common hash prefix, but the length of this prefix may be less than *i*. Therefore, we associate with each bucket an integer giving the length of the common hash prefix. In

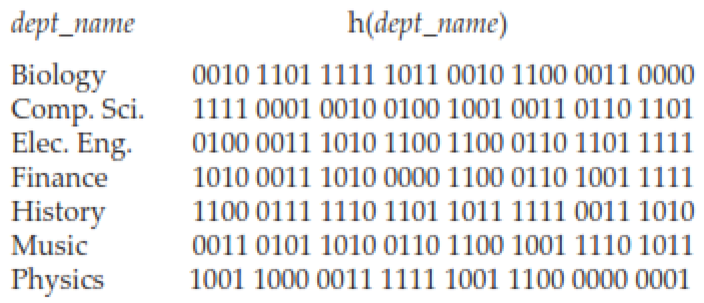
Figure the integer associated with bucket *j* is shown as i_j . The number of bucket-address-table entries that point to bucket *j* is $_2(i - i_j)$

Queries and Updates:

- To locate the bucket containing search-key value K_l , the system takes the first *i* high-order bits of $h(K_l)$, looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.
- To insert a record with search-key value K_l , the system follows the same procedure for lookup as before, ending up in some bucket—say, *j*. If there is room in the bucket, the system inserts the record in the bucket. If, on the other hand, the bucket is full, it must split the bucket and redistribute the current records, plus the new one. To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses.

- If $i = i_j$, only one entry in the bucket address table points to bucket *j*. Therefore, the system needs to increase the size of the bucket address table so that it can include pointers to the two buckets that result from splitting bucket *j*. It does so by considering an additional bit of the hash value. It increments the value of *i* by 1, thus doubling the size of the bucket address table. It replaces each entry by two entries, both of which contain the same pointer as the original entry. Now two entries in the bucket address table point to bucket *j*. The system allocates a new bucket (bucket *z*), and sets the second entry to point to the new bucket. It sets i_j and i_z to *i*. Next, it rehashes each record in bucket *j* and, depending on the first *i* bits (remember the system has added 1 to *i*), either keeps it in bucket *j* or allocates it to the newly created bucket.

- The system now reattempts the insertion of the new record. Usually, the attempt will succeed. However, if all the records in bucket *j*, as well as the new record, have the same hash-value prefix, it will be necessary to split a bucket again, since all the records in bucket *j* and the new record are assigned to the same bucket. If the hash function has been chosen carefully, it is unlikely that a single insertion will require that a bucket be split more than once, unless there are a large number of records with the same search key. If all the records in bucket *j* have the same search-key value, no amount of splitting will help. In such cases, overflow buckets are used to store the records, as in static hashing.
- If $i > i_j$, then more than one entry in the bucket address table points to bucket *j*. Thus, the system can split bucket *j* without increasing the size of the bucket address table. Observe that all the entries that point to bucket *j* correspond to hash prefixes that have the same value on the leftmost i_j bits. The system allocates a new bucket (bucket *z*), and sets i_j and i_z to the value that results from adding 1 to the original i_j value. Next, the system needs to adjust the entries in the bucket address table that previously pointed to bucket *j*. (Note that with the new value for i_j , not all the entries correspond to hash prefixes that have the same value on the leftmost i_j bits.) The system leaves the first half of the entries as they were (pointing to bucket *j*), and sets all the remaining entries to point to the newly created bucket (bucket *z*). Next, as in the previous case, the system rehashes each record in bucket *j*, and allocates it either to bucket *j* or to the newly created bucket *z*.



dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Figure. Hash function for *dept name.*

- The system then reattempts the insert. In the unlikely case that it again fails, it applies one of the two cases, $i = i_j$ or $i > i_j$, as appropriate.
- Note that, in both cases, the system needs to recompute the hash function on only the records in bucket j.
- To delete a record with search-key value K_l, the system follows the same procedure for lookup as before, ending up in some bucket—say, j. It removes both the search key from the bucket and the record from the file. The bucket, too, is removed if it becomes empty. Note that, at this point, several buckets can be coalesced, and the size of the bucket address table can be cut in half. The procedure for deciding on which buckets can be coalesced and how to coalesce buckets is left to you to do as an exercise. The conditions under which the bucket address table can be reduced in size are also left to you as an exercise. Unlike coalescing of buckets, changing the size of the bucket address table is a rather expensive operation if the table is large. Therefore it may be worthwhile to reduce the bucket-address-table size only if the number of buckets reduces greatly.
- To illustrate the operation of insertion, we use the *instructor* file in Figure 11.1 and assume that the search key is *dept name* with the 32-bit hash values as appear in Figure 11.27. Assume that, initially, the file is empty, as in Figure 11.28. We insert the records one by one. To illustrate all the features of extendable hashing in a small structure, we shall make the unrealistic assumption that a bucket can hold only two records.

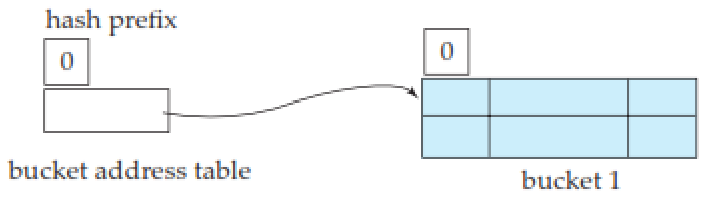
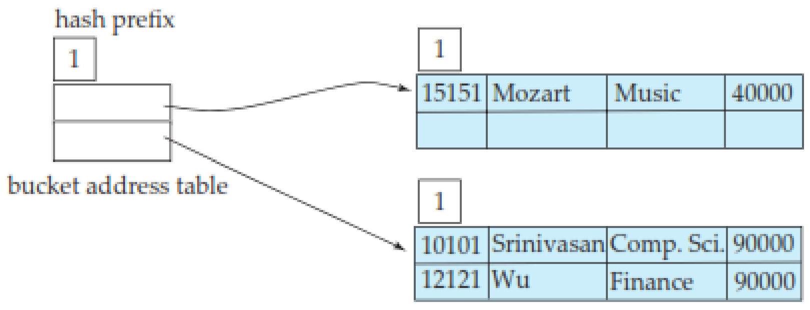


Figure. Initial extendable hash structure.



15151	Mozart	Music	40000

10101	Srinivasan	Comp. Sci.	90000
12121	Wu	Finance	90000

Figure. Hash structure after three insertions.

- We insert the record (10101, Srinivasan, Comp. Sci., 65000). The bucket address table contains a pointer to the one bucket, and the system inserts the record. Next, we insert the record (12121, Wu, Finance, 90000). The system also places this record in the one bucket of our structure.

- When we attempt to insert the next record (15151, Mozart, Music, 40000), we find that the bucket is full. Since $i = i_0$, we need to increase the number of bits that we use from the hash value. We now use 1 bit, allowing us $2^1 = 2$ buckets. This increase in the number of bits necessitates doubling the size of the bucket address table to two entries. The system splits the bucket, placing in the new bucket those records whose search key has a hash value beginning with 1, and leaving in the original bucket the other records. Figure 11.29 shows the state of our structure after the split.

- Next, we insert (22222, Einstein, Physics, 95000). Since the first bit of *h*(Physics) is 1, we must insert this record into the bucket pointed to by the "1" entry in the bucket address table. Once again, we find the bucket full and $i = i_1$. We increase the number of bits that we use from the hash to 2. This increase in the number of bits necessitates doubling the size of the bucket address table to four entries, as in Figure 11.30. Since the bucket of Figure 11.29 for hash prefix 0 was not split, the two entries of the bucket address table of 00 and 01 both point to this bucket.

- For each record in the bucket of Figure 11.29 for hash prefix 1 (the bucket being split), the system examines the first 2 bits of the hash value to determine which bucket of the new structure should hold it.

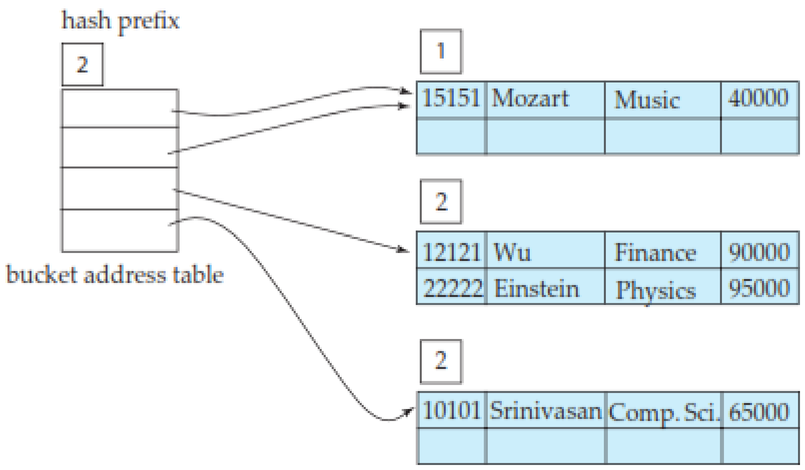


Figure . Hash structure after four insertions.

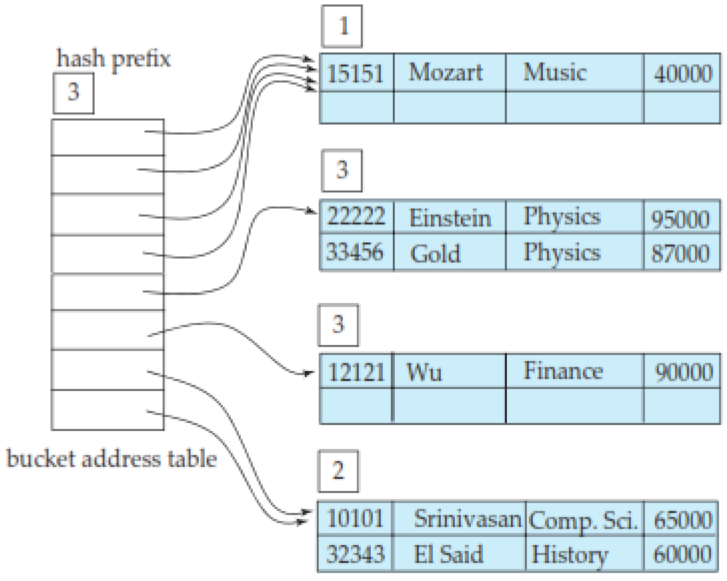
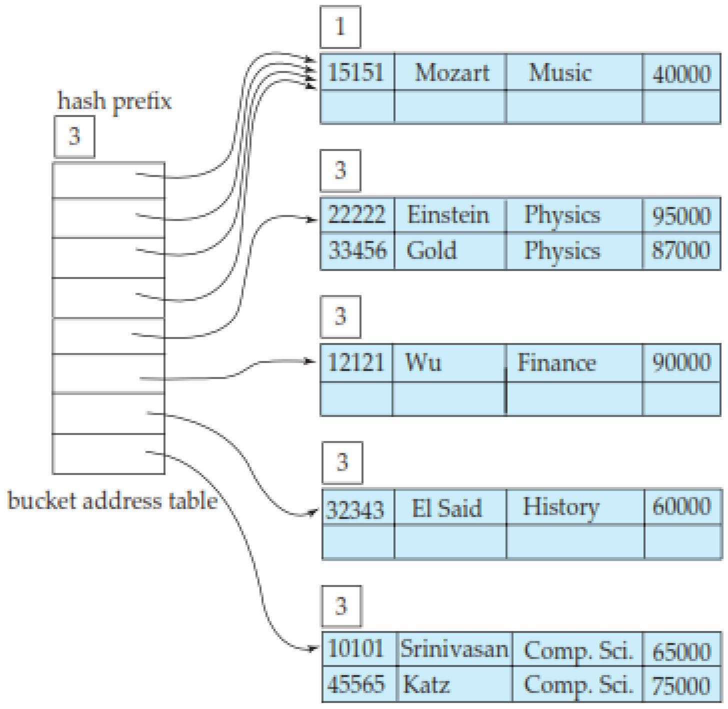


Figure . Hash structure after six insertions.



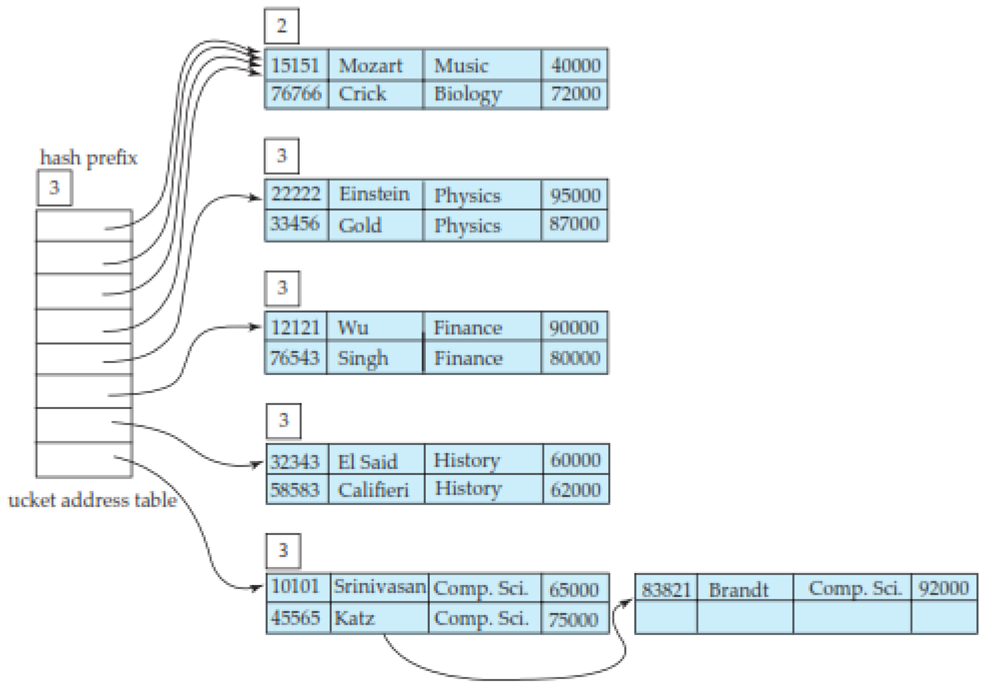
1
15151 Mozart Music 40000
hash prefix
3
3
22222 Einstein Physics 95000
33456 Gold Physics 87000
3
12121 Wu Finance 90000
3
32343 El Said History 60000
bucket address table
3
10101 Srinivasan Comp. Sci. 65000
45565 Katz Comp. Sci. 75000

Figure. Hash structure after seven insertions.

- Next, we insert (32343, El Said, History, 60000), which goes in the same bucket as Comp. Sci. The following insertion of (33456, Gold, Physics, 87000) results in a bucket overflow, leading to an increase in the number of bits, and a doubling of the size of the bucket address table.

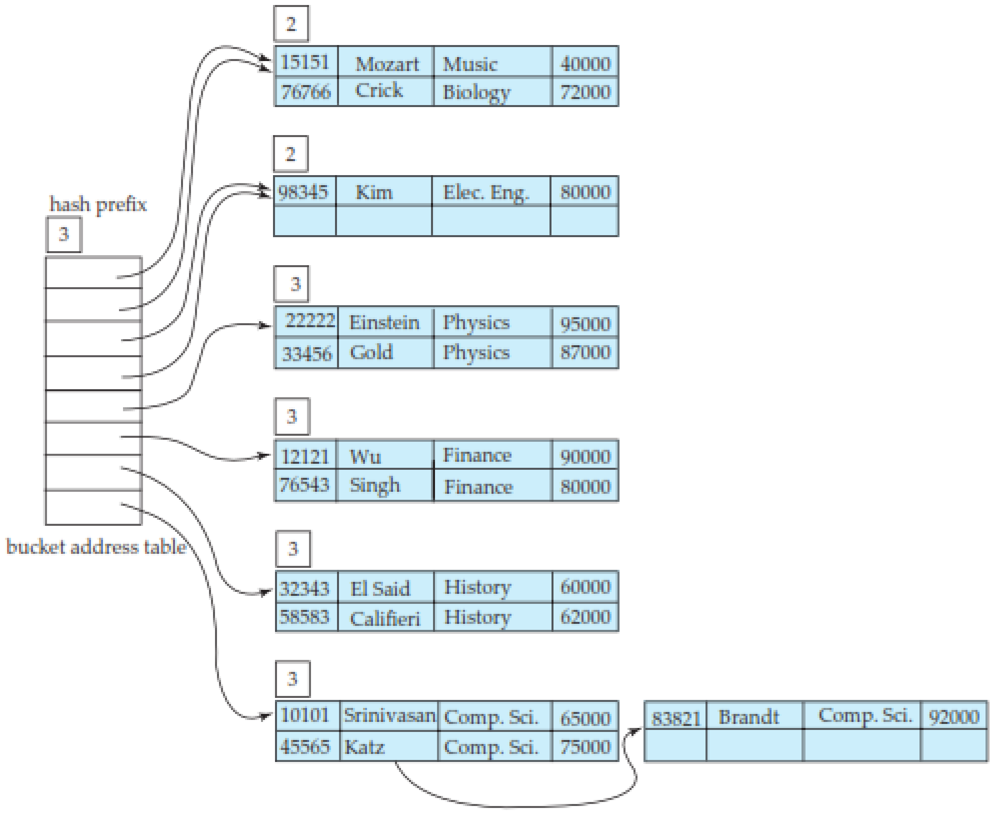
- The insertion of (45565, Katz, Comp. Sci., 75000) leads to another bucket over-flow; this overflow, however, can be handled without increasing the number of bits, since the bucket in question has two pointers pointing to it.

Next, we insert the records of “Califieri”, “Singh”, and “Crick” without any bucket overflow. The insertion of the third Comp. Sci. record (83821, Brandt, Comp. Sci., 92000), however, leads to another overflow. This overflow cannot be handled by increasing the number of bits, since there are three records with exactly the same hash value. Hence the system uses an overflow bucket, as in figure.



hash prefix
3
ucket address table
2
15151 Mozart Music 40000
76766 Crick Biology 72000
3
22222 Einstein Physics 95000
33456 Gold Physics 87000
3
12121 Wu Finance 90000
76543 Singh Finance 80000
3
32343 El Said History 60000
58583 Califieri History 62000
3
10101 Srinivasan Comp. Sci. 65000
45565 Katz Comp. Sci. 75000
83821 Brandt Comp. Sci. 92000

Figure. Hash structure after eleven insertions.



hash prefix
3
bucket address table
2
15151 Mozart Music 40000
76766 Crick Biology 72000
2
98345 Kim Elec. Eng. 80000
3
22222 Einstein Physics 95000
33456 Gold Physics 87000
3
12121 Wu Finance 90000
76543 Singh Finance 80000
3
32343 El Said History 60000
58583 Califieri History 62000
3
10101 Srinivasan Comp. Sci. 65000
45565 Katz Comp. Sci. 75000
83821 Brandt Comp. Sci. 92000

Figure. Extendable hash structure for the *instructor* file.

Static Hashing versus Dynamic Hashing:

- We now examine the advantages and disadvantages of extendable hashing, com-pared with static hashing. The main advantage of extendable hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Although the bucket address table incurs additional overhead, it contains one pointer for each hash value for the current prefix length. This table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

- A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the bucket itself. This extra reference has only a minor effect on per-formance. Although the hash structures that we discussed in Section 11.6 do not have this extra level of indirection, they lose their minor performance advantage as they become full.

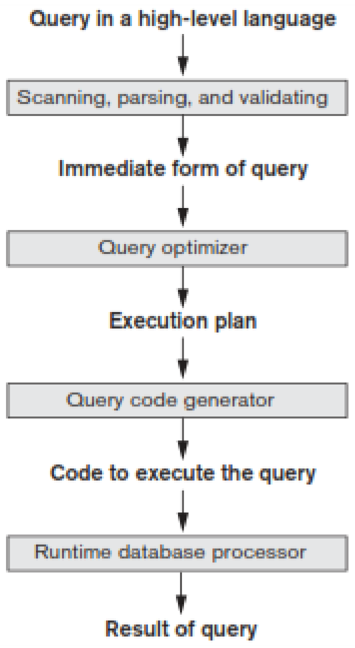
- Thus, extendable hashing appears to be a highly attractive technique, pro-vided that we are willing to accept the added complexity involved in its im-plementation. The bibliographical notes reference more detailed descriptions of extendable hashing implementation.

- The bibliographical notes also provide references to another form of dynamic hashing called **linear hashing**, which avoids the extra level of indirection associ-ated with extendable hashing, at the possible cost of more overflow buckets.

4.10. QUERY PROCESSING OVERVIEW:

- A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.
- The **scanner** identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query, whereas the **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language.
- The query must also be **validated** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried.
- An internal representation of the query is then created, usually as a tree data structure called a **query tree**. It is also possible to represent the query using a graph data structure called a **query graph**.
- The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

- Figure shows the different steps of processing a high-level query. The **query optimizer** module has the task of producing a good execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.


Query in a high-level language
Scanning, parsing, and validating
Immediate form of query
Query optimizer
Execution plan
Query code generator
Code to execute the query
Runtime database processor
Result of query

Code can be:
Executed directly (interpreted mode)
Stored and executed later whenever needed (compiled mode)

Figure . Typical steps when processing a high-level query.

- The term *optimization* is actually a inaccurate name because in some cases the chosen execution plan is not the optimal (or absolute best) strategy—it is just a *reasonably efficient strategy* for executing the query.

Finding the optimal strategy is usually too time-consuming—except for the simplest of queries. In addition, trying to find the optimal query execution strategy may require detailed information on how the files are implemented and even on the contents of the files—information that may not be fully available in the DBMS catalog. Hence, *planning of a good execution strategy* may be a more accurate description than *query optimization.*

- For lower-level navigational database languages in legacy systems—such as the network DML or the hierarchical DL/1—the programmer must choose the query execution strategy while writing a database program.
- If a DBMS provides only a navigational language, there is *limited need or opportunity* for extensive query optimization by the DBMS; instead, the programmer is given the capability to choose the query execution strategy. On the other hand, a high-level query language—such as SQL for relational DBMSs (RDBMSs) or OQL for object DBMSs (ODBMSs)—is more declarative in nature because it specifies what the intended results of the query are, rather than identifying the details of *how* the result should be obtained. Query optimization is thus necessary for queries that are specified in a high-level query language.

- We will concentrate on describing query optimization in the *context of an RDBMS* because many of the techniques we describe have also been adapted for other types of database management systems, such as ODBMSs. A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or near-optimal strategy. Each DBMS typically has a number of general database access algorithms that implement relational algebra operations such as SELECT or JOIN or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query, as well as to the *particular physical database design*, can be considered by the query optimization module.

Translating SQL Queries into Relational Algebra:

- In practice, SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into *query blocks*, which form the basic units that can be translated into the algebraic operators and optimized. A **query block** contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT—these operators must also be included in the extended algebra.

Consider the following SQL query on the EMPLOYEE relation :

SELECT Lname, Fname FROM EMPLOYEE WHERE Salary > (SELECT MAX (Salary) FROM EMPLOYEE WHERE Dno=5);

This query retrieves the names of employees (from any department in the company) who earn a salary that is greater than the *highest salary in department 5*. The query includes a nested subquery and hence would be decomposed into two blocks. The inner block is:

SELECT MAX (Salary) FROM EMPLOYEE WHERE Dno=5)

This retrieves the highest salary in department 5. The outer query block is:

SELECT Lname, Fname FROM EMPLOYEE WHERE Salary > c ;

where c represents the result returned from the inner block. The inner block could be translated into the following extended relational algebra expression:

$$\Im_{\text{MAX Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$$

and the outer block into the expression:

$\pi_{\text{Lname,Fname}}(\sigma_{\text{Salary}>c}(\text{EMPLOYEE}))$

- The *query optimizer* would then choose an execution plan for each query block. Notice that in the above example, the inner block needs to be evaluated only once to produce the maximum salary of employees in department 5, which is then used—as the constant c—by the outer block. We called this a *nested query (without correlation with the outer query)* in Section 5.1.2. It is much harder to optimize the more com-plex *correlated nested queries* (see Section 5.1.3), where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block.

4.11. ALGORITHMS FOR SELECT OPERATIONS

Implementing the SELECT Operation

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions.

We discuss some of the algorithms for implementing SELECT in this section. We will use the following operations, specified on the relational database in Figure 3.5, to illustrate our discussion:

OP1: $\sigma_{\text{Ssn} = \text{'123456789'}}(\text{EMPLOYEE})$
OP2: $\sigma_{\text{Dnumber} > 5}(\text{DEPARTMENT})$
OP3: $\sigma_{\text{Dno} = 5}(\text{EMPLOYEE})$
OP4: $\sigma_{\text{Dno} = 5 \text{ AND Salary} > 30000 \text{ AND Sex} = \text{'F'}}(\text{EMPLOYEE})$
OP5: $\sigma_{\text{Essn}=\text{'123456789' AND Pno} = 10}(\text{WORKS\_ON})$

Search Methods for Simple Selection. A number of search algorithms are possible for selecting records from a file. These are also known as **file scans**, because they scan the records of a file to search for and retrieve records that satisfy a selection condition. If the search algorithm involves the use of an index, the index search is called an **index scan**. The following search methods (S1 through S6) are examples of some of the search algorithms that can be used to implement a select operation:

S1—Linear search (brute force algorithm). Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition. Since the records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

S2—Binary search. If the selection condition involves an equality comparison on a key attribute on which the file is **ordered**, binary search—which is more efficient than linear search—can be used. An example is OP1 if Ssn is the ordering attribute for the EMPLOYEE file.

S3a—Using a primary index. If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn = ‘123456789’ in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).

S3b—Using a hash key. If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = '123456789' in OP1—use the hash key to retrieve the record. Note that this condition retrieves a single record (at most).

S4—Using a primary index to retrieve multiple records. If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, Dnumber > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (Dnumber = 5), then retrieve all subsequent records in the (ordered) file. For the condition Dnumber < 5, retrieve all the preceding records.

S5—Using a clustering index to retrieve multiple records. If the selection condition involves an equality comparison on a **non key attribute** with a clustering index—for example, Dno = 5 in OP3—use the index to retrieve all the records satisfying the condition.

S6—Using a secondary (B^+-tree) index on an equality comparison. This search method can be used to retrieve a single record if the indexing field is a **key** (has unique values) or to retrieve multiple records if the indexing field is **not a key**. This can also be used for comparisons involving >, >=, <, or <=.

Method S1 (**linear search**) applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition.

Method S2 (**binary search**) requires the file to be sorted on the search attribute.

The methods that use an index (S3a, S4, S5, and S6) are generally referred to as **index searches**, and they require the appropriate index to exist on the search attribute. Methods S4 and S6 can be used to retrieve records in a certain *range*—for example, 30000 <= Salary <= 35000. Queries involving such conditions are called **range queries**.

Search Methods for Complex Selection. If a condition of a SELECT operation is a **conjunctive condition**—that is, if it is made up of several simple conditions connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

S7—Conjunctive selection using an individual index. If an attribute involved in any **single simple condition** in the conjunctive select condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record *satisfies the remaining simple conditions* in the conjunctive select condition.

S8—Conjunctive selection using a composite index. If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index (or hash structure) exists on the combined fields— for example, if an index has been created on the composite key (Essn, Pno) of the WORKS_ON file for OP5—we can use the index directly.

S9—Conjunctive selection by intersection of record pointers. If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers gives the record pointers that satisfy the conjunctive select condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions. In general, method S9 assumes that each of the indexes is on a *nonkey field* of the file, because if one of the conditions is an equality condition on a key field, only one record will satisfy the whole condition.

- Whenever a single condition specifies the selection—such as OP1, OP2, or OP3— the DBMS can only check whether or not an access path exists on the attribute involved in that condition. If an access path (such as index or hash key or sorted file) exists, the method corresponding to that access path is used; otherwise, the brute force, linear search approach of method S1 can be used. Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever *more than one* of the attributes involved in the conditions have an access path. The optimizer should choose the access path that *retrieves the fewest records* in the most efficient way by estimating the different costs (see Section 19.8) and choosing the method with the least estimated cost.

- Selectivity of a Condition. When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the
- *selectivity* of each condition. The **selectivity (*sl*)** is defined as the ratio of the num-ber of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus is a number between zero and one. *Zero selec-tivity* means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition. In gen-eral, the selectivity will not be either of these two extremes, but will be a fraction that estimates the percentage of file records that will be retrieved.

- Although exact selectivities of all conditions may not be available, **estimates of selectivities** are often kept in the DBMS catalog and are used by the optimizer. For example, for an equality condition on a key attribute of relation $r(R)$, $s = 1/|r(R)|$, where $|r(R)|$ is the number of tuples in relation $r(R)$. For an equality condition on a nonkey attribute with *i distinct values, s* can be estimated by $(|r(R)|/i)/|r(R)|$ or $1/i$, assuming that the records are evenly or **uniformly distributed** among the distinct values.[8] Under this assumption, $|r(R)|/i$ records will satisfy an equality condition on this attribute. In general, the number of records satisfying a selection condition with selectivity *sl* is estimated to be $|r(R)| * sl$. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records. In certain cases, the actual distribution of records among the various distinct values of the attribute is kept by the DBMS in the form of a *histogram*, in order to get more accurate esti-mates of the number of records that satisfy a particular condition.

- Disjunctive Selection Conditions. Compared to a conjunctive selection condi-tion, a **disjunctive condition** (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process and optimize. For example, consider OP4 :

$$\text{OP4}: \sigma_{\text{Dno=5 OR Salary > 30000 OR Sex='F'}}(\text{EMPLOYEE})$$

- With such a condition, little optimization can be done, because the records satisfy-ing the disjunctive condition are the *union* of the records satisfying the individual conditions. Hence, if any *one* of the conditions does not have an access path, we are compelled to use the brute force, linear search approach. Only if an access path exists on *every* simple condition in the disjunction can we optimize the selection by retrieving the records satisfying each condition—or their record ids—and then applying the *union* operation to eliminate duplicates.

- A DBMS will have available many of the methods discussed above, and typically many additional methods. The query optimizer must choose the appropriate one for executing each SELECT operation in a query. This optimization uses formulas that estimate the costs for each available access method. The optimizer chooses the access method with the lowest estimated cost.

4.12. IMPLEMENTING THE JOIN OPERATION:

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN varieties, so we consider just these two here since we are only giving an overview of query processing and optimization.The term **join** refers to an EQUIJOIN (or NATURAL JOIN).

There are many possible ways to implement a **two-way join**, which is a join on two files. Joins involving more than two files are called **multiway joins**. The number of possible ways to execute multiway joins grows very rapidly. In this section we discuss techniques for implementing *only two-way joins*.

consider the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we discuss next are for a join operation of the form:

$R \bowtie_{A=B} S$

where *A* and *B* are the **join attributes**, which should be domain-compatible attrib-utes of *R* and *S*, respectively. The methods we discuss can be extended to more gen-eral forms of join. We illustrate four of the most common techniques for performing such a join, using the following sample operations:

OP6: EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT
OP7: DEPARTMENT $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE

Methods for Implementing Joins.

J1—Nested-loop join (or nested-block join). This is the default (brute force) algorithm, as it does not require any special access paths on either file in the join. For each record *t* in *R* (outer loop), retrieve every record *s* from *S*

(inner loop) and test whether the two records satisfy the join condition *t*[*A*] = *s*[*B*].[9]

J2—Single-loop join (using an access structure to retrieve the matching records). If an index (or hash key) exists for one of the two join attributes— say, attribute *B* of file *S*—retrieve each record *t* in *R* (loop over file *R*), and then use the access structure (such as an index or a hash key) to retrieve directly all matching records *s* from *S* that satisfy *s*[*B*] = *t*[*A*].

J3—Sort-merge join. If the records of *R* and *S* are *physically sorted* (ordered) by value of the join attributes *A* and *B*, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for *A* and *B*. If the files are not sorted, they may be sorted first by using external sorting (see Section 19.2). In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file—unless both *A* and *B* are nonkey attributes, in which case the method needs to be modified slightly. A sketch of the sort-merge join algorithm is given in Figure 19.3(a). We use *R*(*i*) to refer to the *i*th record in file *R*. A variation of the sort-merge join can be used when secondary indexes exist on both join attributes. The indexes provide the ability to access (scan) the records in order of the join attributes, but the records themselves are physically scattered all over the file blocks, so this method may be quite inefficient, as every record access may involve accessing a different disk block.

J4—Partition-hash join. The records of files *R* and *S* are partitioned into smaller files. The partitioning of each file is done using the same hashing function *h* on the join attribute *A* of *R* (for partitioning file R) and *B* of *S* (for partitioning file S). First, a single pass through the file with fewer records (say, *R*) hashes its records to the various partitions of R; this is called the **partitioning phase**, since the records of *R* are partitioned into the hash buck-ets. In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned, so that the partitioned subfiles of *R* are all kept in main memory. The collection of records with the same value of *h*(*A*) are placed in the same partition, which is a **hash bucket** in a hash table in main memory. In the second phase, called the **probing phase**, a single pass through the other file (*S*) then hashes each of its records using the same hash function *h*(*B*) to *probe* the appropriate bucket, and that record is combined with all matching records from *R* in that bucket. This simplified description of partition-hash join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss the general case of partition-hash join that does not require this assumption below. In practice, techniques J1 to J4 are implemented by accessing *whole disk blocks* of a file, rather than individual records. Depending on the available number of buffers in memory, the number of blocks read in from the file can be adjusted.

- How Buffer Space and Choice of Outer-Loop File Affect Performance of Nested-Loop Join. The buffer space available has an important effect on some of the join algorithms. First, let us consider the nested-loop approach (J1). Looking again at the operation OP6 above, assume that the number of buffers available in main memory for implementing the join is n_B = 7 blocks (buffers). Recall that we assume that each memory buffer is the same size as one disk block.
- For illustration, assume that the DEPARTMENT file consists of r_D = 50 records stored in b_D = 10 disk blocks and that the EMPLOYEE file consists of r_E = 6000 records stored in b_E = 2000 disk blocks. It is advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop (that is, $n_B - 2$ blocks).
- The algorithm can then read one block at a time for the inner-loop file and use its records to **probe** (that is, search) the outer-loop blocks that are currently in main memory for matching records. This reduces the total number of block accesses.
- An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the **result file**—the disk file that will contain the join result—whenever it is filled. This result buffer block then is reused to hold additional join result records.

```
(a) sort the tuples in R on attribute A;        (* assume R has n tuples (records) *)
    sort the tuples in S on attribute B;        (* assume S has m tuples (records) *)
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m)
    do {  if R(i) [A] > S(j) [B]
             then  set j ← j + 1
          elseif R(i) [A] < S(j) [B]
             then  set i ← i + 1
          else {   (* R(i) [A] = S(j) [B], so we output a matched tuple *)
                   output the combined tuple <R(i) , S(j)> to T;

                   (* output other tuples that match R(i), if any *)
                   set l ← j + 1;
                   while (l ≤ m) and (R(i) [A] = S(l) [B])
                   do {  output the combined tuple <R(i) , S(l)> to T;
                         set l ← l + 1
                   }


             (* output other tuples that match S(j), if any *)
             set k ← i + 1;
             while (k ≤ n) and (R(k) [A] = S(j) [B])
             do {  output the combined tuple <R(k) , S(j)> to T;
                   set k ← k + 1
             }
             set i ← k, j ← l
       }
    }
```

Figure. Implementing JOIN by using sort-merge, where *R* has *n* tuples and *S* has *m* tuples. (a) Implementing the operation $T \leftarrow R \bowtie_{A=B} S$.

4.13. USING HEURISTICS IN QUERY OPTIMIZATION

- Optimization techniques apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance.
- The scanner and parser of an SQL query first generate a data structure that corresponds to an *initial query representation*, which is then optimized according to heuristic rules. This leads to an *optimized query representation*, which corresponds to the query execution strategy. Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations, because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

Notation for Query Trees and Query Graphs:

- A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query.

- Figure (a) shows a query tree for query Q2: For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birthdate. This query is specified on the COMPANY relational schema and corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})) \bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

This corresponds to the following SQL query:

```
Q2:      SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate FROM PROJECT AS P,
          DEPARTMENT AS D, EMPLOYEE AS E WHERE    P.Dnum=D.Dnumber AND
         D.Mgr_ssn=E.Ssn AND P.Plocation= 'Stafford';
```

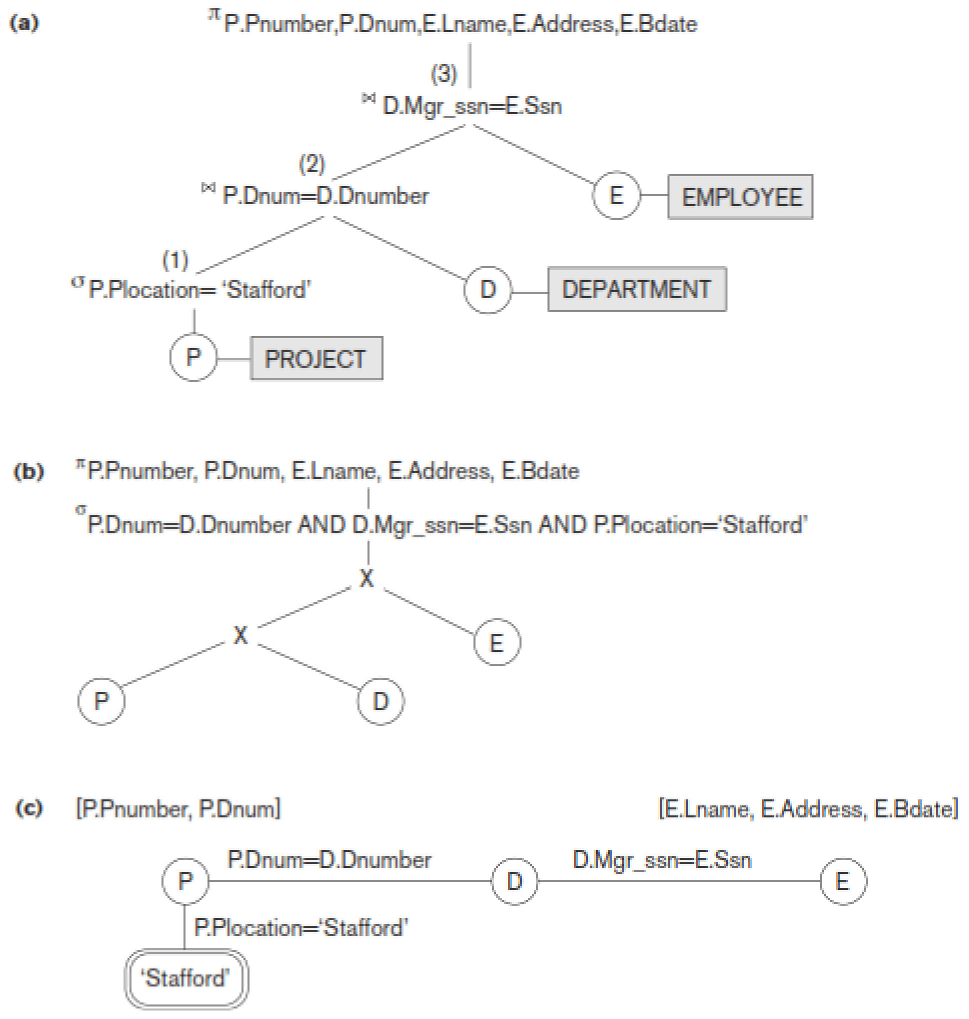

(a)
π P.Pnumber,P.Dnum,E.Lname,E.Address,E.Bdate
(3)
⋈ D.Mgr_ssn=E.Ssn
(2)
⋈ P.Dnum=D.Dnumber
E
EMPLOYEE
(1)
σ P.Plocation= 'Stafford'
D
DEPARTMENT
P
PROJECT
(b)
π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D
(c)
[P.Pnumber, P.Dnum]
[E.Lname, E.Address, E.Bdate]
P.Dnum=D.Dnumber
D.Mgr_ssn=E.Ssn
P
D
E
P.Plocation='Stafford'
'Stafford'

Figure. Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

- In Figure(a), the leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE, respectively, and the internal tree nodes represent the *relational algebra operations* of the expression. When this query tree is executed, the node marked (1) in Figure (a) must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

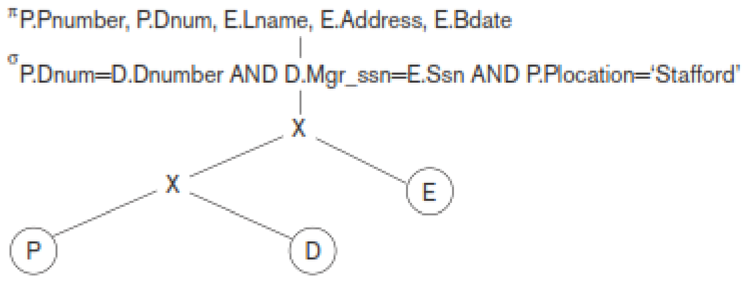
- As we can see, the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the **query graph** notation. Figure (c) shows the query graph for query Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure (c) . Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

- The query graph representation does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query. Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

HEURISTIC OPTIMIZATION OF QUERY TREES:

- In general, many different relational algebra expressions can be **equivalent**; that is, they can represent the *same query*.

- The query parser will typically generate a standard **initial query tree** to correspond to an SQL query, without doing any optimization. For example, for a SELECT-PROJECT-JOIN query, such as Q2, the initial tree is shown in Figure (b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes.

(b)

π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D

- Such a canonical query tree represents a relational algebra expression that is *very inefficient if executed directly*, because of the CARTESIAN PRODUCT (×) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each.
- However, the initial query tree in Figure(b) is in a simple standard form that can be easily created from the SQL query. It will never be executed. The heuristic query optimizer will transform this initial query tree into an equivalent **final query tree** that is efficient to execute.

- The optimizer must include rules for *equivalence among relational algebra expressions* that can be applied to transform the initial tree into the final, optimized query tree. First we discuss informally how a query tree is transformed by using heuristics, and then we discuss general transformation rules and show how they can be used in an algebraic heuristic optimizer.

- Example of Transforming a Query. Consider the following query Q: *Find the last names of employees born after 1957 who work on a project named 'Aquarius'*. This query can be specified in SQL as follows:

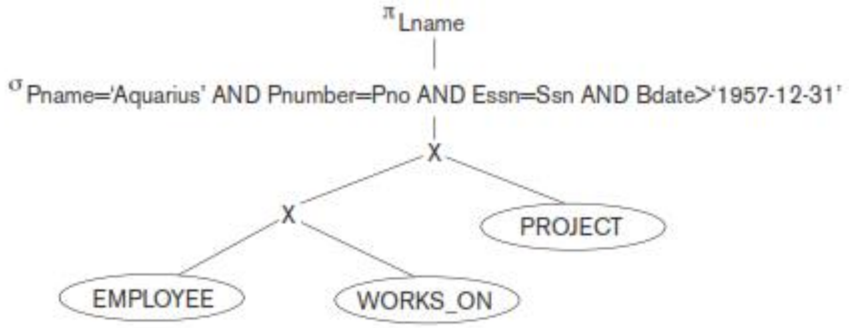
  SELECT Lname FROM EMPLOYEE, WORKS_ON, PROJECT WHERE Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate > '1957-12-31';

- The initial query tree for Q is shown in Figure(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to execute. This particular query needs only one record from the PROJECT relation— for the 'Aquarius' project—and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'. Figure (b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

Figure Steps in converting a query tree during heuristic optimization.

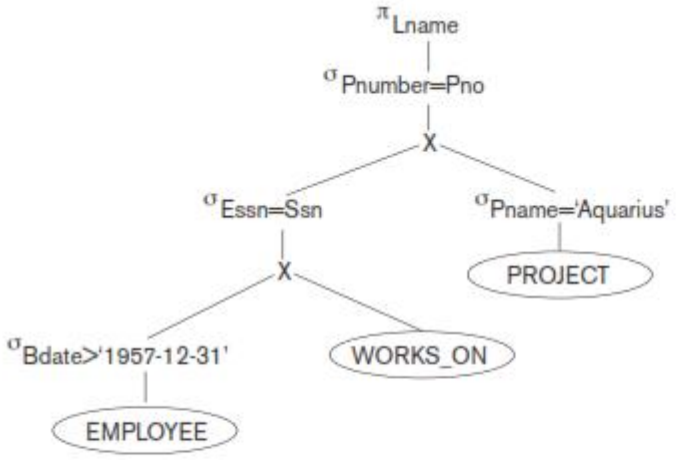
a) Initial (canonical) query tree for SQL query Q.
b) Moving SELECT operations down the query tree.
c) Applying the more restrictive SELECT operation first.
d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
e) Moving PROJECT operations down the query tree.

(a)

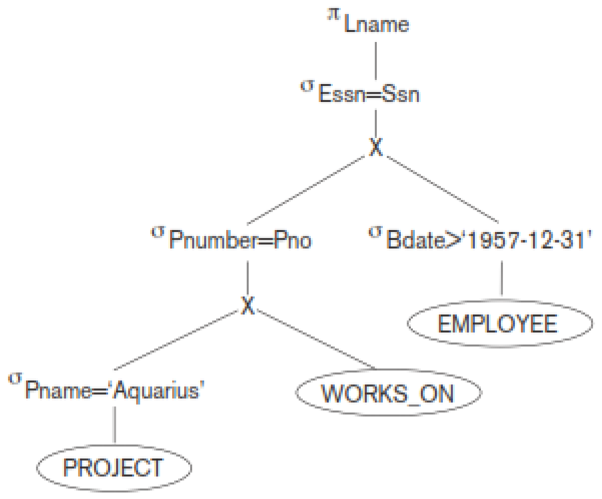


π Lname
σ Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'
X
X
PROJECT
EMPLOYEE
WORKS_ON

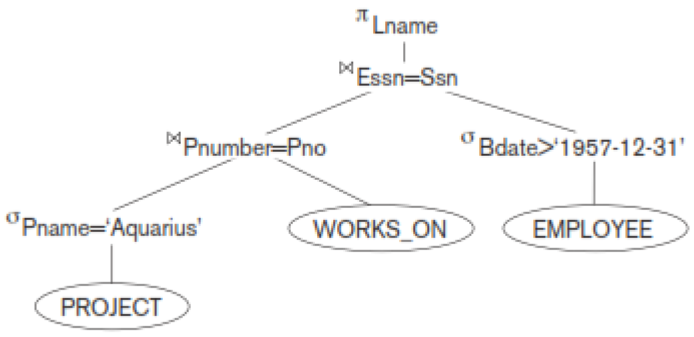
(b)



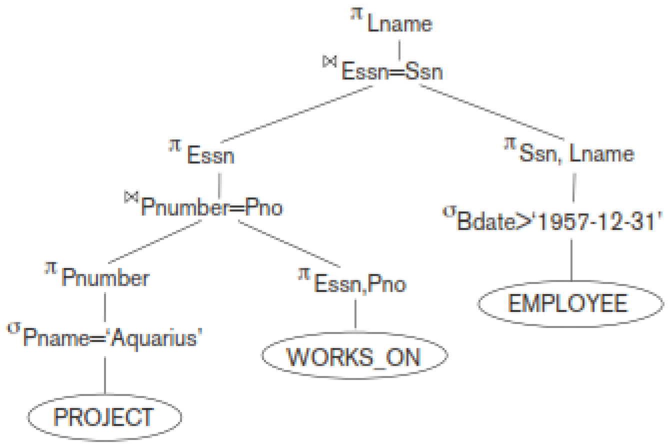
(c)



(d)



(e)



- A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure (c). This uses the information that Pnumber is a key attribute of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only. We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation, as shown in Figure (d). Another improvement is to keep only the attributes needed by subsequent opera-tions in the intermediate relations, by including PROJECT (π) operations as early as possible in the query tree, as shown in Figure(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

- As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules *preserve this equivalence*. We discuss some of these transformation rules next.

- General Transformation Rules for Relational Algebra Operations. There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a *different order* but the two relations represent the same information, we consider the relations to be equivalent.

1. **Cascade of σ** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

   $\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$
2. **Commutativity of σ.** The σ operation is commutative:

   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$
3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:

   $\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$
4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:

   $\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$

5. **Commutativity of ⋈ (and ×).** The join operation is commutative, as is the × operation:

   $R \bowtie_c S \equiv S \bowtie_c R$

   $R \times S \equiv S \times R$

   Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.
6. **Commuting σ with ⋈ (or ×).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

   $\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$

   Alternatively, if the selection condition c can be written as (c_1 AND c_2), where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

   $\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$

   The same rules apply if the ⋈ is replaced by a × operation.

7. **Commuting π with ⋈ (or ×).** Suppose that the projection list is $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

   $\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \ldots, A_n}(R)) \bowtie_c (\pi_{B_1, \ldots, B_m}(S))$

   If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

   $\pi_L(R \bowtie_c S) \equiv \pi_L((\pi_{A_1, \ldots, A_n, A_{n+1}, \ldots, A_{n+k}}(R)) \bowtie_c (\pi_{B_1, \ldots, B_m, B_{m+1}, \ldots, B_{m+p}}(S)))$

   For ×, there is no condition c, so the first transformation rule always applies by replacing $\bowtie_c$ with ×.
8. **Commutativity of set operations.** The set operations ∪ and ∩ are commutative but − is not.

9. **Associativity of $\bowtie$, $\times$, $\cup$, and $\cap$.** These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have:

$(R\ \theta\ S)\ \theta\ T \equiv R\ \theta\ (S\ \theta\ T)$

10. **Commuting σ with set operations.** The σ operation commutes with $\cup$, $\cap$, and $-$. If θ stands for any one of these three operations (throughout the expression), we have:

$\sigma_c\ (R\ \theta\ S) \equiv (\sigma_c\ (R))\ \theta\ (\sigma_c\ (S))$

11. **The π operation commutes with $\cup$.**

$\pi_L\ (R \cup S) \equiv (\pi_L\ (R)) \cup (\pi_L\ (S))$

12. **Converting a $(\sigma, \times)$ sequence into $\bowtie$.** If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert the $(\sigma, \times)$ sequence into a $\bowtie$ as follows:

$(\sigma_c\ (R \times S)) \equiv (R \bowtie_c S)$

There are other possible transformations. For example, a selection or join condition *c* can be converted into an equivalent condition by using the following standard rules from Boolean algebra (DeMorgan's laws):

NOT (c_1 AND c_2) ≡ (NOT c_1) OR (NOT c_2)
NOT (c_1 OR c_2) ≡ (NOT c_1) AND (NOT c_2)

Additional transformations discussed in Chapters 4, 5, and 6 are not repeated here.
We discuss next how transformations can be used in heuristic optimization.

Outline of a Heuristic Algebraic Optimization Algorithm. We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example in Figure. The steps of the algorithm are as follows:

Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.

Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condi-tion involves attributes from *only one table*, which means that it represents a *selection condition*, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from *two tables*, which means that it represents a *join condition*, the condition is moved to a location down the tree after the two tables are combined.

Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of *most restrictive* SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.[17] Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.[18]

Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.

Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

Identify subtrees that represent groups of operations that can be executed by a single algorithm.

In our example, Figure (b) shows the tree in Figure(a) after applying steps 1 and 2 of the algorithm; Figure(c) shows the tree after step 3; Figure 19.5(d) after step 4; and Figure(e) after step 5. In step 6 we may group together the operations in the subtree whose root is the operation π_{Essn} into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation π_{Essn}, because the first grouping means that this subtree is executed first.

Summary of Heuristics for Algebraic Optimization.

The main heuristic is to apply first the operations that reduce the size of intermediate results. This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible. Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

Converting Query Trees into Query Execution Plans

An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree. As a simple example, consider query Q1 from Chapter 4, whose corresponding relational algebra expression is

$\pi_{\text{Fname, Lname, Address}}(\sigma_{\text{Dname='Research'}}(\text{DEPARTMENT}) \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$

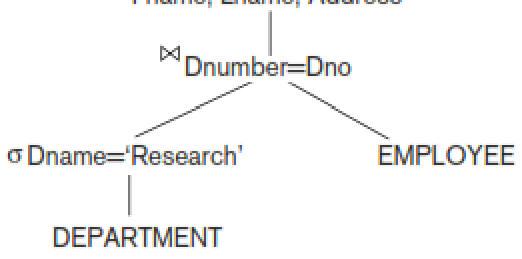


Figure. A query tree for query Q1.

The query tree is shown in Figure 19.6. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), a single-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE), and a scan of the JOIN result for input to the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible.

With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is *physically materialized*). For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table. On the other hand, with **pipelined evaluation**, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence. For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

4.14. USING SELECTIVITY AND COST ESTIMATES IN QUERY OPTIMIZATION:

A query optimizer does not depend solely on heuristic rules; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the *lowest cost estimate*. For this approach to work, accurate *cost estimates* are required so that different strategies can be compared fairly and realistically. In addition, the optimizer must limit the number of execution strategies to be considered; otherwise, too much time will be spent making cost estimates for the many possible execution strategies. Hence, this approach is more suitable for **compiled queries** where the optimization is done at compile time and the resulting execution strategy code is stored and executed directly at runtime. For **interpreted queries**, where the entire process shown in
A full-scale optimization may slow down the response time. A more elaborate optimization is indicated for compiled queries, whereas a partial, less time-consuming optimization works best for interpreted queries.

This approach is generally referred to as **cost-based query optimization**. It uses traditional optimization techniques that search the *solution space* to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal (absolute best) one.

Cost Components for Query Execution:

The cost of executing a query includes the following components:

1) **Access cost to secondary storage.** This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as *disk I/O (input/output) cost*. The cost of search-ing for records in a disk file depends on the type of access structures on that file,

such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

2) **Disk storage cost.** This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.

3) **Computation cost.** This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as *CPU (central processing unit) cost*.

4) **Memory usage cost.** This is the cost pertaining to the number of main memory buffers needed during query execution.

5) **Communication cost.** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases, it would also include the cost of transferring tables and results among various computers during query evaluation.

For large databases, the main emphasis is often on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory buffers. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved, communication cost must be minimized also. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components. That is why some cost functions consider a single factor only—disk access.

Catalog Information Used in Cost Functions:

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions. This information may be stored in the DBMS catalog, where it is accessed by the query optimizer. First, we must know the size of each file. For a file whose records are all of the same type, the **number of records (tuples) (*r*)**, the (average) **record size (*R*)**, and the **number of file blocks (*b*)** (or close estimates of them) are needed. The **blocking factor (*bfr*)** for the file may also be needed. We must also keep track of the *primary file organization* for each file.
The primary file organization records may be *unordered*, *ordered* by an attribute with or without a primary or clustering index, or *hashed* (static hashing or one of the dynamic hashing methods) on a key attribute. Information is also kept on all primary, secondary, or clustering indexes and their indexing attributes. The **number of levels (*x*)** of each multilevel index (primary, secondary, or clustering) is needed for cost functions that estimate the number of block accesses that occur during query execution. In some cost functions the **number of first-level index blocks (b_{I1})** is needed.

Another important parameter is the **number of distinct values (*d*)** of an attribute and the attribute **selectivity (*sl*)**, which is the fraction of records satisfying an equal-ity condition on the attribute. This allows estimation of the **selection cardinality ($s = sl * r$)** of an attribute, which is the *average* number of records that will satisfy an equality selection condition on that attribute. For a *key attribute*, $d = r$, $sl = 1/r$ and s=1. For a *nonkey attribute*, by making an assumption that the *d* distinct values are uniformly distributed among the records, we estimate $sl = (1/d)$ and so $s = (r/d)$.

Information such as the number of index levels is easy to maintain because it does not change very often. However, other information may change frequently; for example, the number of records *r* in a file changes every time a record is inserted or deleted. The query optimizer will need reasonably close but not necessarily completely up-to-the-minute values of these parameters for use in estimating the cost of various execution strategies.

For a nonkey attribute with *d* distinct values, it is often the case that the records are not uniformly distributed among these values. For example, suppose that a company has 5 departments numbered 1 through 5, and 200 employees who are distributed among the departments as follows: (1, 5), (2, 25), (3, 70), (4, 40), (5, 60). In such cases, the optimizer can store a **histogram** that reflects the distribution of employee records over different departments in a table with the two attributes (Dno, Selectivity), which would contain the following values for our example: (1, 0.025), (2, 0.125), (3, 0.35), (4, 0.2), (5, 0.3). The selectivity values stored in the histogram can also be estimates if the employee table changes frequently.

In the next two sections we examine how some of these parameters are used in cost functions for a cost-based query optimizer.

Examples of Cost Functions for SELECT:

We now give cost functions for the selection algorithms S1 to S8 in terms of *number of block transfers* between memory and disk. Algorithm S9 involves an intersection of record pointers after they have been retrieved by some other means, such as algorithm S6, and so the cost function will be based on the cost for S6. These cost functions are estimates that ignore computation time, storage cost, and other factors. The cost for method S*i* is referred to as C_{Si} block accesses.

S1—Linear search (brute force) approach. We search all the file blocks to retrieve all records satisfying the selection condition; hence, $C_{S1a} = b$. For an *equality condition on a key attribute*, only half the file blocks are searched *on the average* before finding the record, so a rough estimate for $C_{S1b} = (b/2)$ if the record is found; if no record is found that satisfies the condition, $C_{S1b} = b$.
S2—Binary search. This search accesses approximately $C_{S2} = \log_2 b + (s/bfr) - 1$ file blocks. This reduces to $\log_2 b$ if the equality condition is on a unique (key) attribute, because $s = 1$ in this case.

S3a—Using a primary index to retrieve a single record. For a primary index, retrieve one disk block at each index level, plus one disk block from the data file. Hence, the cost is one more disk block than the number of index levels: $C_{S3a} = x + 1$.

S3b—Using a hash key to retrieve a single record. For hashing, only one disk block needs to be accessed in most cases. The cost function is approximately $C_{S3b} = 1$ for static hashing or linear hashing, and it is 2 disk block accesses for extendible hashing.

S4—Using an ordering index to retrieve multiple records. If the comparison condition is >, >=, <, or <= on a key field with an ordering index, roughly half the file records will satisfy the condition. This gives a cost function of $C_{S4} = x + (b/2)$. This is a very rough estimate, and although it may be correct on the average, it may be quite inaccurate in individual cases. A more accurate estimate is possible if the distribution of records is stored in a histogram.

S5—Using a clustering index to retrieve multiple records. One disk block is accessed at each index level, which gives the address of the first file disk block in the cluster. Given an equality condition on the indexing attribute, *s* records will satisfy the condition, where *s* is the selection cardinality of the indexing attribute. This means that (s/bfr) file blocks will be in the cluster of file blocks that hold all the selected records, giving $C_{S5} = x + (s/bfr)$.

S6—Using a secondary (B^+-tree) index. For a secondary index on a key (unique) attribute, the cost is $x + 1$ disk block accesses. For a secondary index on a nonkey (nonunique) attribute, *s* records will satisfy an equality condition, where *s* is the selection cardinality of the indexing attribute. However, because the index is nonclustering, each of the records may reside on a different disk block, so the (worst case) cost estimate is $C_{S6a} = x + 1 + s$. The additional *1* is to account for the disk block that contains the record pointers after the index is searched. If the comparison condition is >, >=, <, or <= and half the file records are assumed to satisfy the condition, then (very roughly) half the first-level index blocks are accessed, plus half the file records via the index. The cost estimate for this case, approximately, is $C_{S6b} = x + (b_{I1}/2) + (r/2)$. The $r/2$ factor can be refined if better selectivity estimates are available through a histogram. The latter method C_{S6b} can be very costly.

S7—Conjunctive selection. We can use either S1 or one of the methods S2 to S6 discussed above. In the latter case, we use one condition to retrieve the records and then check in the main memory buffers whether each retrieved record satisfies the remaining conditions in the conjunction. If multiple indexes exist, the search of each index can produce a set of record pointers (record ids) in the main memory buffers. The intersection of the sets of record pointers (referred to in S9) can be computed in main memory, and then the resulting records are retrieved based on their record ids.

S8—Conjunctive selection using a composite index. Same as S3*a*, S5, or S6*a*, depending on the type of index.

Examples of Cost Functions for JOIN:

To develop reasonably accurate cost functions for JOIN operations, we need to have an estimate for the size (number of tuples) of the file that results *after* the JOIN operation. This is usually kept as a ratio of the size (number of tuples) of the resulting join file to the size of the CARTESIAN PRODUCT file, if both are applied to the same input files, and it is called the **join selectivity (*js*)**. If we denote the number of tuples of a relation R by $|R|$, we have:

$js = |(R \bowtie_c S)| / |(R \times S)| = |(R \bowtie_c S)| / (|R| * |S|)$

If there is no join condition c, then $js = 1$ and the join is the same as the CARTESIAN PRODUCT. If no tuples from the relations satisfy the join condition, then $js = 0$. In general, $0 <= js <= 1$. For a join where the condition c is an equality comparison $R.A = S.B$, we get the following two special cases:

1) If A is a key of R, then $|(R \bowtie_c S)| \leq |S|$, so $js \leq (1/|R|)$. This is because each record in file S will be joined with at most one record in file R, since A is a key of R. A special case of this condition is when attribute B is a *foreign key* of S that references the *primary key* A of R. In addition, if the foreign key B has the NOT NULL constraint, then $js = (1/|R|)$, and the result file of the join will contain $|S|$ records.

2) If B is a key of S, then $|(R \bowtie_c S)| \leq |R|$, so $js \leq (1/|S|)$.

Having an estimate of the join selectivity for commonly occurring join conditions enables the query optimizer to estimate the size of the resulting file after the join operation, given the sizes of the two input files, by using the formula $|(R \bowtie_c S)| = js$

$|R| * |S|$. We can now give some sample *approximate* cost functions for estimating the cost of some of the join algorithms. The join operations are of the form:

$R \bowtie_{A=B} S$

where A and B are domain-compatible attributes of R and S, respectively. Assume that R has b_R blocks and that S has b_S blocks:

J1—Nested-loop join. Suppose that we use R for the outer loop; then we get the following cost function to estimate the number of block accesses for this method, assuming *three memory buffers*. We assume that the blocking factor for the resulting file is bfr_{RS} and that the join selectivity is known:

$C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$

The last part of the formula is the cost of writing the resulting file to disk.

This cost formula can be modified to take into account different numbers of memory buffers. If n_B main memory buffers are available to perform the join, the cost formula becomes:

$C_{J1} = b_R + (b_R/(n_B - 2) * b_S) + ((js * |R| * |S|)/bfr_{RS})$

J2—Single-loop join (using an access structure to retrieve the matching record(s)). If an index exists for the join attribute *B* of *S* with index levels x_B, we can retrieve each record *s* in *R* and then use the index to retrieve all the matching records *t* from *S* that satisfy $t[B] = s[A]$. The cost depends on the type of index. For a secondary index where s_B is the selection cardinality for the join attribute *B* of *S*,[21] we get:

$C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((js * |R| * |S|)/bfr_{RS})$

For a clustering index where s_B is the selection cardinality of *B*, we get
$C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$

For a primary index, we get

$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$

If a hash key exists for one of the two join attributes—say, *B* of *S*—we get
$C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS})$

where $h \geq 1$ is the average number of block accesses to retrieve a record, given its hash key value. Usually, *h* is estimated to be *1* for static and linear hashing and *2* for extendible hashing.

J3—Sort-merge join. If the files are already sorted on the join attributes, the cost function for this method is
$C_{J3a} = b_R + b_S + ((js * |R| * |S|)/bfr_{RS})$
If we must sort the files, the cost of sorting must be added.

UNIT V ADVANCED TOPICS

Distributed Databases: Architecture, Data Storage, Transaction Processing – Object-based Databases: Object Database Concepts, Object-Relational features, ODMG Object Model, ODL, OQL - XML Databases: XML Hierarchical Model, DTD, XML Schema, XQuery – Information Retrieval: IR Concepts, Retrieval Models, Queries in IR systems.

5.1. DISTRIBUTED DATABASE:

- **Distributed database (DDB)** is a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed data-base management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.

Parallel versus Distributed Architectures:

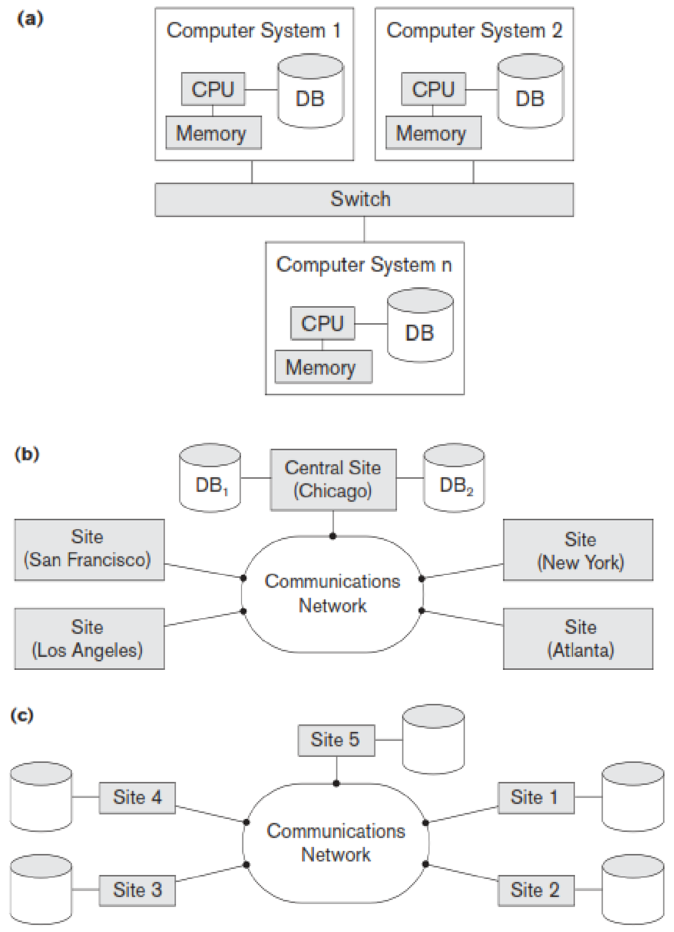
There are two main types of multiprocessor system architectures that are common-place:

1) Shared memory (tightly coupled) architecture. Multiple processors share secondary (disk) storage and also share primary memory.

2) Shared disk (loosely coupled) architecture. Multiple processors share sec-ondary (disk) storage but each has their own primary memory.

- These architectures enable processors to communicate without the overhead of exchanging messages over a network. Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared nothing architecture**.
- In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment where heterogeneity of hardware and operating system at each node is very common. Shared nothing architecture is also considered as an environment for parallel databases.

Figure. Some different database system architectures.
(a) Shared nothing architecture.
(b) A networked architecture with a centralized database at one of the sites.
(c) A truly distributed database architecture.



5.2. GENERAL ARCHITECTURE OF PURE DISTRIBUTED DATABASES

- In Figure, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency.
- To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency.

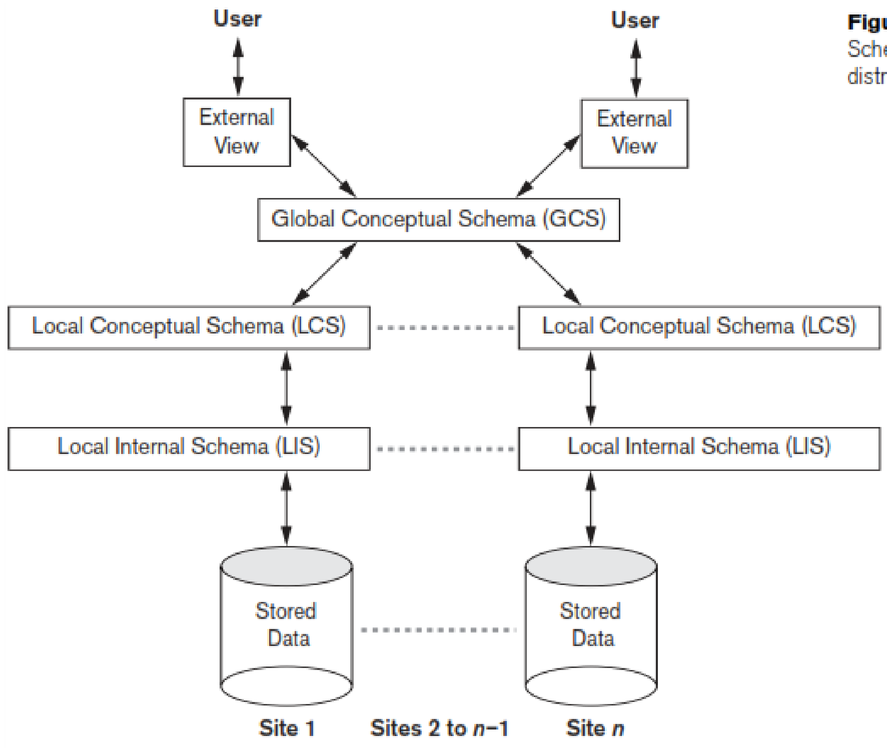

User
User
External View
External View
Global Conceptual Schema (GCS)
Local Conceptual Schema (LCS)
Local Conceptual Schema (LCS)
Local Internal Schema (LIS)
Local Internal Schema (LIS)
Stored Data
Stored Data
Site 1
Sites 2 to n−1
Site n

Figure 25.4
Schema architecture of distributed databases.

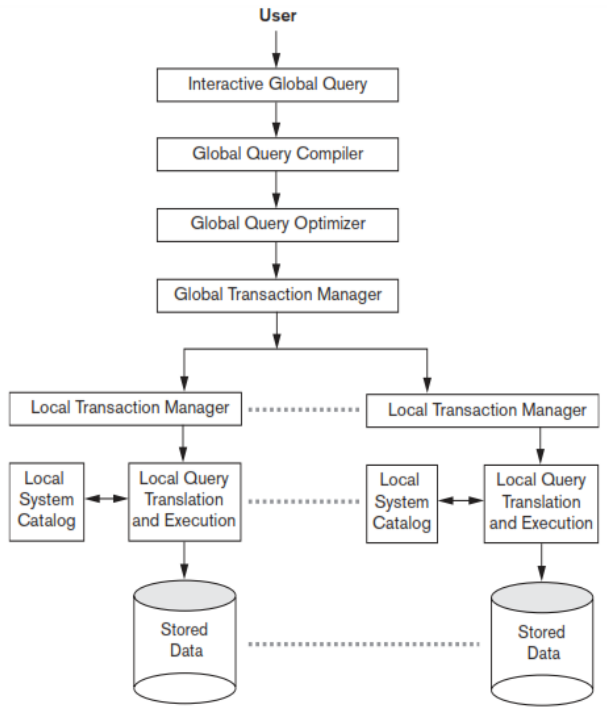

User
Interactive Global Query
Global Query Compiler
Global Query Optimizer
Global Transaction Manager
Local Transaction Manager
Local Transaction Manager
Local System Catalog
Local Query Translation and Execution
Local System Catalog
Local Query Translation and Execution
Stored Data
Stored Data

Figure. Data Component architecture of distributed databases.

- Figure shows the component architecture of a DDB. It is an extension of its centralized counterpart. For the sake of simplicity, common elements are not shown here. The global query compiler references the global conceptual schema from the global system catalog to verify and impose defined constraints.

- The global query optimizer references both global and local conceptual schemas and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU, I/O, and network latencies) and estimated sizes of intermediate results. The latter is particularly important in queries involving joins. Having computed the cost for each candidate, the optimizer selects the candidate with the minimum cost for execution.
- Each local DBMS would have their local query optimizer, transaction man-ager, and execution engines as well as the local system catalog, which houses the local schemas. The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.

Federated Database Schema Architecture:

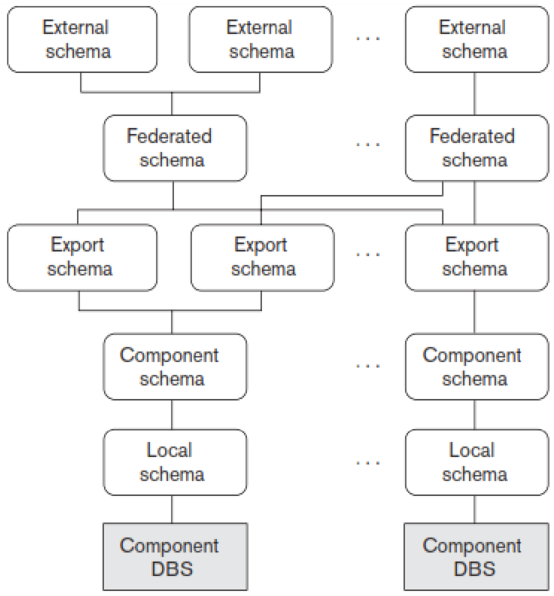


Figure . The five-level schema architecture in a federated database system (FDBS).

Typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure.

1) The **local schema** is the conceptual schema (full database definition) of a component database
2) The **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema.
3) The **export schema** represents the subset of a component schema that is available to the FDBS.
4) The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas.
5) The **external schemas** define the schema for a user group or an application.

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations.

An Overview of Three-Tier Client-Server Architecture:

In the three-tier client-server architecture, the following three layers exist:

Presentation layer (client). This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.

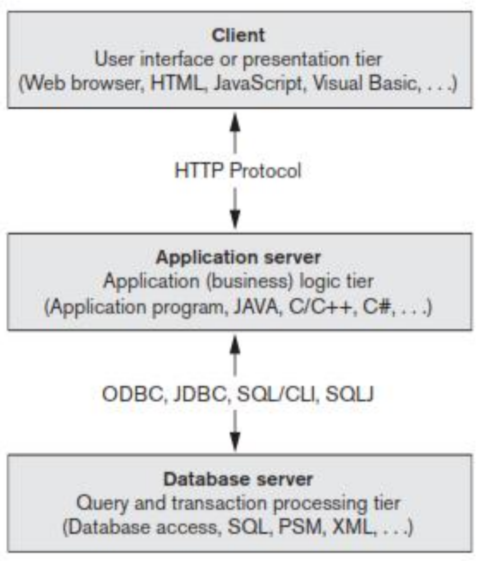

Client
User interface or presentation tier
(Web browser, HTML, JavaScript, Visual Basic, . . .)
HTTP Protocol
Application server
Application (business) logic tier
(Application program, JAVA, C/C++, C#, . . .)
ODBC, JDBC, SQL/CLI, SQLJ
Database server
Query and transaction processing tier
(Database access, SQL, PSM, XML, . . .)

Figure . The three-tier client-server architecture.

Application layer (business logic). This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

Database server. This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML when transmitted between the application server and the database server.

- Exactly how to divide the DBMS functionality between the client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, where an **SQL server** is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, and SQL/CLI.

- In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between an application server and database server might proceed as follows during the processing of an SQL query:

- The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
- Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange, so the database server may format the query result into XML before sending it to the application server.
- The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.
- The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.
- If the DDBMS has the capability to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify the sites at which the data referenced in the query or transaction resides. This property is called **distribution transparency**. Some DDBMSs do not provide distribution transparency, instead requiring that applications are aware of the details of data distribution.

5.3. DATA STORAGE:

- The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.

1. Data Fragmentation:

In a DDB, decisions must be made regarding which site should be used to store which portions of the database.

The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS_ON, and DEPENDENT. In many cases, however, a relation can be divided into smaller logical units for distribution.
We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* can be used to partition each relation by department.

a) Horizontal Fragmentation.

A **horizontal fragment** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment are specified by a condition on one or more attributes of the relation. Often, only a single attribute is involved.

For example, we may define three horizontal fragments on the EMPLOYEE relation with the following conditions: (Dno = 5), (Dno = 4), and (Dno = 1)—each fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (Dnum = 5), (Dnum = 4), and (Dnum = 1)—each fragment contains the PROJECT tuples controlled by a particu-lar department.

Horizontal fragmentation divides a relation *horizontally* by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites in the distributed system. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. This way, related data between the primary and the secondary relations gets fragmented in the same way.

b) Vertical Fragmentation.

Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation** divides a relation “vertically” by columns. A **vertical fragment** of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—Name, Bdate, Address, and Sex—and the second includes work-related information—Ssn, Salary, Super_ssn, and Dno. This vertical fragmentation is not quite proper, because if the two fragments are stored separately, we cannot put the original employee tuples back together, since there is *no common attribute* between the two fragments.

It is necessary to include the primary key or some candidate key attribute in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the Ssn attribute to the personal information fragment.

Notice that each horizontal fragment on a relation R can be specified in the relational algebra by a $\sigma_{C_i}(R)$ operation. A set of horizontal fragments whose conditions C_1, C_2, ..., C_n include all the tuples in R—that is, every tuple in R satisfies (C_1 OR C_2 OR ... OR C_n)—is called a **complete horizontal fragmentation** of R. In many cases a complete horizontal fragmentation is also **disjoint**; that is, no tuple in R satisfies (C_i AND C_j) for any $i \neq j$. Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation R from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation R can be specified by a $\pi_{L_i}(R)$ operation in the relational algebra. A set of vertical fragments whose projection lists L_1, L_2, ..., L_n include all the attributes in R but share only the primary key attribute of R is called a **complete vertical fragmentation** of R. In this case the projection lists satisfy the following two conditions:

$L_1 \cup L_2 \cup ... \cup L_n$ = ATTRS(R).

$L_i \cap L_j$ = PK(R) for any $i \neq j$, where ATTRS(R) is the set of attributes of R and PK(R) is the primary key of R.

To reconstruct the relation R from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the vertical fragments (assuming no horizontal fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal fragmentation may also have been applied. The two vertical fragments of the EMPLOYEE relation with projection lists L_1 = {Ssn, Name, Bdate, Address, Sex} and L_2 = {Ssn, Salary, Super_ssn, Dno} constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation by the conditions (Salary > 50000) and (Dno = 4); they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists L_1 = {Name, Address} and L_2 = {Ssn, Name, Salary}; these lists violate both conditions of a complete vertical fragmentation.

Mixed (Hybrid) Fragmentation. We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case, the original relation can be reconstructed by applying UNION *and* OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a **fragment** of a relation R can be specified by a SELECT-PROJECT combination of operations $\pi_L(\sigma_C(R))$. If

= TRUE (that is, all tuples are selected) and $L \neq$ ATTRS(R), we get a vertical fragment, and if $C \neq$ TRUE and L = ATTRS(R), we get a horizontal fragment. Finally, if
$\neq$ TRUE and $L \neq$ ATTRS(R), we get a mixed fragment. Notice that a relation can itself be considered a fragment with C = TRUE and L = ATTRS(R). In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful—although not necessary—to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to sites of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is stored. If a fragment is stored at more than one site, it is said to be **replicated**. We discuss data replication and allocation next.

DATA REPLICATION AND ALLOCATION

- Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a **fully replicated distributed database**. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval for global queries because the results of such queries can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations drastically, since a single logical update must be performed on every copy of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there was no replication.
- The other extreme from full replication involves having **no replication**—that is, each fragment is stored at exactly one site. In this case, all fragments *must be* disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation**.

- Between these two extremes, we have a wide spectrum of **partial replication** of the data—that is, some fragments of the database may be replicated whereas others may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers—such as sales forces, financial planners, and claims adjustors—carry partially replicated databases with them on laptops and PDAs and synchronize them periodically with the server database. A description of the replication of fragments is sometimes called a **replication schema**.

- Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called **data distribution** (or **data allocation**). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For example, if high availability is required, transactions can be submitted at any site, and most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database. Suppose that the company has three computer sites— one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the EMPLOYEE and PROJECT information for the employees *who work in that department* and the projects *controlled by that department.* Further, we assume that these sites mainly access the Name, Ssn, Salary, and Super_ssn attributes of EMPLOYEE. Site 1 is used by company headquarters and accesses all employee and project information regularly, in addition to keeping track of DEPENDENT information for insurance purposes.

According to these requirements, the whole database can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, first we can horizontally fragment DEPARTMENT by its key Dnumber. Then we apply derived fragmentation to the EMPLOYEE, PROJECT, and DEPT_LOCATIONS relations based on their foreign keys for department number—called Dno, Dnum, and Dnumber, respectively.

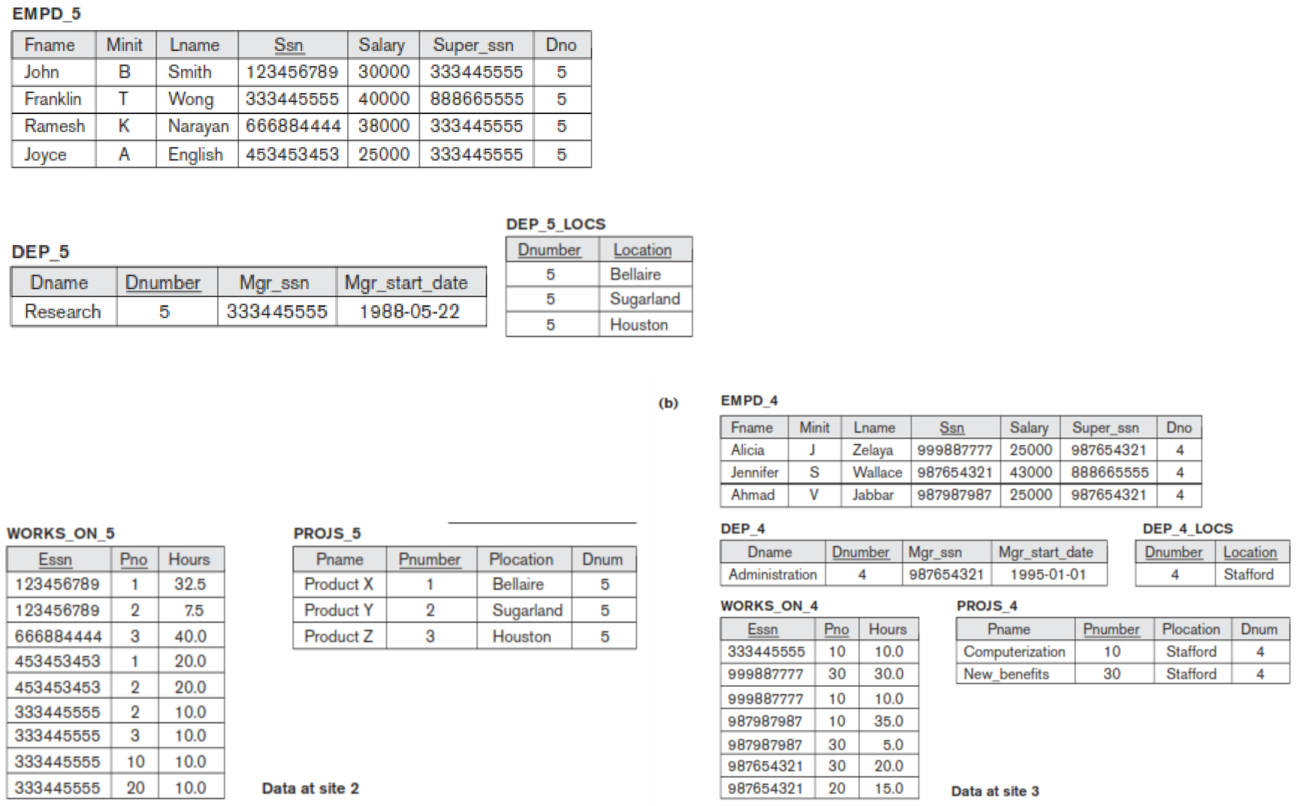
We can vertically fragment the resulting EMPLOYEE fragments to include only the attributes {Name, Ssn, Salary, Super_ssn, Dno}. Figure shows the mixed fragments EMPD_5 and EMPD_4, which include the EMPLOYEE tuples satisfying the conditions Dno = 5 and Dno = 4, respectively. The horizontal fragments of PROJECT, DEPARTMENT, and DEPT_LOCATIONS are similarly fragmented by department number. All these fragments—stored at sites 2 and 3—are replicated because they are also stored at headquarters—site 1.

We must now fragment the WORKS_ON relation and decide which fragments of WORKS_ON to store at sites 2 and 3. We are confronted with the problem that no attribute of WORKS_ON directly indicates the department to which each tuple belongs. In fact, each tuple in WORKS_ON relates an employee e to a project *P*. We could fragment WORKS_ON based on the department *D* in which e works *or* based on the department *D* that controls *P*. Fragmentation becomes easy if we have a constraint stating that *D* = *D* for all WORKS_ON tuples—that is, if employees can work only on projects controlled by the department they work for. However, there is no such constraint in our database. For example, the WORKS_ON tuple <333445555, 10, 10.0> relates an employee who works for department 5 with a project controlled by department 4. In this case, we could fragment WORKS_ON based on the department in which the employee works (which is expressed by the condition *C*) and then fragment further based on the department that controls the projects that employee is working on, as shown in Figure.

In Figure, the union of fragments G_1, G_2, and G_3 gives all WORKS_ON tuples for employees who work for department 5. Similarly, the union of fragments G_4, G_5, and G_6 gives all WORKS_ON tuples for employees who work for department 4. On the other hand, the union of fragments G_1, G_4, and G_7 gives all WORKS_ON tuples for projects controlled by department 5. The condition for each of the fragments G_1 through G_9 is shown in figure. The relations that represent M:N relationships, such as WORKS_ON, often have several possible logical fragmentations. In our distribution in Figure 25.8, we choose to include all fragments that can be joined to either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments G_1, G_2, G_3, G_4, and G_7 at site 2 and the union of fragments G_4, G_5, G_6, G_2, and G_8 at site 3. Notice that fragments G_2 and G_4 are replicated at both sites. This allocation strategy permits the join between the local EMPLOYEE or PROJECT fragments at site 2 or site 3 and the local WORKS_ON fragment to be per-formed completely locally. This clearly demonstrates how complex the problem of database fragmentation and allocation is for large databases. The Selected Bibliography at the end of this chapter discusses some of the work done in this area.

Figure. Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.

(a)



EMPD_5

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
John	B	Smith	123456789	30000	333445555	5
Franklin	T	Wong	333445555	40000	888665555	5
Ramesh	K	Narayan	666884444	38000	333445555	5
Joyce	A	English	453453453	25000	333445555	5

DEP_5

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22

DEP_5_LOCS

Dnumber	Location
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON_5

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0

PROJS_5

Pname	Pnumber	Plocation	Dnum
Product X	1	Bellaire	5
Product Y	2	Sugarland	5
Product Z	3	Houston	5

Data at site 2

(b)

EMPD_4

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP_4

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Administration	4	987654321	1995-01-01

DEP_4_LOCS

Dnumber	Location
4	Stafford

WORKS_ON_4

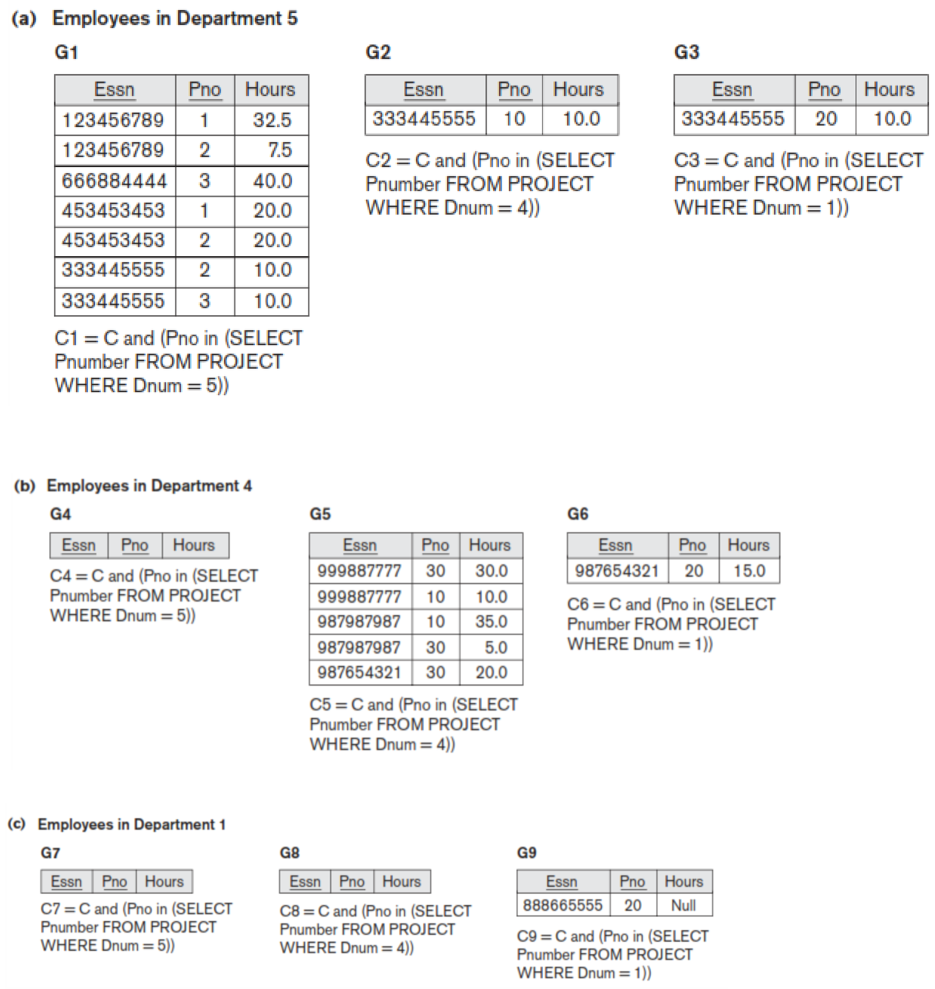
Essn	Pno	Hours
333445555	10	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0

PROJS_4

Pname	Pnumber	Plocation	Dnum
Computerization	10	Stafford	4
New_benefits	30	Stafford	4

Data at site 3

Figure. Complete and disjoint fragments of the WORKS_ON relation.
(a) Fragments of WORKS_ON for employees working in department 5 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=5)]).
(b) Fragments of WORKS_ON for employees working in department 4 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=4)]).
(c) Fragments of WORKS_ON for employees working in department 1 (C=[Essn in (SELECT Ssn FROM EMPLOYEE WHERE Dno=1)]).



(a) Employees in Department 5

G1

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0

C1 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G2

Essn	Pno	Hours
333445555	10	10.0

C2 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G3

Essn	Pno	Hours
333445555	20	10.0

C3 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(b) Employees in Department 4

G4

Essn	Pno	Hours

C4 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G5

Essn	Pno	Hours
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0

C5 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G6

Essn	Pno	Hours
987654321	20	15.0

C6 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

(c) Employees in Department 1

G7

Essn	Pno	Hours

C7 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G8

Essn	Pno	Hours

C8 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G9

Essn	Pno	Hours
888665555	20	Null

C9 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

5.4. TRANSACTION MANAGEMENT IN DISTRIBUTED DATABASES:

- The global and local transaction management software modules, along with the concurrency control and recovery manager of a DDBMS, collectively guarantee the ACID properties of transactions.
- The **global transaction manager** is supporting distributed transactions. The site where the transaction originated can temporarily assume the role of global transaction manager and coordinate the execution of database operations with transaction managers across multiple sites. Transaction managers export their functionality as an interface to the application programs.
- The operations exported by this interface are BEGIN_TRANSACTION, READ or WRITE, END_TRANSACTION, COMMIT_TRANSACTION, and ROLLBACK (or ABORT).
- The manager stores bookkeeping information related to each transaction, such as a unique identifier, originating site, name, and so on. For READ operations, it returns a local copy if valid and available. For WRITE operations, it ensures that updates are visible across all sites containing copies (replicas) of the data item. For ABORT operations, the manager ensures that no effects of the transaction are reflected in any site of the distributed database. For COMMIT operations, it ensures that the effects of a write are persistently recorded on all databases containing copies of the data item. Atomic

termination (COMMIT/ ABORT) of distributed transactions is commonly implemented using the two-phase commit protocol.

- The transaction manager passes to the concurrency controller the database operation and associated information. The controller is responsible for acquisition and release of associated locks. If the transaction requires access to a locked resource, it is delayed until the lock is acquired. Once the lock is acquired, the operation is sent to the runtime processor, which handles the actual execution of the database operation. Once the operation is completed, locks are released and the transaction manager is updated with the result of the operation.

Two-Phase Commit Protocol

The *two-phase commit protocol* (**2PC**) requires a **global recovery manager**, or **coordinator**, to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables) . The two-phase commit protocol has certain drawbacks that led to the development of the three-phase commit protocol.

Three-Phase Commit Protocol

1) The biggest drawback of 2PC is that it is a blocking protocol. Failure of the coordinator blocks all participating sites, causing them to wait until the coordinator recovers. This can cause performance degradation, especially if participants are holding locks to shared resources.
2) Another problematic scenario is when both the coordinator and a participant that has committed crash together. In the two-phase commit protocol, a participant has no way to ensure that all participants got the commit message in the second phase. Hence once a decision to commit has been made by the coordinator in the first phase, participants will commit their transactions in the second phase independent of receipt of a global commit message by other participants. Thus, in the situation that both the coordinator and a committed participant crash together, the result of the transaction becomes uncertain or nondeterministic. Since the transaction has already been committed by one participant, it cannot be aborted on recovery by the coordinator. Also, the transaction cannot be optimistically committed on recovery since the original vote of the coordinator may have been to abort.

These problems are solved by the three-phase commit (3PC) protocol, which essentially divides the second commit phase into two subphases called **prepare-to-commit** and **commit**. The prepare-to-commit phase is used to communicate the result of the vote phase to all participants. If all participants vote yes, then the coordinator instructs them to move into the prepare-to-commit state. The commit subphase is identical to its two-phase counterpart. Now, if the coordinator crashes during this subphase, another participant can see the transaction through to completion. It can simply ask a crashed participant if it received a prepare-to-commit message. If it did not, then it safely assumes to abort. Thus the state of the protocol can be recovered irrespective of which participant crashes. Also, by limiting the time required for a transaction to commit or abort to a maximum time-out period, the protocol ensures that a transaction attempting to commit via 3PC releases locks on time-out.

The main idea is to limit the wait time for participants who have committed and are waiting for a global commit or abort from the coordinator. When a participant receives a precommit message, it knows that the rest of the participants have voted to commit. If a precommit message has not been received, then the participant will abort and release all locks.

Operating System Support for Transaction Management

The following are the main benefits of operating system (OS)-supported transaction management:

Typically, DBMSs use their own semaphores to guarantee mutually exclusive access to shared resources. Since these semaphores are implemented in user space at the level of the DBMS application software, the OS has no knowledge about them. Hence if the OS deactivates a DBMS process holding a lock, other DBMS processes wanting this lock resource get queued. Such a situation can cause serious performance degradation. OS-level knowledge of semaphores can help eliminate such situations.

Specialized hardware support for locking can be exploited to reduce associated costs. This can be of great importance, since locking is one of the most common DBMS operations.

Providing a set of common transaction support operations though the kernel allows application developers to focus on adding new features to their products as opposed to reimplementing the common functionality for each application. For example, if different DDBMSs are to coexist on the same machine and they chose the two-phase commit protocol, then it is more beneficial to have this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

5.5. OBJECT DATABASE CONCEPTS

Introduction to Object-Oriented Concepts and Features

- An **object** typically has two components:
  1) state (value)
  2) Behavior (operations).
- It can have a *complex data structure* as well as *specific operations* defined by the programmer.
- Objects in an OOPL exist only during program execution; therefore, they are called ***transient objects.***
- An OO database can extend the existence of objects so that they are stored permanently in a database, and hence the objects become ***persistent objects*** that exist beyond program termination and can be retrieved later and shared by other programs.
- In other words, OO databases store persistent objects permanently in secondary storage, and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as indexing mechanisms to efficiently locate the objects, concurrency control to allow object sharing among concurrent programs, and recovery from failures. An OO database system will typically interface with one or more OO programming languages to provide persistent and shared object capabilities.

- The internal structure of an object in OOPLs includes the specification of **instance variables**, which hold the values that define the internal state of the object. An instance variable is similar to the concept of an *attribute* in the relational model, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some models insist that all operations a user can apply to an object must be predefined. This forces a *complete encapsulation* of objects. This rigid approach has been relaxed in most OO data models for two reasons.

- First, database users often need to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects. Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify on the fly.

▪ To encourage encapsulation, an operation is defined in two parts. The first part, called the *signature* or *interface* of the operation, specifies the operation name and arguments (or parameters). The second part, called the *method* or *body*, specifies the *implementation* of the operation, usually written in some general-purpose programming language. Operations can be invoked by passing a *message* to an object, which includes the operation name and the parameters. The object then executes the method for that operation. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations. Hence, encapsulation provides a form of data and operation independence.

- Another key concept in OO systems is that of type and class hierarchies and *inheritance.* This permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes. This makes it easier to develop the data types of a system incrementally, and to *reuse* existing type definitions when creating new types of objects.
- One problem in early OO database systems involved representing *relationships* among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships because it is useful to identify these relationships and make them visible to users. The ODMG object database standard has recognized this need and it explicitly represents binary relationships via a pair of *inverse references*.
- Another OO concept is *operator overloading*, which refers to an operation's ability to be applied to different types of objects; in such a situation, an *operation name* may refer to several distinct *implementations,* depending on the type of object it is applied to. This feature is also called *operator polymorphism.* For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle. This may require the use of *late binding* of the operation name to the appropriate method at runtime, when the type of object to which the operation is applied becomes known.

Object Identity, and Objects versus Literals:

One goal of an ODMS (Object Data Management System) is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, an ODMS provides a **unique identity** to each independent object stored in the data-base. This unique identity is typically implemented via a unique, system-generated **object identifier (OID)**. The value of an OID is not visible to the external user, but is used internally by the system to identify each object uniquely and to create and manage inter-object references. The OID can be assigned to program variables of the appropriate type when needed.

The main property required of an OID is that it be **immutable**; that is, the OID value of a particular object should not change. This preserves the identity of the real-world object being represented. Hence, an ODMS must have some mechanism for generating OIDs and preserving the immutability property. It is also desirable that each OID be used only once; that is, even if an object is removed from the database, its OID should not be assigned to another object. These two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed or corrected.

We can compare this with the relational model, where each relation must have a primary key attribute whose value identifies each tuple uniquely. In the relational model, if the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same real-world object (for example, the object identifier may be represented as Emp_id in one relation and as Ssn in another).

It is inappropriate to base the OID on the physical address of the object in storage, since the physical address can change after a physical reorganization of the database. However, some early ODMSs have used the physical address as the OID to increase the efficiency of object retrieval. If the physical address of the object changes, an *indirect pointer* can be placed at the former address, which gives the new physical location of the object. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the current physical address of the object in storage.

Some early OO data models required that everything—from a simple value to a complex object—was represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. This allows two identical basic values to have different OIDs, which can be useful in some cases.
For example, the integer value 50 can sometimes be used to mean a weight in kilograms and at other times to mean the age of a person. Then, two basic objects with distinct OIDs could be created, but both objects would represent the integer value 50. Although useful as a theoretical model, this is not very practical, since it leads to the generation of too many OIDs. Hence, most OO database systems allow for the representation of both objects and **literals** (or values).

Every object must have an immutable OID, whereas a literal value has no OID and its value just stands for itself. Thus, a literal value is typically stored within an object and *cannot be referenced* from other objects. In many systems, complex structured literal values can also be created without having a corresponding OID if needed.

Complex Type Structures for Objects and Literals:

Another feature of an ODMS (and ODBs in general) is that objects and literals may have a *type structure* of *arbitrary complexity* in order to contain all of the necessary information that describes the object or literal.
In contrast, in traditional database systems, information about a complex object is often *scattered* over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation. In ODBs, a complex type may be constructed from other types by *nesting* of **type constructors**.
The three most basic constructors are
1) atom
2) struct (or tuple)
3) collection.

1)Atom:
One type constructor has been called the **atom** constructor. This includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages: integers, strings, floating point numbers, enumerated types, Booleans, and so on. They are called **single-valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value.

2) struct:
A second type constructor is referred to as the **struct** (or **tuple**) constructor. This can create standard structured types, such as the tuples (record types) in the basic relational model. A structured type is made up of several components, and is also sometimes referred to as a *compound* or *composite* type. More accurately, the struct constructor is not considered to be a type, but rather a **type generator**, because many different structured types can be created.
For example, two different structured types that can be created are:
struct Name<FirstName: string, MiddleInitial: char, LastName: string>
struct CollegeDegree<Major: string, Degree: string, Year: date>.

To create complex nested type structures in the object model, the *collection* type constructors are needed. Notice that the type constructors *atom* and *struct* are the only ones available in the original (basic) relational model.

3) <u>Collection</u>:

Collection (or *multivalued*) type constructors include the **set(T)**, **list(T)**, **bag(T)**, **array(T)**, and **dictionary(K,T)** type constructors.

These allow part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be **type generators** because many different types can be created. For example, set(*string*), set(*integer*), and set(*Employee*) are three different types that can be created from the *set* type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type set(*string*) must be string values.

The *atom constructor* is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly. The *tuple constructor* can create structured values and objects of the form <a_1:i_1, a_2:i_2, ..., a_n:i_n>, where each a_j is an attribute name and each i_j is a value or an OID.

The other commonly used constructors are collectively referred to as collection types, but have individual differences among them.

1) The **set constructor** will create objects or literals that are a set of *distinct* elements {i_1, i_2, ..., i_n}, all of the same type.
2) The **bag constructor** (sometimes called a *multiset*) is similar to a set except that the elements in a bag *need not be distinct*.
3) The **list constructor** will create an *ordered list* [i_1, i_2, ..., i_n] of OIDs or values of the same type. A list is similar to a **bag** except that the elements in a list are *ordered*, and hence we can refer to the first, second, or *j*th element.
4) The **array constructor** creates a single-dimensional array of elements of the same type. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size.
5) The **dictionary constructor** creates a collection of two tuples (*K*, *V*), where the value of a key *K* can be used to retrieve the corresponding value *V*.

The main characteristic of a collection type is that its objects or values will be a *collection of objects or values of the same type* that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The **tuple** type constructor is often called a **structured type**, since it corresponds to the **struct** construct in the C and C++ programming languages.

An **object definition language** (**ODL**) that incorporates the preceding type constructors can be used to define the object types for a particular database application.

The type constructors can be used to define the *data structures* for an OO *database schema*. Figure.1 shows how we may declare EMPLOYEE and DEPARTMENT types.

In Figure.1, the attributes that refer to other objects—such as Dept of EMPLOYEE or Projects of DEPARTMENT—are basically OIDs that serve as **references** to other objects to represent *relationships* among the objects. For example, the attribute Dept of EMPLOYEE is of type DEPARTMENT, and hence is used to refer to a specific DEPARTMENT object (the DEPARTMENT object where the employee works). The value of such an attribute would be an OID for a specific DEPARTMENT object. A binary relationship can be represented in one direction, or it can have an *inverse reference*. The latter representation makes it easy to traverse the relationship in both directions. For example, in Figure.1 the attribute Employees of DEPARTMENT has as its value a *set of references* (that is, a set of OIDs) to objects of type EMPLOYEE; these are the employees who work for the DEPARTMENT. The inverse is the reference attribute Dept of EMPLOYEE.

```
define type
EMPLOYEE
```

```
      tuple (  Fname:      string;
               Minit:      char;
               Lname:      string;
               Ssn:        string;
               Birth_date: DATE;
               Address:    string;
               Sex:        char;
               Salary:     float;
               Supervisor: EMPLOYEE;
               Dept:       DEPARTMENT;);

   define type DATE
      tuple (  Year:       integer;
               Month:      integer;
               Day:        integer;      );
   define type
   DEPARTMENT
      tuple (  Dname:      string;
               Dnumber:    integer;
               Mgr:        tuple (    Manager:     EMPLOYEE;
                                      Start_date:  DATE;  );
               Locations:  set(string);
               Employees:  set(EMPLOYEE);
               Projects:   set(PROJECT);  );
```

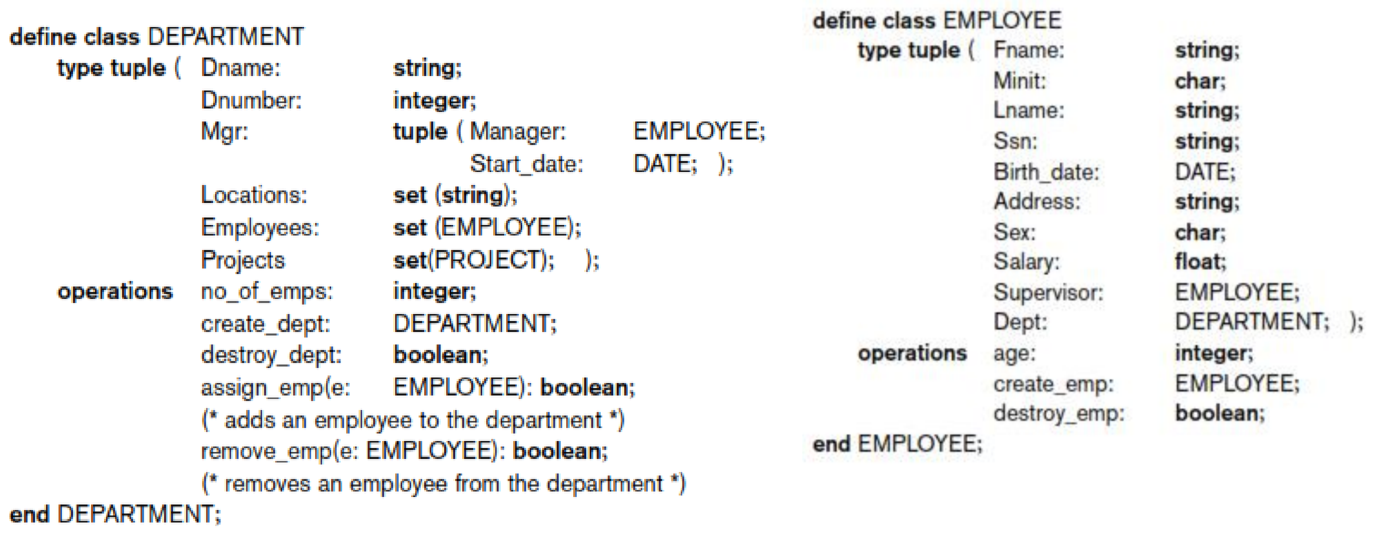
Figure.1. Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

Encapsulation of Operations and Persistence of Objects:

Encapsulation of Operations:

- The concepts of encapsulation is applied to database objects in ODBs by defining the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations. Still other operations may perform a combination of retrieval, calculation, and update.
- In general, the **implementation** of an operation can be specified in a *general-purpose programming language* that provides flexibility and power in defining the operations.
- **The external users of the object are only made aware of the interface of the operations,** which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of any hidden internal data structures of the object and the implementation of the operations that access these structures. The interface part of an operation is sometimes called the **signature**, and the operation implementation is sometimes called the **method**.
- For database applications, the requirement that all objects be completely encapsulated is too stringent. One way to relax this requirement is to divide the structure of an object into **visible** and **hidden** attributes (instance variables).
- Visible attributes can be seen by and are directly accessible to the database users and programmers via the query language.

- The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations. Most ODMSs employ high-level query languages for accessing visible attributes.
- **The term class is often used to refer to a type definition, along with the definitions of the operations for that type.**
- Figure.2 shows how the type definitions in Figure.1 can be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition.
- A method (implementation) for each operation must be defined elsewhere using a programming language. Typical operations include the **object constructor** operation (often called *new*), which is used to create a new object, and the **destructor** operation, which is used to destroy (delete) an object. A number of **object modifier** operations can also be declared to modify the states (values) of various attributes of an object.
- Additional operations can **retrieve** information about the object.



```
define class DEPARTMENT
    type tuple (  Dname:          string;
                  Dnumber:        integer;
                  Mgr:            tuple ( Manager:      EMPLOYEE;
                                          Start_date:   DATE;   );
                  Locations:      set (string);
                  Employees:      set (EMPLOYEE);
                  Projects        set(PROJECT);    );
    operations    no_of_emps:     integer;
                  create_dept:    DEPARTMENT;
                  destroy_dept:   boolean;
                  assign_emp(e:   EMPLOYEE): boolean;
                  (* adds an employee to the department *)
                  remove_emp(e: EMPLOYEE): boolean;
                  (* removes an employee from the department *)
end DEPARTMENT;
```

```
define class EMPLOYEE
    type tuple (  Fname:        string;
                  Minit:        char;
                  Lname:        string;
                  Ssn:          string;
                  Birth_date:   DATE;
                  Address:      string;
                  Sex:          char;
                  Salary:       float;
                  Supervisor:   EMPLOYEE;
                  Dept:         DEPARTMENT;   );
    operations    age:          integer;
                  create_emp:   EMPLOYEE;
                  destroy_emp:  boolean;
end EMPLOYEE;
```

Figure .2. Adding operations to the definitions of EMPLOYEE and DEPARTMENT.

- An operation is typically applied to an object by using the **dot notation**. For example, if d is a reference to a DEPARTMENT object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted).
- The only exception is the constructor operation, which returns a reference to a new DEPARTMENT object. Hence, it is customary in some OO models to have a default name for the constructor operation that is the name of the class itself, although this was not used in Figure .2. The dot notation is also used to refer to attributes of an object—for example, by writing d.Dnumber or d.Mgr_Start_date.

<u>**Specifying Object Persistence via Naming and Reachability:**</u>

An ODBS is often closely coupled with an object-oriented programming language (OOPL). The OOPL is used to specify the method (operation) implementations as well as other application code. Not all objects are meant to be stored permanently in the data-base.

- **Transient objects** exist in the executing program and disappear once the program terminates.
- **Persistent objects** are stored in the database and persist after program termination. The typical mechanisms for making an object persistent are *naming* and *reachability.*

- The **naming mechanism** involves giving an object a unique persistent name within a particular database. This persistent **object name** can be given via a specific statement or operation in the program, as shown in Figure.3. The named persistent objects are used as **entry points** to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called **reachability**.

- The reachability mechanism works by making the object reachable from some other persistent object. An object *B* is said to be **reachable** from an object *A* if a sequence of references in the database lead from object *A* to object *B*.

If we first create a named persistent object *N*, whose state is a *set* (or possibly a *bag*) of objects of some class *C*, we can make objects of *C* persistent by *adding them* to the set, thus making them reachable from *N*. Hence, *N* is a named object that defines a **persistent collection** of objects of class *C*. In the object model standard, *N* is called the **extent** of *C* .

For example, we can define a class DEPARTMENT_SET whose objects are of type set(DEPARTMENT). We can create an object of type DEPARTMENT_SET, and give it a persistent name ALL_DEPARTMENTS, as shown in Figure 3. Any DEPARTMENT object that is added to the set of ALL_DEPARTMENTS by using the add_dept operation becomes persistent by virtue of its being reachable from ALL_DEPARTMENTS.

```
define class DEPARTMENT_SET
    type set (DEPARTMENT);
    operations   add_dept(d: DEPARTMENT):    boolean;
            (* adds a department to the DEPARTMENT_SET object *)
                 remove_dept(d: DEPARTMENT): boolean;
            (* removes a department from the DEPARTMENT_SET object *)
                 create_dept_set:       DEPARTMENT_SET;
                 destroy_dept_set:      boolean;
end DEPARTMENT_SET;
...
persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
(* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
...
d:= create_dept;
(* create a new DEPARTMENT object in the variable d *)
...
b:= ALL_DEPARTMENTS.add_dept(d);
(* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```

Figure. Creating persistent objects by naming and reachability.

- In traditional database models, such as the relational model, *all* objects are assumed to be persistent. Hence, when a table such as EMPLOYEE is created in a relational database, it represents both the *type declaration* for EMPLOYEE and a *persistent set* of *all* EMPLOYEE records (tuples).
- In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) or bag(EMPLOYEE) whose value is the *collection of references* (OIDs) to all persistent EMPLOYEE objects, if this is desired, as shown in Figure 3. This allows transient and persistent objects to follow the same type and class declarations of the ODL and the OOPL. In general, it is possible to define several persistent collections for the same class definition, if desired.

<u>TYPE HIERARCHIES AND INHERITANCE:</u>

Simplified Model for Inheritance:

- Inheritance allows the definition of new types based on other predefined types, leading to a **type** (or **class**) **hierarchy**.
- A type is defined by assigning it a type name, and then defining a number of attributes (instance variables) and operations (methods) for the type. In the simplified model, the attributes and operations are together called *functions*, since attributes resemble functions with zero arguments.
- A function name can be used to refer to the value of an attribute or to refer to the resulting value of an operation (method). We use the term **function** to refer to both attributes *and* operations, since they are treated similarly in a basic introduction to inheritance.
- A type in its simplest form has a **type name** and a list of visible (*public*) **functions**. When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:

  TYPE_NAME: function, function, ..., function

For example, a type that describes characteristics of a PERSON may be defined as follows:

  PERSON: Name, Address, Birth_date, Age, Ssn

- In the PERSON type, the Name, Address, Ssn, and Birth_date functions can be implemented as stored attributes, whereas the Age function can be implemented as an operation that calculates the Age from the value of the Birth_date attribute and the current date.
- The concept of **subtype** is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which is referred to as the **supertype**. For example, suppose that we want to define two new types EMPLOYEE and STUDENT as follows:

  EMPLOYEE: Name, Address, Birth_date, Age, Ssn, Salary, Hire_date, Seniority
  STUDENT: Name, Address, Birth_date, Age, Ssn, Major, Gpa

- Since both STUDENT and EMPLOYEE include all the functions defined for PERSON plus some additional functions of their own, we can declare them to be **subtypes** of PERSON. Each will inherit the previously defined functions of PERSON—namely, Name, Address, Birth_date, Age, and Ssn.
- For STUDENT, it is only necessary to define the new (local) functions Major and Gpa, which are not inherited. Presumably, Major can be defined as a stored attribute, whereas Gpa may be implemented as an operation that calculates the student's grade point average by accessing the Grade values that are internally stored (hidden) within each STUDENT object as *hidden attributes.* For EMPLOYEE, the Salary and Hire_date functions may be stored attributes, whereas Seniority may be an operation that calculates Seniority from the value of Hire_date.

- Therefore, we can declare EMPLOYEE and STUDENT as follows:

EMPLOYEE subtype-of PERSON: Salary, Hire_date, Seniority
STUDENT subtype-of PERSON: Major, Gpa

- In general, a subtype includes *all* of the functions that are defined for its supertype plus some additional functions that are *specific* only to the subtype. Hence, it is pos-sible to generate a **type hierarchy** to show the supertype/subtype relationships among all the types declared in the system.

- Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are persistently stored in the database.

- Constraints on Extents Corresponding to a Type Hierarchy. In most ODBs, an **extent** is defined to store the collection of persistent objects for each type or sub-type. In this case, the constraint is that every object in an extent that corresponds to a subtype must also be a member of the *extent* that corresponds to its supertype. Some OO database systems have a predefined system type (called the ROOT class or the OBJECT class) whose extent contains all the objects in the system.[18]
- Classification then proceeds by assigning objects into additional subtypes that are meaningful to the application, creating a **type hierarchy** (or **class hierarchy**) for the system. All extents for system- and user-defined classes are subsets of the extent corresponding to the class OBJECT, directly or indirectly. In the ODMG model, the user may or may not specify an extent for each class (type), depending on the application.

- An extent is a named persistent object whose value is a **persistent collection** that holds a collection of objects of the same type that are stored permanently in the database. The objects can be accessed and shared by multiple programs. It is also possible to create a **transient collection**, which exists temporarily during the execution of a program but is not kept when the program terminates. For example, a transient collection may be created in a program to hold the result of a query that selects some objects from a persistent collection and copies those objects into the transient collection. The program can then manipulate the objects in the transient collection, and once the program terminates, the transient collection ceases to exist. In general, numerous collections—transient or persistent—may contain objects of the same type.

<u>Polymorphism of Operations (Operator Overloading).</u>

- Another characteristic of OO systems in general is that they provide for **polymorphism** of operations, which is also known as **operator overloading**. This concept allows the same *operator name* or *symbol* to be bound to two or more different *implementations* of the operator, depending on the type of objects to which the operator is applied.
- A simple example from programming languages can illustrate this concept. In some languages, the operator symbol "+" can mean different things when applied to operands (objects) of different types. If the operands of "+" are of type *integer*, the operation invoked is integer addition. If the operands of "+" are of type *floating point*, the operation invoked is floating point addition. If the operands of "+" are of type *set*, the operation invoked is set union. The compiler can determine which operation to execute based on the types of operands supplied.
- In OO databases, a similar situation may occur. We can use the GEOMETRY_OBJECT example to illustrate operation polymorphism in ODB.
- In this example, the function Area is declared for all objects of type GEOMETRY_OBJECT. However, the implementation of the method for Area may differ for each subtype of GEOMETRY_OBJECT. One possibility is to have a general implementation for calculating the area of a generalized GEOMETRY_OBJECT (for example, by writing a general algorithm to calculate the area of a polygon) and then to rewrite more efficient algorithms to calculate the areas of specific types of geo-metric objects, such as a circle, a rectangle, a triangle, and so on. In this case, the Area function is *overloaded* by different implementations.

- The ODMS must now select the appropriate method for the Area function based on the type of geometric object to which it is applied. In strongly typed systems, this can be done at compile time, since the object types must be known. This is termed **early** (or **static**) **binding**. However, in systems with weak typing or no typing (such as Smalltalk and LISP), the type of the object to which a function is applied may not be known until runtime. In this case, the function must check the type of object at runtime and then invoke the appropriate method. This is often referred to as **late** (or **dynamic**) **binding**.

Multiple Inheritance and Selective Inheritance.

- **Multiple inheritance** occurs when a certain subtype *T* is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes. For example, we may create a subtype ENGINEERING_MANAGER that is a subtype of both MANAGER and ENGINEER. This leads to the creation of a **type lattice** rather than a type hierarchy.
- One problem that can occur with multiple inheritance is that the supertypes from which the subtype inherits may have distinct functions of the same name, creating an ambiguity. For example, both MANAGER and ENGINEER may have a function called Salary. If the Salary function is implemented by different methods in the MANAGER and ENGINEER supertypes, an ambiguity exists as to which of the two is inherited by the subtype ENGINEERING_MANAGER.
- It is possible, however, that both ENGINEER and MANAGER inherit Salary from the same supertype (such as EMPLOYEE) higher up in the lattice. The general rule is that if a function is inherited from some *common supertype*, then it is inherited only once. In such a case, there is no ambiguity; the problem only arises if the functions are distinct in the two supertypes.
- There are several techniques for dealing with ambiguity in multiple inheritance.

  1) One solution is to have the system check for ambiguity when the subtype is created, and to let the user explicitly choose which function is to be inherited at this time.
  2) A second solution is to use some system default.
  3) A third solution is to disallow multiple inheritance altogether if name ambiguity occurs, instead forcing the user to change the name of one of the functions in one of the supertypes.

- Indeed, some OO systems do not permit multiple inheritance at all. In the object database standard, multiple inheritance is allowed for operation inheritance of interfaces, but is not allowed for EXTENDS inheritance of classes.
- **Selective inheritance** occurs when a subtype inherits only some of the functions of a supertype. Other functions are not inherited. In this case, an EXCEPT clause may be used to list the functions in a supertype that are *not* to be inherited by the sub-type. The mechanism of selective inheritance is not typically provided in ODBs, but it is used more frequently in artificial intelligence applications.

5.6 . OBJECT-RELATIONAL FEATURES:

Object Database Extensions to SQL

- SQL was first specified by Chamberlin and Boyce (1974) and underwent enhancements and standardization in 1989 and 1992. The language continued its evolution with a new standard, initially called SQL3 while being developed, and later known as SQL:99 for the parts of SQL3 that were approved into the standard. Starting with the version of SQL known as SQL3, features from object databases were incorporated into the SQL standard.
- At first, these extensions were known as SQL/Object, but later they were incorporated in the main part of SQL, known as SQL/Foundation. We will use that latest standard, SQL:2008.

- The relational model with object database enhancements is sometimes referred to as the **object-relational model**. Additional revisions were made to SQL in 2003 and 2006 to add features related to XML.
- The following are some of the object database features that have been included in SQL:

1) Some **type constructors** have been added to specify complex objects. These include the *row type*, which corresponds to the tuple (or struct) constructor. An *array type* for specifying collections is also provided. Other collection type constructors, such as *set*, *list*, and *bag* constructors, were not part of the original SQL/Object specifications but were later included in the standard.

2) A mechanism for specifying **object identity** through the use of *reference type* is included.

3) Encapsulation of operations is provided through the mechanism of user-defined types (UDTs) that may include operations as part of their declara-tion. These are somewhat similar to the concept of *abstract data types* that were developed in programming languages. In addition, the concept of user-defined routines (UDRs) allows the definition of general methods (opera-tions).

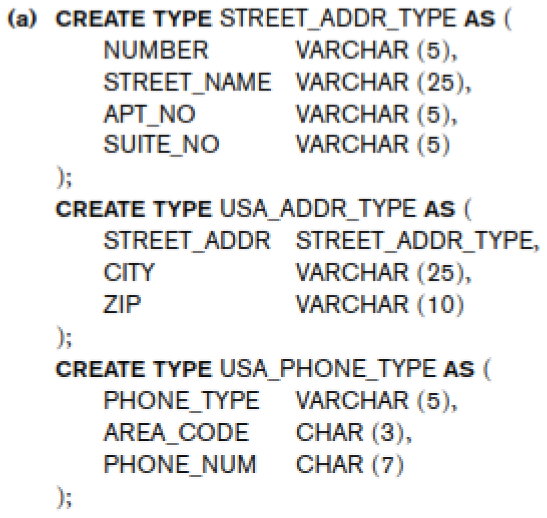
4) Inheritance mechanisms are provided using the keyword UNDER.

<u>User-Defined Types and Complex Structures for Objects:</u>

- To allow the creation of complex-structured objects, and to separate the declaration of a type from the creation of a table, SQL now provides **user-defined types** (**UDT**s). In addition, four collection types have been included to allow for multival-ued types and attributes in order to specify complex-structured objects rather than just simple (flat) records. The user will create the UDTs for a particular application as part of the database schema. A **UDT** may be specified in its simplest form using the following syntax:

CREATE TYPE TYPE_NAME AS (<component declarations>);

  - Figure 4 illustrates some of the object concepts in SQL. We will explain the examples in this figure gradually as we explain the concepts. First, a UDT can be used as either the type for an attribute or as the type for a table. By using a UDT as the type for an attribute within another UDT, a complex structure for objects (tuples) in a table can be created, much like that achieved by nesting type constructors. This is similar to using the *struct* type constructor.



```
(a) CREATE TYPE STREET_ADDR_TYPE AS (
        NUMBER          VARCHAR (5),
        STREET_NAME     VARCHAR (25),
        APT_NO          VARCHAR (5),
        SUITE_NO        VARCHAR (5)
    );
    CREATE TYPE USA_ADDR_TYPE AS (
        STREET_ADDR     STREET_ADDR_TYPE,
        CITY            VARCHAR (25),
        ZIP             VARCHAR (10)
    );
    CREATE TYPE USA_PHONE_TYPE AS (
        PHONE_TYPE      VARCHAR (5),
        AREA_CODE       CHAR (3),
        PHONE_NUM       CHAR (7)
    );
```

```
(b) CREATE TYPE PERSON_TYPE AS (
        NAME            VARCHAR (35),
        SEX             CHAR,
        BIRTH_DATE      DATE,
        PHONES          USA_PHONE_TYPE ARRAY [4],
        ADDR            USA_ADDR_TYPE
    INSTANTIABLE
    NOT FINAL
    REF IS SYSTEM GENERATED
    INSTANCE METHOD AGE() RETURNS INTEGER;
    CREATE INSTANCE METHOD AGE() RETURNS INTEGER
        FOR PERSON_TYPE
        BEGIN
            RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                      TODAY'S DATE AND SELF.BIRTH_DATE */
        END;
    );


(c) CREATE TYPE GRADE_TYPE AS (
        COURSENO        CHAR (8),
        SEMESTER        VARCHAR (8),
        YEAR            CHAR (4),
        GRADE           CHAR
    );
    CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (
        MAJOR_CODE      CHAR (4),
        STUDENT_ID      CHAR (12),
        DEGREE          VARCHAR (5),
        TRANSCRIPT      GRADE_TYPE ARRAY [100]


 INSTANTIABLE
 NOT FINAL
 INSTANCE METHOD GPA() RETURNS FLOAT;
 CREATE INSTANCE METHOD GPA() RETURNS FLOAT
     FOR STUDENT_TYPE
     BEGIN
         RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
                   SELF.TRANSCRIPT */
     END;
 );
 CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (
     JOB_CODE            CHAR (4),
     SALARY              FLOAT,
     SSN                 CHAR (11)
 INSTANTIABLE
 NOT FINAL
 );
 CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
     DEPT_MANAGED        CHAR (20)
 INSTANTIABLE
 );
```

```
(d) CREATE TABLE PERSON OF PERSON_TYPE
        REF IS PERSON_ID SYSTEM GENERATED;
    CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
        UNDER PERSON;
    CREATE TABLE MANAGER OF MANAGER_TYPE
        UNDER EMPLOYEE;
    CREATE TABLE STUDENT OF STUDENT_TYPE
        UNDER PERSON;
```

```
(e) CREATE TYPE COMPANY_TYPE AS (
        COMP_NAME        VARCHAR (20),
        LOCATION         VARCHAR (20));
    CREATE TYPE EMPLOYMENT_TYPE AS (
        Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
        Company REF (COMPANY_TYPE) SCOPE (COMPANY) );
    CREATE TABLE COMPANY OF COMPANY_TYPE (
        REF IS COMP_ID SYSTEM GENERATED,
        PRIMARY KEY (COMP_NAME) );
    CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;
```

Figure. Illustrating some of the object features of SQL.
(a) Using UDTs as types for attributes such as Address and Phone,
(b) Specifying UDT for PERSON_TYPE,
(c) Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two sub-types of PERSON_TYPE
(d) Creating tables based on some of the UDTs, and illustrating table inheritance,
(e) Specifying relation-ships using REF and SCOPE.

- For example, in Figure 4(a), the UDT STREET_ADDR_TYPE is used as the type for the STREET_ADDR attribute in the UDT USA_ADDR_TYPE. Similarly, the UDT USA_ADDR_TYPE is in turn used as the type for the ADDR attribute in the UDT PERSON_TYPE in Figure 4(b).
- If a UDT does not have any operations, as in the examples in Figure 4(a), it is possible to use the concept of ROW TYPE to directly create a structured attribute by using the keyword ROW. For example, we could use the following instead of declaring STREET_ADDR_TYPE as a separate type as in Figure 4(a):

```
CREATE TYPE USA_ADDR_TYPE AS (
    STREET_ADDR        ROW (          NUMBER       VARCHAR (5),
                        STREET_NAME        VARCHAR (25),
                        APT_NO        VARCHAR (5),
                        SUITE_NO      VARCHAR (5) ),
    CITY           VARCHAR (25),
    ZIP            VARCHAR (10)
);
```

- To allow for collection types in order to create complex-structured objects, four constructors are now included in SQL: ARRAY, MULTISET, LIST, and SET. These are similar to the type constructors. In the initial specification of SQL/Object, only the ARRAY type was specified, since it can be used to simulate the other types, but the three additional collection types were included in the latest version of the SQL standard. In Figure 4(b), the PHONES attribute of PERSON_TYPE has as its type an array whose elements are of the previously defined UDT USA_PHONE_TYPE. This array has a maximum of four elements, meaning that we can store up to four phone numbers per person. An array can also have no maximum number of elements if desired.

- An array type can have its elements referenced using the common notation of square brackets. For example, PHONES[1] refers to the first location value in a PHONES attribute. A built-in function CARDINALITY can return the cur-rent number of elements in an array (or any other collection type). For example, PHONES[CARDINALITY (PHONES)] refers to the last element in the array.
- The commonly used dot notation is used to refer to components of a ROW TYPE or a UDT. For example, ADDR.CITY refers to the CITY component of an ADDR attribute.

Object Identifiers Using Reference Types:

- Unique system-generated object identifiers can be created via the **reference type** in the latest version of SQL. For example, in Figure 4(b), the phrase:

  REF IS SYSTEM GENERATED

indicates that whenever a new PERSON_TYPE object is created, the system will assign it a unique system-generated identifier. It is also possible not to have a system-generated object identifier and use the traditional keys of the basic relational model if desired.

- In general, the user can specify that system-generated object identifiers for the indi-vidual rows in a table should be created. By using the syntax:

  REF IS <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;

the user declares that the attribute named <OID_ATTRIBUTE> will be used to iden-tify individual tuples in the table. The options for <VALUE_GENERATION _METHOD> are SYSTEM GENERATED or DERIVED. In the former case, the system will automatically generate a unique identifier for each tuple. In the latter case, the traditional method of using the user-provided primary key value to identify tuples is applied.

Creating Tables Based on the UDTs:

- For each UDT that is specified to be instantiable via the phrase INSTANTIABLE (see Figure 11.4(b)), one or more tables may be created. This is illustrated in Figure 4(d), where we create a table PERSON based on the PERSON_TYPE UDT. Notice that the UDTs in Figure 4(a) are non instantiable, and hence can only be used as types for attributes, but not as a basis for table creation. In Figure 4(b), the attribute PERSON_ID will hold the system-generated object identifier whenever a new PERSON record (object) is created and inserted in the table.

Encapsulation of Operations:

- In SQL, a **user-defined type** can have its own behavioral specification by specifying methods (or operations) in addition to the attributes. The general form of a UDT specification with methods is as follows:

CREATE TYPE <TYPE-NAME> (
<LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES> <DECLARATION OF
FUNCTIONS (METHODS)>);

- For example, in Figure 4(b), we declared a method Age() that calculates the age of an individual object of type PERSON_TYPE.
- The code for implementing the method still has to be written. We can refer to the method implementation by specifying the file that contains the code for the method, or we can write the actual code within the type declaration itself..

- SQL provides certain built-in functions for user-defined types. For a UDT called TYPE_T, the **constructor function** TYPE_T( ) returns a new object of that type. In the new UDT object, every attribute is initialized to its default value. An **observer function** *A* is implicitly created for each attribute *A* to read its value. Hence, *A*(*X*) or *X*.*A* returns the value of attribute *A* of TYPE_T if *X* is of type TYPE_T. A **mutator function** for updating an attribute sets the value of the attribute to a new value. SQL allows these functions to be blocked from public use; an EXECUTE privilege is needed to have access to these functions.
- In general, a UDT can have a number of user-defined functions associated with it. The syntax is

  INSTANCE METHOD <NAME> (<ARGUMENT_LIST>) RETURNS <RETURN_TYPE>;

- Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended PSM language of SQL (see Chapter 13). External func-tions are written in a host language, with only their signature (interface) appearing in the UDT definition. An external function definition can be declared as follows:

  DECLARE EXTERNAL <FUNCTION_NAME> <SIGNATURE> LANGUAGE
  <LANGUAGE_NAME>;

Attributes and functions in UDTs are divided into three categories:

PUBLIC (visible at the UDT interface)
PRIVATE (not visible at the UDT interface)
PROTECTED (visible only to subtypes)

It is also possible to define virtual attributes as part of UDTs, which are computed and updated using functions.

Specifying Inheritance and Overloading of Functions:

- SQL has rules for dealing with **type inheritance** (specified via the UNDER keyword). In general, both attributes and instance methods (operations) are inherited. The phrase NOT FINAL must be included in a UDT if subtypes are allowed to be created under that UDT (see Figure 4(a) and (b), where PERSON_TYPE, STUDENT_TYPE, and EMPLOYEE_TYPE are declared to be NOT FINAL). Associated with type inheritance are the rules for overloading of function implementations and for resolution of function names. These inheritance rules can be summarized as follows:

  1) All attributes are inherited.
  2) The order of supertypes in the UNDER clause determines the inheritance hierarchy.
  3) An instance of a subtype can be used in every context in which a supertype instance is used.
  4) A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
  5) When a function is called, the best match is selected based on the types of all arguments.
  6) For dynamic linking, the runtime types of parameters is considered.

- Consider the following examples to illustrate type inheritance, which are illustrated in Figure 4(c). Suppose that we want to create two subtypes of PERSON_TYPE: EMPLOYEE_TYPE and STUDENT_TYPE. In addition, we also create a subtype MANAGER_TYPE that inherits all the attributes (and methods) of EMPLOYEE_TYPE but has an additional attribute DEPT_MANAGED. These subtypes are shown in Figure 4(c).
- In general, we specify the local attributes and any additional specific methods for the subtype, which inherits the attributes and operations of its supertype.

- Another facility in SQL is **table inheritance** via the supertable/subtable facility. This is also specified using the keyword UNDER (see Figure 4(d)). Here, a new record that is inserted into a subtable, say the MANAGER table, is also inserted into its supertables EMPLOYEE and PERSON. Notice that when a record is inserted in MANAGER, we must provide values for all its inherited attributes. INSERT, DELETE, and UPDATE operations are appropriately propagated.

Specifying Relationships via Reference:

- A component attribute of one tuple may be a **reference** (specified using the keyword REF) to a tuple of another (or possibly the same) table. An example is shown in Figure 4(e).
- The keyword SCOPE specifies the name of the table whose tuples can be referenced by the reference attribute. Notice that this is similar to a foreign key, except that the system-generated value is used rather than the primary key value.
- SQL uses a **dot notation** to build **path expressions** that refer to the component attributes of tuples and row types. However, for an attribute whose type is REF, the dereferencing symbol –> is used. For example, the query below retrieves employees working in the company named 'ABCXYZ' by querying the EMPLOYMENT table:

```
SELECT E.Employee–>NAME
FROM EMPLOYMENT AS E
WHERE E.Company–>COMP_NAME = 'ABCXYZ';
```

- In SQL, –> is used for **dereferencing** and has the same meaning assigned to it in the C programming language. Thus, if *r* is a reference to a tuple and *a* is a component attribute in that tuple, then *r* –> *a* is the value of attribute *a* in that tuple.
- If several relations of the same type exist, SQL provides the SCOPE keyword by which a reference attribute may be made to point to a tuple within a specific table of that type.

5.8. ODMG(Object Data Management Group) OBJECT MODEL :

One of the reasons for the success of commercial relational DBMSs is the SQL standard. The lack of a standard for ODMSs for several years may have caused some potential users to shy away from converting to this new technology. Subsequently, a consortium of ODMS vendors and users, called ODMG (Object Data Management Group), proposed a standard that is known as the ODMG-93 or ODMG 1.0 standard. This was revised into ODMG 2.0, and later to ODMG 3.0. The standard is made up of several parts, including the **object model**, the **object definition language** (**ODL**), the **object query language** (**OQL**), and the **bindings** to object-oriented programming languages.

Overview of the Object Model of ODMG

The **ODMG object model** is the data model upon which the object definition language (ODL) and object query language (OQL) are based. It is meant to provide a standard data model for object databases, just as SQL describes a standard data model for relational databases. It also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts.

Objects and Literals. :

Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an object identifier and a **state** (or current value), whereas a literal has a value (state) but *no object identifier.* In either case, the value can have a complex structure. The object state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, but it does not change.

An **object** has five aspects: **identifier, name, lifetime, structure, and creation**.

The **object identifier** is a unique system-wide identifier (or Object_id). Every object must have an object identifier.

Some objects may optionally be given a unique **name** within a particular ODMS—this name can be used ~~to locate the object, and~~ the system should return the object given that name. Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object type—such as *extents*—will have a name. These names are used as **entry points** to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also have unique names, and it is possible to give *more than one* name to an object. All names within a particular ODMS must be unique.

The **lifetime** of an object specifies whether it is a *persistent object* (that is, a database object) or *transient object* (that is, an object in an executing pro-gram that disappears after the program terminates). Lifetimes are independent of types—that is, some objects of a particular type may be transient whereas others may be persistent.

The **structure** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is *atomic* or not.

An **atomic object** refers to a single object that follows a user-defined type, such as Employee or Department. If an object is not atomic, then it will be composed of other objects. For example, a *collection object* is not an atomic object, since its state will be a collection of other objects.

In the ODMG model, an atomic object is any *individual user-defined object*. All values of the basic built-in data types are considered to be *literals*.

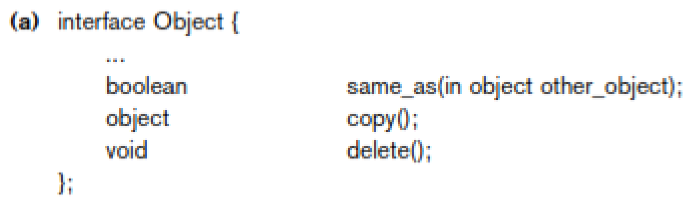
Object **creation** refers to the manner in which an object can be created. This is typically accomplished via an operation *new* for a special Object_Factory interface. We shall describe this in more detail later in this section.

In the object model, a **literal** is a value that *does not have* an object identifier. However, the value may have a simple or complex structure. There are three types of literals: atomic, structured, and collection.

Atomic literals correspond to the values of basic data types and are prede-fined. The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords long, short, unsigned long, and unsigned short in ODL), regular and double precision floating point numbers (float, double), Boolean values (boolean), single characters (char), character strings (string), and enumeration types (enum), among others.

Structured literals correspond roughly to values that are constructed using the tuple constructor. The built-in structured literals include Date, Interval, Time, and Timestamp.

Additional user-defined structured literals can be defined as needed by each application. User-defined structures are created using the STRUCT keyword in ODL, as in the C and C++ programming languages.



```
(a)  interface Object {
         ...
         boolean              same_as(in object other_object);
         object               copy();
         void                 delete();
     };
```

(b) Class Date : Object {

```
        enum                 Weekday
                                  { Sunday, Monday, Tuesday, Wednesday,
                                    Thursday, Friday, Saturday };
        enum                 Month
                                  { January, February, March, April, May, June,
                                    July, August, September, October, November,
                                    December };
        unsigned short       year();
        unsigned short       month();
        unsigned short       day();
        ...
        boolean              is_equal(in Date other_date);
        boolean              is_greater(in Date other_date);
        ...   };
```

```
Class Time : Object {
    ...
    unsigned short       hour();
    unsigned short       minute();
    unsigned short       second();
    unsigned short       millisecond();
    ...
    boolean              is_equal(in Time a_time);
    boolean              is_greater(in Time a_time);
    ...
    Time                 add_interval(in Interval an_interval);
    Time                 subtract_interval(in Interval an_interval);
    Interval             subtract_time(in Time other_time);   };
class Timestamp : Object {
    ...
    unsigned short       year();
    unsigned short       month();
    unsigned short       day();
    unsigned short       hour();
    unsigned short       minute();
    unsigned short       second();
    unsigned short       millisecond();
    ...
    Timestamp            plus(in Interval an_interval);
Timestamp           minus(in Interval an_interval);
boolean             is_equal(in Timestamp a_timestamp);
boolean             is_greater(in Timestamp a_timestamp);
...  };
class Interval :    Object {
unsigned short      day();
unsigned short      hour();
unsigned short      minute();
unsigned short      second();
unsigned short      millisecond();
...
Interval            plus(in Interval an_interval);
Interval            minus(in Interval an_interval);
Interval            product(in long a_value);
Interval            quotient(in long a_value);
boolean             is_equal(in interval an_interval);
boolean             is_greater(in interval an_interval);
...  };
```

(c) interface Collection : Object {

```
    ...
    exception           ElementNotFound{ Object element; };
    unsigned long       cardinality();
    boolean             is_empty();
    ...
    boolean             contains_element(in Object element);
    void                insert_element(in Object element);
    void                remove_element(in Object element)
                            raises(ElementNotFound);
    iterator            create_iterator(in boolean stable);
    ...  };
interface Iterator {
    exception           NoMoreElements();
    ...
    boolean             at_end();
    void                reset();
    Object              get_element() raises(NoMoreElements);
    void                next_position() raises(NoMoreElements);
    ...  };
interface set : Collection {
    set                 create_union(in set other_set);
    ...
    boolean             is_subset_of(in set other_set);
    ...  };
interface bag : Collection {
    unsigned long       occurrences_of(in Object element);
    bag                 create_union(in Bag other_bag);
    ...  };
    ...   };
interface list : Collection {
      exception              Invalid_Index{unsigned_long index; );
      void                   remove_element_at(in unsigned long index)
                                   raises(InvalidIndex);
      Object                 retrieve_element_at(in unsigned long index)
                                   raises(InvalidIndex);
      void                   replace_element_at(in Object element, in unsigned long index)
                                   raises(InvalidIndex);
      void                   insert_element_after(in Object element, in unsigned long index)
                                   raises(InvalidIndex);
      ...
      void                   insert_element_first(in Object element);
      ...
      void                   remove_first_element() raises(ElementNotFound);
      ...
      Object                 retrieve_first_element() raises(ElementNotFound);
      ...
      list                   concat(in list other_list);
      void                   append(in list other_list);
};
interface array      : Collection {
      exception              Invalid_Index{unsigned_long index; };
      exception              Invalid_Size{unsigned_long size; };
      void                   remove_element_at(in unsigned long index)
                                   raises(InvalidIndex);
      Object                 retrieve_element_at(in unsigned long index)
                                   raises(InvalidIndex);
      void                   replace_element_at(in unsigned long index, in Object element)
                                   raises(InvalidIndex);
      void                   resize(in unsigned long new_size)
                                   raises(InvalidSize);
};

struct association { Object key; Object value; };
interface dictionary : Collection {
      exception              DuplicateName{string key; };
      exception              KeyNotFound{Object key; };
      void                   bind(in Object key, in Object value)
                                   raises(DuplicateName);
      void                   unbind(in Object key) raises(KeyNotFound);
      Object                 lookup(in Object key) raises(KeyNotFound);
      boolean                contains_key(in Object key);
};
```

Collection literals specify a literal value that is a collection of objects or values but the collection itself does not have an Object_id. The collections in the object model can be defined by the *type generators* set<*T*>, bag<*T*>, list<*T*>, and array<*T*>, where *T* is the type of objects or values in the collection.[28] Another collection type is dictionary<*K*, *V*>, which is a collection of associations <*K*, *V*>, where each *K* is a key (a unique search value) associated with a value *V*; this can be used to create an index on a collection of values *V*.

Figure gives a simplified view of the basic types and type generators of the object model. The notation of ODMG uses three concepts: interface, literal, and class. Following the ODMG terminology, we use the word

behavior to refer to *operations* and **state** to refer to *properties* (attributes and relationships). An interface specifies only behavior of an object type and is typically **noninstantiable** (that is, no objects are created corresponding to an interface). Although an interface may have state properties (attributes and relationships) as part of its specifications, these *cannot* be inherited from the interface. Hence, an interface serves to define operations that can be *inherited* by other interfaces, as well as by classes that define the user-defined objects for a particular application. A class specifies both state (attributes) and behavior (operations) of an object type, and is **instantiable**. Hence, database and application objects are typically created based on the user-specified class declara-tions that form a database schema. Finally, a **literal** declaration specifies state but no behavior. Thus, a literal instance holds a simple or complex structured value but has neither an object identifier nor encapsulated operations.

Figure is a simplified version of the object model. In the object model, all objects inherit the basic inter-face operations of Object, shown in Figure (a); these include operations such as copy (creates a new copy of the object), delete (deletes the object), and same_as (compares the object's identity to another object). In general, operations are applied to objects using the **dot notation**. For example, given an object *O*, to com-pare it with another object *P*, we write

O.same_as(*P*)

The result returned by this operation is Boolean and would be true if the identity of *P* is the same as that of *O*, and false otherwise. Similarly, to create a copy *P* of object *O*, we write

P = *O*.copy()

An alternative to the dot notation is the **arrow notation:** *O*–>same_as(*P*) or *O*–>copy().

Inheritance in the Object Model of ODMG

In the ODMG object model, two types of inheritance relationships exist: behavior-only inheritance and state plus behavior inheritance. **Behavior inheritance** is also known as *ISA* or *interface inheritance*, and is specified by the colon (:) notation. Hence, in the ODMG object model, behavior inheritance requires the supertype to be an interface, whereas the subtype could be either a class or another interface.

The other inheritance relationship, called EXTENDS **inheritance**, is specified by the keyword extends. It is used to inherit both state and behavior strictly among classes, so both the supertype and the subtype must be classes. Multiple inheritance via extends is not permitted. However, multiple inheritance is allowed for behavior inheritance via the colon (:) notation. Hence, an interface may inherit behavior from several other interfaces. A class may also inherit behavior from several inter-faces via colon (:) notation, in addition to inheriting behavior and state from *at most one* other class via extends. In Section 11.3.4 we will give examples of how these two inheritance relationships—":" and extends—may be used.

Built-in Interfaces and Classes in the Object Model

Figure shows the built-in interfaces and classes of the object model. All inter-faces, such as Collection, Date, and Time, inherit the basic Object interface. In the object model, there is a distinction between collection objects, whose state contains multiple objects or literals, versus atomic (and structured) objects, whose state is an individual object or literal.

Collection objects inherit the basic Collection interface shown in Figure 11.5(c), which shows the operations for all collection objects. Given a collection object *O*, the *O*.cardinality() operation returns the number of elements in the collection. The operation *O*.is_empty() returns true if the collection *O* is empty, and returns false otherwise. The operations *O*.insert_element(*E*) and *O*.remove_element(*E*) insert or remove an element *E* from the collection *O*. Finally, the operation *O*.contains_element(*E*) returns true if the collection *O* includes element *E*, and returns false otherwise. The operation *I* = *O*.create_iterator() creates an **iterator object** *I* for the collection object *O*, which can iterate over each element in the collection. The interface for iterator objects is also shown in Figure 11.5(c). The *I*.reset() operation sets the iterator at the first element in a collection (for an unordered collection, this would be some arbitrary element), and *I*.next_position() sets the iterator to the next element. The *I*.get_element() retrieves the **current element**, which is the element at which the iterator is currently positioned.

The ODMG object model uses **exceptions** for reporting errors or particular condi-tions. For example, the ElementNotFound exception in the Collection interface would be raised by the *O*.remove_element(*E*) operation if *E* is not an element in the collection *O*. The NoMoreElements exception in the iterator interface would be raised by the *I*.next_position() operation if the iterator is currently positioned at the last element in the collection, and hence no more elements exist for the iterator to point to.

Collection objects are further specialized into set, list, bag, array, and dictionary, which inherit the operations of the Collection interface. A set<*T*> type generator can be used to create objects such that the value of object *O* is a *set whose elements are of type T*. The Set interface includes the additional operation *P* = *O*.create_union(*S*) (see Figure 11.5(c)), which returns a new object *P* of type set<*T*> that is the union of the two sets *O* and *S*. Other operations similar to create_union (not shown in Figure 11.5(c)) are create_intersection(*S*) and create_difference(*S*). Operations for set com-parison include the *O*.is_subset_of(*S*) operation, which returns true if the set object *O* is a subset of some other set object *S*, and returns false otherwise. Similar opera-tions (not shown in Figure 11.5(c)) are is_proper_subset_of(*S*), is_superset_of(*S*), and is_proper_superset_of(*S*). The bag<*T*> type generator allows duplicate elements in the collection and also inherits the Collection interface. It has three operations— create_union(b), create_intersection(b), and create_difference(b)—that all return a new object of type bag<*T*>.

A list<*T*> object type inherits the Collection operations and can be used to create col-lections where the order of the elements is important. The value of each such object *O* is an *ordered list whose elements are of type T*. Hence, we can refer to the first, last, and *i*th element in the list. Also, when we add an element to the list, we must specify the position in the list where the element is inserted. Some of the list operations are shown in Figure 11.5(c). If *O* is an object of type list<*T*>, the operation *O*.insert_element_first(*E*) inserts the element *E* before the first element in the list *O*, so that *E* becomes the first element in the list. A similar operation (not shown) is *O*.insert_element_last(*E*). The operation *O*.insert_element_after(*E*, *I*) in Figure 11.5(c) inserts the element *E* after the *i*th element in the list *O* and will raise the exception InvalidIndex if no *i*th element exists in *O*. A similar operation (not shown) is *O*.insert_element_before(*E*, *I*). To remove elements from the list, the operations are *E*

O.remove_first_element(), *E* = *O*.remove_last_element(), and *E* = *O*.remove_element _at(*I*); these operations remove the indicated element from the list *and* return the

element as the operation's result. Other operations retrieve an element without removing it from the list. These are *E* = *O*.retrieve_first_element(), *E* = *O*.retrieve _last_element(), and *E* = *O*.retrieve_element_at(*I*). Also, two operations to manipulate lists are defined. They are *P* = *O*.concat(*I*), which creates a new list *P* that is the con-catenation of lists *O* and *I* (the elements in list *O* followed by those in list *I*), and *O*.append(*I*), which appends the elements of list *I* to the end of list *O* (without creat-ing a new list object).

The array<*T*> object type also inherits the Collection operations, and is similar to list. Specific operations for an array object *O* are *O*.replace_element_at(*I*, *E*), which replaces the array element at position *I* with element *E*; *E* = *O*.remove_element_at(*I*), which retrieves the *i*th element and replaces it with a NULL value; and

E = *O*.retrieve_element_at(*I*), which simply retrieves the *i*th element of the array. Any of these operations can raise the exception InvalidIndex if *I* is greater than the array's size. The operation *O*.resize(*N*) changes the number of array elements to *N*.

The last type of collection objects are of type dictionary<*K*,*V*>. This allows the cre-ation of a collection of association pairs <*K*,*V*>, where all *K* (key) values are unique. This allows for associative retrieval of a particular pair given its key value (similar to an index). If *O* is a collection object of type dictionary<*K*,*V*>, then *O*.bind(*K*,*V*) binds value *V* to the key *K* as an association <*K*,*V*> in the collection, whereas *O*.unbind(*K*) removes the association with key *K* from *O*, and *V* = *O*.lookup(*K*) returns the value *V* associated with key *K* in *O*. The latter two operations can raise the exception KeyNotFound. Finally, *O*.contains_key(*K*) returns true if key *K* exists in *O*, and returns false otherwise.

Figure 11.6 is a diagram that illustrates the inheritance hierarchy of the built-in con-structs of the object model. Operations are inherited from the supertype to the sub-type. The collection interfaces described above are *not directly instantiable*; that is, one cannot directly create objects based on these interfaces. Rather, the interfaces can be used to generate user-defined collection types—of type set, bag, list, array, or dictionary—for a particular database application. If an attribute or class has a collec-tion type, say a set, then it will inherit the operations of the set interface. For exam-ple, in a UNIVERSITY database application, the user can specify a type for set<STUDENT>, whose state would be sets of STUDENT objects. The programmer can then use the operations for set<*T*> to manipulate an instance of type set<STUDENT>. Creating application classes is typically done by utilizing the object definition language ODL.

It is important to note that all objects in a particular collection *must be of the same type*. Hence, although the keyword any appears in the specifications of collection interfaces in Figure (c), this does not mean that objects of any type can be inter-mixed within the same collection. Rather, it means that any type can be used when specifying the type of elements for a particular collection (including other collection types!).

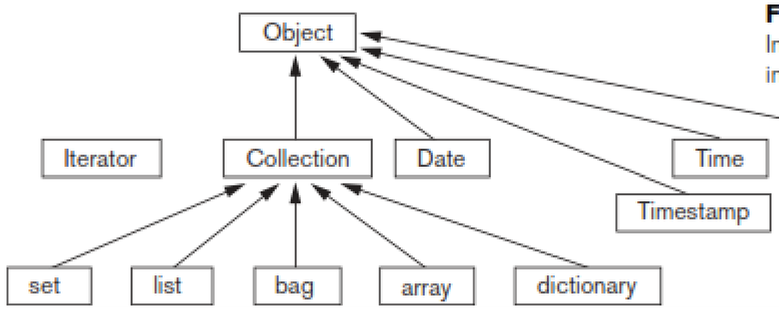


Figure 11.6
Inheritance hierarchy for the built-in interfaces of the object model.

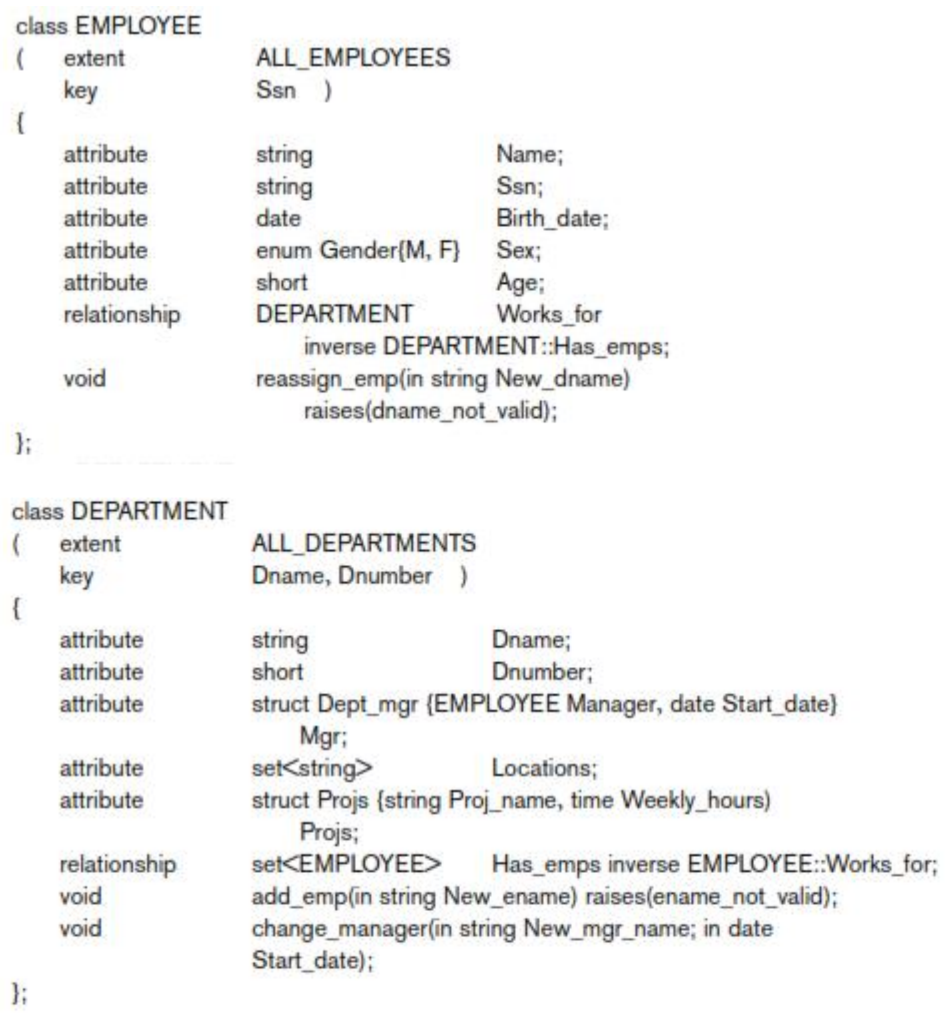
<u>Atomic (User-Defined) Objects</u>

The previous section described the built-in collection types of the object model. Now we discuss how object types for *atomic objects* can be constructed. These are specified using the keyword class in ODL. In the object model, any user-defined object that is not a collection object is called an **atomic object**.

For example, in a UNIVERSITY database application, the user can specify an object type (class) for STUDENT objects. Most such objects will be **structured objects**; for example, a STUDENT object will have a complex structure, with many attributes, relationships, and operations, but it is still considered atomic because it is not a collection. Such a user-defined atomic object type is defined as a class by specifying its **properties** and **operations**. The properties define the state of the object and are further distinguished into **attributes** and **relationships**. In this subsection, we elaborate on the three types of components—attributes, relationships, and operations—that a user-defined object type for atomic (structured) objects can include. We illustrate our discussion with the two classes EMPLOYEE and DEPARTMENT shown in Figure 11.7.

An attribute is a property that describes some aspect of an object. Attributes have values (which are typically literals having a simple or complex structure) that are stored within the object. However, attribute values can also be Object_ids of other objects. Attribute values can even be specified via methods that are used to calculate the attribute value. In Figure the attributes for EMPLOYEE are Name, Ssn, Birth_date, Sex, and Age, and those for DEPARTMENT are Dname, Dnumber, Mgr, Locations, and Projs. The Mgr and Projs attributes of DEPARTMENT have complex structure and are defined via struct, which corresponds to the *tuple constructor*. Hence, the value of Mgr in each DEPARTMENT object will have two components: Manager, whose value is an Object_id that references the EMPLOYEE object that manages the DEPARTMENT, and Start_date, whose value is a date. The locations attribute of DEPARTMENT is defined via the set constructor, since each DEPARTMENT object can have a set of locations.

A relationship is a property that specifies that two objects in the database are related. In the object model of ODMG, only binary relationships are explicitly represented, and each binary relationship is represented by a *pair of inverse references* specified via the keyword relationship. In Figure, one relationship exists that relates each EMPLOYEE to the DEPARTMENT in which he or she works— the Works_for relationship of EMPLOYEE. In the inverse direction, each DEPARTMENT is related to the set of EMPLOYEES that work in the DEPARTMENT— the Has_emps relationship of DEPARTMENT. The keyword inverse specifies that these two properties define a single conceptual relationship in inverse directions.



```
class EMPLOYEE
(     extent            ALL_EMPLOYEES
      key               Ssn    )
{
      attribute         string              Name;
      attribute         string              Ssn;
      attribute         date                Birth_date;
      attribute         enum Gender{M, F}   Sex;
      attribute         short               Age;
      relationship      DEPARTMENT          Works_for
                            inverse DEPARTMENT::Has_emps;
      void              reassign_emp(in string New_dname)
                            raises(dname_not_valid);
};

class DEPARTMENT
(     extent            ALL_DEPARTMENTS
      key               Dname, Dnumber    )
{
      attribute         string              Dname;
      attribute         short               Dnumber;
      attribute         struct Dept_mgr {EMPLOYEE Manager, date Start_date}
                            Mgr;
      attribute         set<string>         Locations;
      attribute         struct Projs {string Proj_name, time Weekly_hours)
                            Projs;
      relationship      set<EMPLOYEE>       Has_emps inverse EMPLOYEE::Works_for;
      void              add_emp(in string New_ename) raises(ename_not_valid);
      void              change_manager(in string New_mgr_name; in date
                        Start_date);
};
```

Figure . The attributes, relationships, and operations in a class definition.

By specifying inverses, the database system can maintain the referential integrity of the relationship automatically. That is, if the value of Works_for for a particular EMPLOYEE *E* refers to DEPARTMENT *D*, then the value of Has_emps for DEPARTMENT *D* must include a reference to *E* in its set of EMPLOYEE references. If the database designer desires to have a relationship to be represented in *only one direction*, then it has to be modeled as an attribute (or operation). An example is the Manager component of the Mgr attribute in DEPARTMENT.

In addition to attributes and relationships, the designer can include **operations** in object type (class) specifications. Each object type can have a number of **operation signatures**, which specify the operation name, its argument types, and its returned value, if applicable. Operation names are unique within each object type, but they can be overloaded by having the same operation name appear in distinct object types. The operation signature can also specify the names of **exceptions** that can

occur during operation execution. The implementation of the operation will include the code to raise these exceptions. In Figure the EMPLOYEE class has one operation: reassign_emp, and the DEPARTMENT class has two operations: add_emp and change_manager.

Extents, Keys, and Factory Objects

In the ODMG object model, the database designer can declare an *extent* (using the keyword extent) for any object type that is defined via a class declaration. The extent is given a name, and it will contain all persistent objects of that class. Hence, the extent behaves as a *set object* that holds all persistent objects of the class. In Figure the EMPLOYEE and DEPARTMENT classes have extents called ALL_EMPLOYEES and ALL_DEPARTMENTS, respectively. This is similar to creating two objects—one of type set<EMPLOYEE> and the second of type set<DEPARTMENT>—and making them persistent by naming them ALL_EMPLOYEES and ALL_DEPARTMENTS. Extents are also used to automatically enforce the set/subset relationship between the extents of a supertype and its subtype. If two classes A and B have extents ALL_A and ALL_B, and class B is a subtype of class A (that is, class B extends class A), then the collection of objects in ALL_B must be a subset of those in ALL_A at any point. This constraint is automatically enforced by the database system.

A class with an extent can have one or more keys. A **key** consists of one or more properties (attributes or relationships) whose values are constrained to be unique for each object in the extent. For example, in Figure the EMPLOYEE class has the Ssn attribute as key (each EMPLOYEE object in the extent must have a unique Ssn value), and the DEPARTMENT class has two distinct keys: Dname and Dnumber (each DEPARTMENT must have a unique Dname and a unique Dnumber). For a composite key that is made of several properties, the properties that form the key are contained in parentheses. For example, if a class VEHICLE with an extent ALL_VEHICLES has a key made up of a combination of two attributes State and License_number, they would be placed in parentheses as (State, License_number) in the key declaration.

Next, we present the concept of **factory object**—an object that can be used to gen-erate or create individual objects via its operations. Some of the interfaces of factory objects that are part of the ODMG object model are shown in Figure. The interface ObjectFactory has a single operation, new(), which returns a new object with an Object_id. By inheriting this interface, users can create their own factory interfaces for each user-defined (atomic) object type, and the programmer can implement the operation *new* differently for each type of object. Figure also shows a DateFactory interface, which has additional operations for creating a new calendar_date, and for creating an object whose value is the current_date, among other operations. As we can see, a factory object basically provides the **constructor operations** for new objects.

```
interface ObjectFactory {
      Object          new();
};

interface SetFactory : ObjectFactory {
      Set             new_of_size(in long size);
};

interface ListFactory : ObjectFactory {
      List            new_of_size(in long size);
};

interface ArrayFactory : ObjectFactory {
      Array           new_of_size(in long size);
};

interface DictionaryFactory : ObjectFactory {
      Dictionary      new_of_size(in long size);
};
```

```
interface DictionaryFactory : ObjectFactory {
      Dictionary      new_of_size(in long size);
};
```

```
interface DateFactory : ObjectFactory {
      exception       InvalidDate{};
      ...
      Date            calendar_date(    in unsigned short year,
                                        in unsigned short month,
                                        in unsigned short day      )
                      raises(InvalidDate);
      ...
      Date            current();
};

interface DatabaseFactory {
      Database        new();
};

interface Database {
      ...
      void          open(in string database_name)
                          raises(DatabaseNotFound, DatabaseOpen);
      void          close() raises(DatabaseClosed, ...);
      void          bind(in Object an_object, in string name)
                          raises(DatabaseClosed, ObjectNameNotUnique, ...);
      Object        unbind(in string name)
                          raises(DatabaseClosed, ObjectNameNotFound, ...);
      Object        lookup(in string object_name)
                          raises(DatabaseClosed, ObjectNameNotFound, ...);
      ... };
```

Figure. Interfaces to illustrate factory objects and database objects.

Finally, we discuss the concept of a **database**. Because an ODBMS can create many different databases, each with its own schema, the ODMG object model has inter-faces for DatabaseFactory and Database objects, as shown in Figure. Each data-base has its own *database name,* and the bind operation can be used to assign individual unique names to persistent objects in a particular database. The lookup operation returns an object from the database that has the specified object_name, and the unbind operation removes the name of a persistent named object from the database.

5.9. THE OBJECT DEFINITION LANGUAGE (ODL)

The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any particular programming language. Its main use is to create object specifications—that is, classes and interfaces. Hence, ODL is not a full programming language. A user can specify a database schema in ODL independently of any programming language, and then use the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming languages, such as C++, Smalltalk, and Java.

Figure (b) shows a possible object schema for part of the UNIVERSITY database. The graphical notation for Figure(b) is shown in Figure (a) and can be considered as a variation of EER diagrams with the added concept of interface inheritance but without several EER concepts, such as categories (union types) and attributes of relationships.

Figure shows one possible set of ODL class definitions for the UNIVERSITY database. In general, there may be several possible mappings from an object schema diagram (or EER schema diagram) into ODL classes.

Figure shows the straightforward way of mapping part of the UNIVERSITY database. Entity types are mapped into ODL classes, and inheritance is done using extends. However, there is no direct way to map categories (union types) or to do multiple inheritance. In Figure the classes PERSON, FACULTY, STUDENT, and GRAD_STUDENT have the extents PERSONS, FACULTY, STUDENTS, and GRAD_STUDENTS, respectively. Both FACULTY and STUDENT extends PERSON and GRAD_STUDENT extends STUDENT. Hence, the collection of STUDENTS (and the collection of FACULTY) will be constrained to be a subset of the collection of PERSONs at any time. Similarly, the collection of GRAD_STUDENTs will be a subset of STUDENTs. At the same time, individual STUDENT and FACULTY objects will inherit the properties (attributes and relationships) and operations of PERSON, and individual GRAD_STUDENT objects will inherit those of STUDENT.

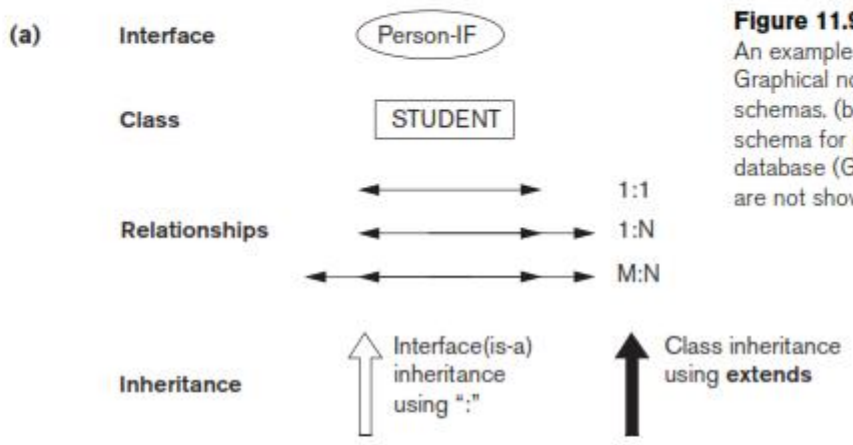
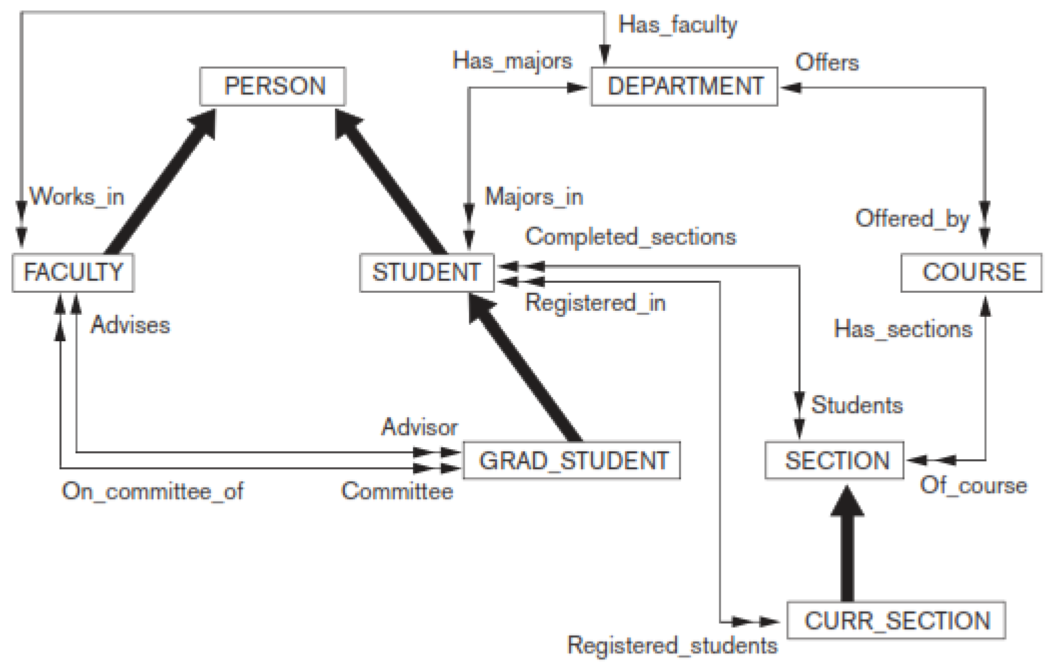

(a)
Interface
Person-IF
Class
STUDENT
Relationships
1:1
1:N
M:N
Inheritance
Interface(is-a) inheritance using ":"
Class inheritance using **extends**

Figure 11.9
An example of a database schema. (a) Graphical notation for representing ODL schemas. (b) A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).

(b)



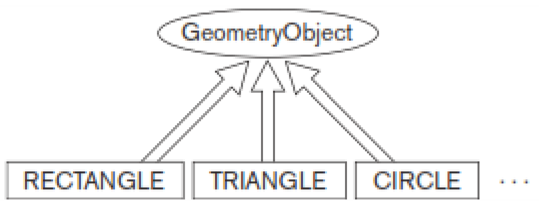
```
class PERSON
(     extent        PERSONS
      key           Ssn  )
{     attribute     struct Pname {  string    Fname,
                                    string    Mname,
                                    string    Lname  }      Name;
      attribute     string                                  Ssn;
      attribute     date                                    Birth_date;
      attribute     enum Gender{M, F}                       Sex;
      attribute     struct Address {  short   No,
                                      string  Street,
                                      short   Apt_no,
                                      string  City,
                                      string  State,
                                      short   Zip  }        Address;
      short         Age();    };
```

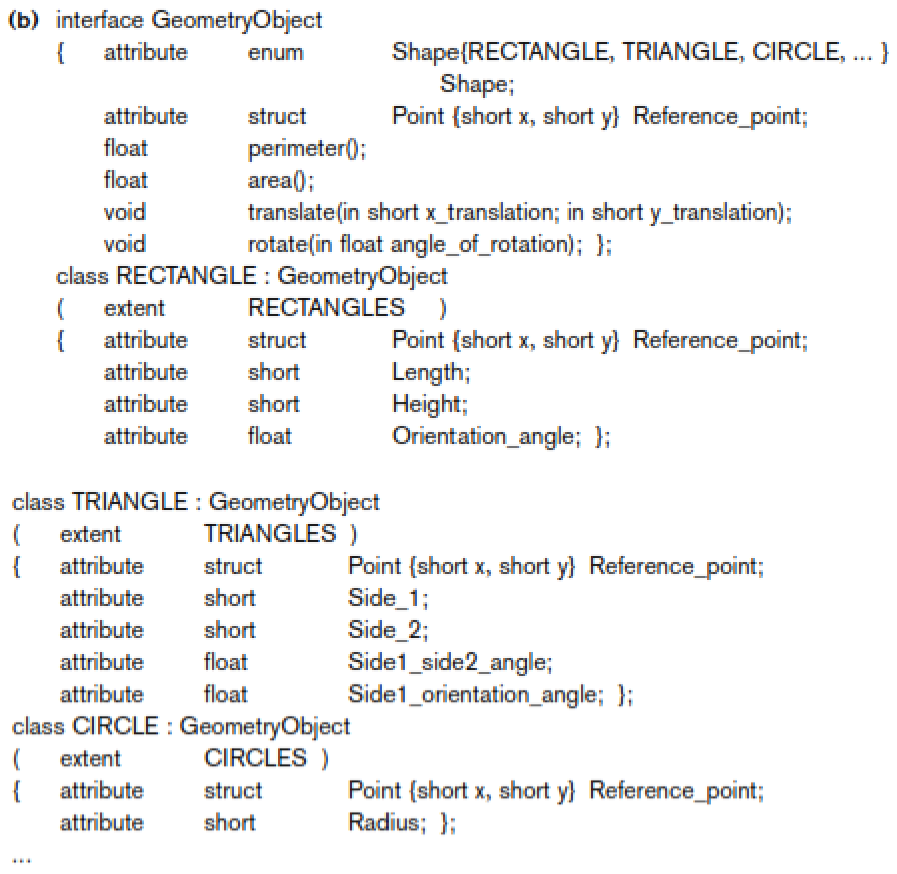
```
class FACULTY extends PERSON
(     extent          FACULTY  )
{     attribute       string              Rank;
      attribute       float               Salary;
      attribute       string              Office;
      attribute       string              Phone;
      relationship    DEPARTMENT     Works_in inverse DEPARTMENT::Has faculty;
      relationship    set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
      relationship    set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
      void            give_raise(in float raise);
      void            promote(in string new rank);  };
class GRADE
(     extent          GRADES  )
{
      attribute       enum GradeValues{A,B,C,D,F,I, P} Grade;
      relationship    SECTION Section inverse SECTION::Students;
      relationship STUDENT Student inverse STUDENT::Completed_sections;  };

class STUDENT extends PERSON
(     extent          STUDENTS  )
{     attribute       string              Class;
      attribute       DEPARTMENT          Minors_in;
      relationship    DEPARTMENT Majors_in inverse DEPARTMENT::Has_majors;
      relationship    set<GRADE> Completed_sections inverse GRADE::Student;
      relationship    set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
      void            change_major(in string dname) raises(dname_not_valid);
      float           gpa();
      void            register(in short secno) raises(section_not_valid);
      void            assign_grade(in short secno; IN GradeValue grade)
                          raises(section_not_valid,grade_not_valid);  };


class DEGREE
{     attribute       string              College;
      attribute       string              Degree;
      attribute       string              Year;      };
class GRAD_STUDENT extends STUDENT
(     extent          GRAD_STUDENTS  )
{     attribute       set<DEGREE>         Degrees;
      relationship    FACULTY Advisor inverse FACULTY::Advises;
      relationship    set<FACULTY>    Committee inverse FACULTY::On_committee_of;
      void            assign_advisor(in string Lname; in string Fname)
                          raises(faculty_not_valid);
      void            assign_committee_member(in string Lname; in string Fname)
                          raises(faculty_not_valid);  };
```

```
class DEPARTMENT
(     extent          DEPARTMENTS
      key             Dname )
{     attribute       string            Dname;
      attribute       string            Dphone;
      attribute       string            Doffice;
      attribute       string            College;
      attribute       FACULTY           Chair;
      relationship    set<FACULTY> Has_faculty inverse FACULTY::Works_in;
      relationship    set<STUDENT> Has_majors inverse STUDENT::Majors_in;
      relationship    set<COURSE> Offers inverse COURSE::Offered_by;  };
class COURSE
(     extent          COURSES
      key             Cno )
{     attribute       string            Cname;
      attribute       string            Cno;
      attribute       string            Description;
      relationship    set<SECTION> Has_sections inverse SECTION::Of_course;
      relationship    <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };


class SECTION
(     extent          SECTIONS )
{     attribute       short             Sec_no;
      attribute       string            Year;
      attribute       enum Quarter{Fall, Winter, Spring, Summer}
                          Qtr;
      relationship    set<GRADE> Students inverse GRADE::Section;
      relationship    course Of_course inverse COURSE::Has_sections;  };
class CURR_SECTION extends SECTION
(     extent          CURRENT_SECTIONS )
{     relationship    set<STUDENT> Registered_students
                          inverse STUDENT::Registered_in
      void            register_student(in string Ssn)
                          raises(student_not_valid, section_full);  };
```

Figure . Possible ODL schema for the UNIVERSITY database

(a)





```
(b) interface GeometryObject
    {   attribute   enum        Shape{RECTANGLE, TRIANGLE, CIRCLE, ... }
                                    Shape;
        attribute   struct      Point {short x, short y}  Reference_point;
        float       perimeter();
        float       area();
        void        translate(in short x_translation; in short y_translation);
        void        rotate(in float angle_of_rotation);  };
    class RECTANGLE : GeometryObject
    (   extent      RECTANGLES    )
    {   attribute   struct      Point {short x, short y}  Reference_point;
        attribute   short       Length;
        attribute   short       Height;
        attribute   float       Orientation_angle;  };

    class TRIANGLE : GeometryObject
    (   extent      TRIANGLES  )
    {   attribute   struct      Point {short x, short y}  Reference_point;
        attribute   short       Side_1;
        attribute   short       Side_2;
        attribute   float       Side1_side2_angle;
        attribute   float       Side1_orientation_angle;  };
    class CIRCLE : GeometryObject
    (   extent      CIRCLES  )
    {   attribute   struct      Point {short x, short y}  Reference_point;
        attribute   short       Radius;  };
    ...
```

Figure. An illustration of interface inheritance via ":".
(a) Graphical schema representation,
(b) Corresponding interface and class definitions in ODL.

The classes DEPARTMENT, COURSE, SECTION, and CURR_SECTION in Figure are straightforward mappings of the corresponding entity types in figure(b). However, the class GRADE requires some explanation. The GRADE class corresponds to the M:N relationship between STUDENT and SECTION in Figure (b). The reason it was made into a separate class (rather than as a pair of inverse relationships) is because it includes the relationship attribute Grade.

Hence, the M:N relationship is mapped to the class GRADE, and a pair of 1:N relationships, one between STUDENT and GRADE and the other between SECTION and GRADE. These relationships are represented by the following relationship proper-ties: Completed_sections of STUDENT; Section and Student of GRADE; and Students of SECTION. Finally, the class DEGREE is used to represent the composite, multivalued attribute degrees of GRAD_STUDENT.

Because the previous example does not include any interfaces, only classes, we now utilize a different example to illustrate interfaces and interface (behavior) inheritance. Figure(a) is part of a database schema for storing geometric objects. An interface GeometryObject is specified, with operations to calculate the perimeter and area of a geometric object, plus operations to translate (move) and rotate an object. Several classes (RECTANGLE, TRIANGLE, CIRCLE, ...) inherit the GeometryObject interface. Since GeometryObject is an interface, it is *noninstantiable*—that is, no objects can be created based on this interface directly. However, objects of type RECTANGLE, TRIANGLE, CIRCLE, ... can be created, and these objects inherit all the operations of the GeometryObject interface. Note that with interface inheritance, only operations are inherited, not properties (attributes, relationships). Hence, if a property is needed in the inheriting class, it must be repeated in the class definition, as with the Reference_point attribute in Figure(b). Notice that the inherited operations can have different implementations in each class. For example, the implementations of the area and perimeter operations may be different for RECTANGLE, TRIANGLE, and CIRCLE.

Multiple inheritance of interfaces by a class is allowed, as is multiple inheritance of interfaces by another interface. However, with the extends (class) inheritance, multiple inheritance is *not permitted.* Hence, a class can inherit via extends from at most one class (in addition to inheriting from zero or more interfaces).

5.10. OBJECT QUERY LANGUAGE(OQL)

The object query language OQL is the query language proposed for the ODMG object model. It is designed to work closely with the programming languages for which an ODMG binding is defined, such as C++, Smalltalk, and Java. Hence, an OQL query embedded into one of these programming languages can return objects that match the type system of that language. Additionally, the implementations of class operations in an ODMG schema can have their code written in these program-ming languages. The OQL syntax for queries is similar to the syntax of the relational standard query language SQL, with additional features for ODMG concepts, such as object identity, complex objects, operations, inheritance, polymorphism, and relationships.

Simple OQL Queries, Database Entry Points, and Iterator Variables

The basic OQL syntax is a select ... from ... where ... structure, as it is for SQL. For example, the query to retrieve the names of all departments in the college of ‘Engineering’ can be written as follows:

```
Q0: select  D.Dname
    from    D in DEPARTMENTS
    where   D.College = ‘Engineering’;
```

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object.* For many queries, the entry point is the name of the extent of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. Looking at the extent names in Figure, the named object DEPARTMENTS is of type set<DEPARTMENT>; PERSONS is of type set<PERSON>; FACULTY is of type set<FACULTY>; and so on.

The use of an extent name—DEPARTMENTS in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable**—*D* in Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the where clause. In Q0, only persistent objects *D* in the collection of DEPARTMENTS that satisfy the condi-tion *D*.College = ‘Engineering’ are selected for the query result. For each selected object *D*, the value of *D*.Dname is retrieved in the query result. Hence, the *type of the result* for Q0 is bag<string> because the type of each Dname value is string (even though the actual result is a set because Dname is a key attribute). In general, the result of a query would be of type bag for select ... from ... and of type set for select distinct ... from ... , as in SQL (adding the keyword distinct eliminates duplicates).

Using the example in Q0, there are three syntactic options for specifying iterator variables:

in DEPARTMENTS DEPARTMENTS *D* DEPARTMENTS AS *D*

We will use the first construct in our examples.

The named objects used as database entry points for OQL queries are not limited to the names of extents. Any named persistent object, whether it refers to an atomic (single) object or to a collection object, can be used as a database entry point.

Query Results and Path Expressions

In general, the result of a query can be of any type that can be expressed in the ODMG object model. A query does not have to follow the select ... from ... where ...

structure; in the simplest case, any persistent name on its own is a query, whose result is a reference to that persistent object. For example, the query

Q1: DEPARTMENTS;

returns a reference to the collection of all persistent DEPARTMENT objects, whose type is set<DEPARTMENT>. Similarly, suppose we had given a persistent name CS_DEPARTMENT to a single DEPARTMENT object (the Computer Science department); then, the query

Q1A: CS_DEPARTMENT;

returns a reference to that individual object of type DEPARTMENT. Once an entry point is specified, the concept of a **path expression** can be used to specify a *path* to related attributes and objects. A path expression typically starts at a *persistent object name*, or at the iterator variable that ranges over individual objects in a collection. This name will be followed by zero or more relationship names or attribute names connected using the *dot notation.* For example, referring to the UNIVERSITY data-base in Figure, the following are examples of path expressions, which are also valid queries in OQL:

Q2: CS_DEPARTMENT.Chair;
Q2A: CS_DEPARTMENT.Chair.Rank;
Q2B: CS_DEPARTMENT.Has_faculty;

The first expression Q2 returns an object of type FACULTY, because that is the type of the attribute Chair of the DEPARTMENT class. This will be a reference to the FACULTY object that is related to the DEPARTMENT object whose persistent name is CS_DEPARTMENT via the attribute Chair; that is, a reference to the FACULTY object who is chairperson of the Computer Science department. The second expression Q2A is similar, except that it returns the Rank of this FACULTY object (the Computer Science chair) rather than the object reference; hence, the type returned by Q2A is string, which is the data type for the Rank attribute of the FACULTY class.

Path expressions Q2 and Q2A return single values, because the attributes Chair (of DEPARTMENT) and Rank (of FACULTY) are both single-valued and they are applied to a single object. The third expression, Q2B, is different; it returns an object of type set<FACULTY> even when applied to a single object, because that is the type of the relationship Has_faculty of the DEPARTMENT class. The collection returned will include references to all FACULTY objects that are related to the DEPARTMENT object whose persistent name is CS_DEPARTMENT via the relationship Has_faculty; that is, references to all FACULTY objects who are working in the Computer Science depart-ment. Now, to return the ranks of Computer Science faculty, we *cannot* write

Q3 : CS_DEPARTMENT.Has_faculty.Rank;

because it is not clear whether the object returned would be of type set<string> or bag<string> (the latter being more likely, since multiple faculty may share the same rank). Because of this type of ambiguity problem, OQL does not allow expressions such as Q3 . Rather, one must use an iterator variable over any collections, as in Q3A or Q3B below:

```
Q3A:  select  F.Rank
      from    F in
```

```
                CS_DEPARTMENT.Has_faculty;

Q3B:    select  distinct F.Rank
                F in
        from    CS_DEPARTMENT.Has_faculty;
```

Here, Q3A returns bag<string> (duplicate rank values appear in the result), whereas Q3B returns set<string> (duplicates are eliminated via the distinct keyword). Both Q3A and Q3B illustrate how an iterator variable can be defined in the from clause to range over a restricted collection specified in the query. The variable *F* in Q3A and Q3B ranges over the elements of the collection CS_DEPARTMENT.Has_faculty, which is of type set<FACULTY>, and includes only those faculty who are members of the Computer Science department.

In general, an OQL query can return a result with a complex structure specified in the query itself by utilizing the struct keyword. Consider the following examples:

```
Q4:     CS_DEPARTMENT.Chair.Advises;

  Q4A:      select struct ( name: struct (last_name: S.name.Lname, first_name:
             S.name.Fname), degrees:( select struct (deg: D.Degree, yr: D.Year,
             college: D.College) from D in S.Degrees )) from S in
            CS_DEPARTMENT.Chair.Advises;
```

Here, Q4 is straightforward, returning an object of type set<GRAD_STUDENT> as its result; this is the collection of graduate students who are advised by the chair of the Computer Science department. Now, suppose that a query is needed to retrieve the last and first names of these graduate students, plus the list of previous degrees of each. This can be written as in Q4A, where the variable *S* ranges over the collection of graduate students advised by the chairperson, and the variable *D* ranges over the degrees of each such student *S*. The type of the result of Q4A is a collection of (first-level) structs where each struct has two components: name and degrees.

The name component is a further struct made up of last_name and first_name, each being a single string. The degrees component is defined by an embedded query and is itself a collection of further (second level) structs, each with three string components: deg, yr, and college.

Note that OQL is *orthogonal* with respect to specifying path expressions. That is, attributes, relationships, and operation names (methods) can be used interchange-ably within the path expressions, as long as the type system of OQL is not compromised. For example, one can write the following queries to retrieve the grade point average of all senior students majoring in Computer Science, with the result ordered by GPA, and within that by last and first name:

```
        select struct ( last_name: S.name.Lname, first_name:
Q5A:    S.name.Fname,
                  gpa: S.gpa  )
        from      S in CS_DEPARTMENT.Has_majors
        where     S.Class = 'senior'
        order by  gpa desc, last_name asc, first_name asc;
        select struct ( last_name: S.name.Lname, first_name:
Q5B:    S.name.Fname,
                  gpa: S.gpa  )
        from      S in STUDENTS
        where     S.Majors_in.Dname = 'Computer Science' and
                  S.Class = 'senior'
        order by  gpa desc, last_name asc, first_name asc;
```

Q5A used the named entry point CS_DEPARTMENT to directly locate the reference to the Computer Science department and then locate the students via the relation-ship Has_majors, whereas Q5B searches the STUDENTS extent to locate all students majoring in that department. Notice how attribute names, relationship names, and operation (method) names are all used interchangeably (in an orthogonal manner) in the path expressions: gpa is an operation; Majors_in and Has_majors are relation-ships; and Class, Name, Dname, Lname, and Fname are attributes. The implementa-tion of the gpa operation computes the grade point average and returns its value as a float type for each selected STUDENT.

The order by clause is similar to the corresponding SQL construct, and specifies in which order the query result is to be displayed. Hence, the collection returned by a query with an order by clause is of type *list.*

Other Features of OQL:

Specifying Views as Named Queries. The view mechanism in OQL uses the concept of a **named query**. The define keyword is used to specify an identifier of the named query, which must be a unique name among all named objects, class names, method names, and function names in the schema. If the identifier has the same name as an existing named query, then the new definition replaces the previous definition. Once defined, a query definition is persistent until it is redefined or deleted. A view can also have parameters (arguments) in its definition.

For example, the following view V1 defines a named query Has_minors to retrieve the set of objects for students minoring in a given department:

```
V1: define  Has_minors(Dept_name) as
    select  S
    from    S in STUDENTS
    where   S.Minors_in.Dname = Dept_name;
```

Because the ODL schema in Figure 11.10 only provided a unidirectional Minors_in attribute for a STUDENT, we can use the above view to represent its inverse without having to explicitly define a relationship. This type of view can be used to represent inverse relationships that are not expected to be used frequently. The user can now utilize the above view to write queries such as

```
Has_minors('Computer Science');
```

which would return a bag of students minoring in the Computer Science department. Note that in Figure, we defined Has_majors as an explicit relationship, presumably because it is expected to be used more often.

Extracting Single Elements from Singleton Collections. An OQL query will, in general, return a collection as its result, such as a bag, set (if distinct is specified), or list (if the order by clause is used). If the user requires that a query only return a sin-gle element, there is an element operator in OQL that is guaranteed to return a single element *E* from a singleton collection *C* that contains only one element. If *C* contains more than one element or if *C* is empty, then the element operator *raises an exception.* For example, Q6 returns the single object reference to the Computer Science department:

```
Q6: element  ( select D
               from  D in DEPARTMENTS
               where D.Dname = 'Computer Science' );
```

Since a department name is unique across all departments, the result should be one department. The type of the result is *D*:DEPARTMENT.

Collection Operators (Aggregate Functions, Quantifiers). Because many query expressions specify collections as their result, a number of operators have been defined that are applied to such collections. These include aggregate operators as well as membership and quantification (universal and existential) over a collection.

The aggregate operators (min, max, count, sum, avg) operate over a collection. The operator count returns an integer type. The remaining aggregate operators (min, max, sum, avg) return the same type as the type of the operand collection. Two examples follow. The query Q7 returns the number of students minoring in Computer Science and Q8 returns the average GPA of all seniors majoring in Computer Science.

```
Q7: count ( S in Has_minors('Computer Science'));
Q8: avg ( select S.Gpa
          from   S in STUDENTS
          where  S.Majors_in.Dname = 'Computer Science' and
                 S.Class = 'Senior');
```

Notice that aggregate operations can be applied to any collection of the appropriate type and can be used in any part of a query. For example, the query to retrieve all department names that have more than 100 majors can be written as in Q9:

```
Q9: select  D.Dname
    from    D in DEPARTMENTS
    where   count (D.Has_majors) > 100;
```

The *membership* and *quantification* expressions return a Boolean type—that is, true or false. Let *V* be a variable, *C* a collection expression, *B* an expression of type Boolean (that is, a Boolean condition), and *E* an element of the type of elements in collection *C*. Then:

(*E* in *C*) returns true if element *E* is a member of collection *C*.
(for all *V* in *C* : *B*) returns true if *all* the elements of collection *C* satisfy *B*.
(exists *V* in *C* : *B*) returns true if there is at least one element in *C* satisfying *B*.

To illustrate the membership condition, suppose we want to retrieve the names of all students who completed the course called 'Database Systems I'. This can be writ-ten as in Q10, where the nested query returns the collection of course names that each STUDENT *S* has completed, and the membership condition returns true if 'Database Systems I' is in the collection for a particular STUDENT *S*:

```
Q10:  select S.name.Lname, S.name.Fname
                        From S in STUDENTS
                        Where 'Database Systems I' in
                      ( select C.Section.Of_course.Cname
                       From C in S.Completed_sections);
```

Q10 also illustrates a simpler way to specify the select clause of queries that return a collection of structs; the type returned by Q10 is bag<struct(string, string)>.

One can also write queries that return true/false results. As an example, let us assume that there is a named object called JEREMY of type STUDENT. Then, query Q11 answers the following question: *Is Jeremy a Computer Science minor?* Similarly, Q12 answers the question *Are all Computer Science graduate students advised by Computer Science faculty?* Both Q11 and Q12 return true or false, which are inter-preted as yes or no answers to the above questions:

```
Q11: JEREMY in Has_minors('Computer Science');
Q12: for all G in
            ( select   S
              from     S in GRAD_STUDENTS
              where    S.Majors_in.Dname = 'Computer Science' )
          : G.Advisor in CS_DEPARTMENT.Has_faculty;
```

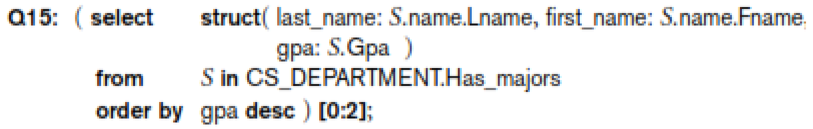
Note that query Q12 also illustrates how attribute, relationship, and operation inheritance applies to queries. Although *S* is an iterator that ranges over the extent GRAD_STUDENTS, we can write *S*.Majors_in because the Majors_in relationship is inherited by GRAD_STUDENT from STUDENT via extends. Finally, to illustrate the exists quantifier, query Q13 answers the following question: *Does any graduate Computer Science major have a 4.0 GPA?* Here, again, the operation gpa is inherited by GRAD_STUDENT from STUDENT via extends.

```
Q13: exists G in
         select  S from   S in GRAD_STUDENTS
         where  S.Majors_in.Dname = 'Computer Science' ) : G.Gpa = 4;
```

Ordered (Indexed) Collection Expressions. As we discussed in Section 11.3.3, collections that are lists and arrays have additional operations, such as retrieving the *i*th, first, and last elements. Additionally, operations exist for extracting a subcollection and concatenating two lists. Hence, query expressions that involve lists or arrays can invoke these operations. We will illustrate a few of these operations using sample queries. Q14 retrieves the last name of the faculty member who earns the highest salary:

```
Q14:  first ( select     struct(facname: F.name.Lname, salary: F.Salary)
              from       F in FACULTY
              order by   salary desc );
```

Q14 illustrates the use of the first operator on a list collection that contains the salaries of faculty members sorted in descending order by salary. Thus, the first ele-ment in this sorted list contains the faculty member with the highest salary. This query assumes that only one faculty member earns the maximum salary. The next query, Q15, retrieves the top three Computer Science majors based on GPA.



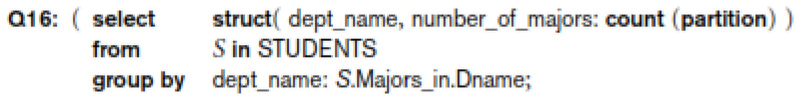
```
Q15:  ( select      struct( last_name: S.name.Lname, first_name: S.name.Fname,
                            gpa: S.Gpa  )
        from        S in CS_DEPARTMENT.Has_majors
        order by    gpa desc ) [0:2];
```

The select-from-order-by query returns a list of Computer Science students ordered by GPA in descending order. The first element of an ordered collection has an index position of 0, so the expression [0:2] returns a list containing the first, second, and third elements of the select ... from ... order by ... result.

The Grouping Operator. The group by clause in OQL, although similar to the corresponding clause in SQL, provides explicit reference to the collection of objects within each *group* or *partition.* First we give an example, and then we describe the general form of these queries.

Q16 retrieves the number of majors in each department. In this query, the students are grouped into the same partition (group) if they have the same major; that is, the

same value for *S*.Majors_in.Dname:



```
Q16:  ( select      struct( dept_name, number_of_majors: count (partition) )
        from        S in STUDENTS
        group by    dept_name: S.Majors_in.Dname;
```

The result of the grouping specification is of type set<struct(dept_name: string, parti-
tion: bag<struct(*S*:STUDENT>)>), which contains a struct for each group (partition) that has two components: the grouping attribute value (dept_name) and the bag of the STUDENT objects in the group

(partition). The select clause returns the grouping attribute (name of the department), and a count of the number of elements in each partition (that is, the number of students in each department), where partition is the keyword used to refer to each partition. The result type of the select clause is set<struct(dept_name: string, number_of_majors: integer)>. In general, the syntax for the group by clause is

group by F_1: E_1, F_2: E_2, ..., F_k: E_k
where F_1: E_1, F_2: E_2, ..., F_k: E_k is a list of partitioning (grouping) attributes and each partitioning attribute specification F_i: E_i defines an attribute (field) name F_i and an expression E_i. The result of applying the grouping (specified in the group by clause) is a set of structures:

set<struct(F_1: T_1, F_2: T_2, ..., F_k: T_k, partition: bag<B>)>

where T_i is the type returned by the expression E_i, partition is a distinguished field name (a keyword), and B is a structure whose fields are the iterator variables (*S* in Q16) declared in the from clause having the appropriate type.

Just as in SQL, a having clause can be used to filter the partitioned sets (that is, select only some of the groups based on group conditions). In Q17, the previous query is modified to illustrate the having clause (and also shows the simplified syntax for the select clause). Q17 retrieves for each department having more than 100 majors, the average GPA of its majors. The having clause in Q17 selects only those partitions (groups) that have more than 100 elements (that is, departments with more than 100 students).

```
Q17:   select      dept_name, avg_gpa: avg ( select P.gpa from P in partition)
       from        S in STUDENTS
       group by    dept_name: S.Majors_in.Dname
       having      count (partition) > 100;
```

Note that the select clause of Q17 returns the average GPA of the students in the partition. The expression

select *P*.Gpa from *P* in partition

returns a bag of student GPAs for that partition. The from clause declares an iterator variable *P* over the partition collection, which is of type bag<struct(*S*: STUDENT)>. Then the path expression *P*.gpa is used to access the GPA of each student in the partition.

5.11. XML DATABASES:

- XML (Extensible Markup Language)—has emerged as the standard for structuring and exchanging data over the Web. XML can be used to provide information about the structure and meaning of the data in the Web pages rather than just specifying how the Web pages are formatted for display on the screen.

- The formatting aspects are specified separately—for example, by using a formatting language such as XSL (Extensible Stylesheet Language) or a transformation language such as XSLT (Extensible Stylesheet Language for Transformations or simply XSL Transformations). XML has also been proposed as a possible model for data storage and retrieval, although only a few experimental database systems based on XML have been developed so far.

- Basic HTML is useful for generating *static* Web pages with fixed text and other objects, but most e-commerce applications require Web pages that provide interactive features with the user. For example, consider the case of an airline customer who wants to check the arrival time and gate information of a particular flight. The user may enter information such as a date and flight number in certain form fields of the Web page. The Web program must first submit a query to the airline database to retrieve this information, and then display it. Such Web pages, where part of the information is extracted from databases or other data sources are called *dynamic* Web pages, because the data extracted and displayed each time will be for different flights and dates.

<u>Structured, Semistructured, and Unstructured Data</u>:

The information stored in databases is known as **structured data** because it is represented in a strict format. For example, each record in a relational database table follows the same format as the other records in that table.

In some applications, data is collected in an ad hoc manner before it is known how it will be stored and managed. This data may have a certain structure, but not all the information collected will have the identical structure. Some attributes may be shared among the various entities, but other attributes may exist only in a few entities. Moreover, additional attributes can be introduced in some of the newer data items at any time, and there is no predefined schema. This type of data is known as **semistructured data**. A number of data models have been introduced for representing semistructured data, often based on using tree or graph data structures rather than the flat relational model structures.

A key difference between structured and semistructured data concerns how the schema constructs (such as the names of attributes, relationships, and entity types) are handled. In semistructured data, the schema information is *mixed in* with the data values, since each data object can have different attributes that are not known in advance. Hence, this type of data is sometimes referred to as **self-describing data**.

In Figure, the **labels** or **tags** on the directed edges represent the schema names: the *names of attributes*, *object types (*or *entity types* or *classes*), and *relationships*. The internal nodes represent individual objects or composite attrib utes. The leaf nodes represent actual data values of simple (atomic) attributes.

There are two main differences between the semistructured model and the object model:

1) The schema information—names of attributes, relationships, and classes (object types) in the semistructured model is intermixed with the objects and their data values in the same data structure.
2) In the semistructured model, there is no requirement for a predefined schema to which the data objects must conform, although it is possible to define a schema if necessary.

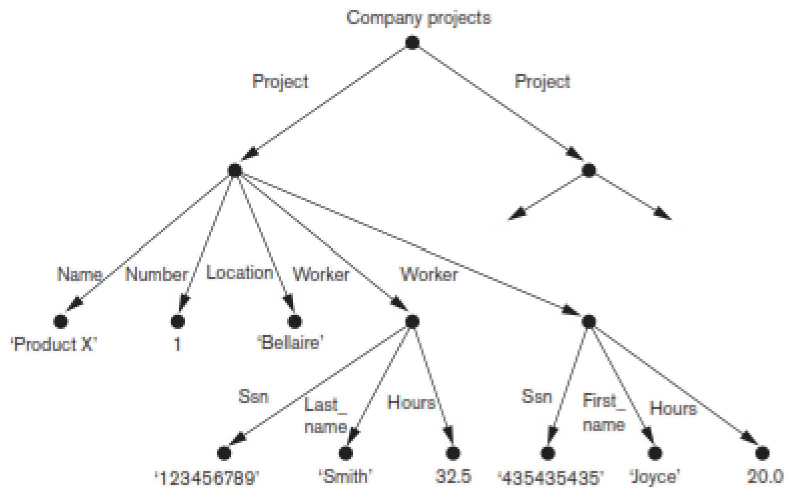


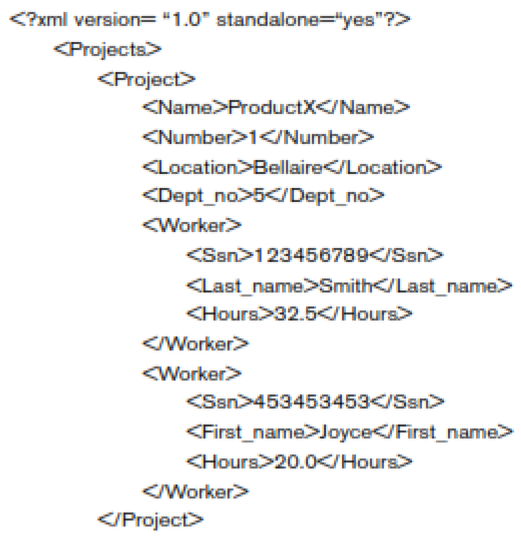
Figure. Representing semistructured data as a graph.

- In addition to structured and semistructured data, a third category exists, known as **unstructured data** because there is very limited indication of the type of data. A typical example is a text document that contains information embedded within it. Web pages in HTML that contain some data are considered to be unstructured data.

5.12. XML HIERARCHICAL (TREE) DATA MODEL

The basic object in XML is the XML document. Two main structuring concepts are used to construct an XML document: **elements** and **attributes**. It is important to note that the term *attribute* in XML is not used in the same manner as is customary in database terminology, but rather as it is used in document description languages such as HTML and SGML. Attributes in XML provide additional information that describes elements, as we will see. There are additional concepts in XML, such as entities, identifiers, and references, but first we concentrate on describing elements and attributes to show the essence of the XML model.

Figure shows an example of an XML element called <Projects>. As in HTML, elements are identified in a document by their start tag and end tag. The tag names are enclosed between angled brackets < ... >, and end tags are further identified by a slash, </ ... >.



```
<?xml version= "1.0" standalone="yes"?>
    <Projects>
        <Project>
            <Name>ProductX</Name>
            <Number>1</Number>
            <Location>Bellaire</Location>
            <Dept_no>5</Dept_no>
            <Worker>
                <Ssn>123456789</Ssn>
                <Last_name>Smith</Last_name>
                <Hours>32.5</Hours>
            </Worker>
            <Worker>
                <Ssn>453453453</Ssn>
                <First_name>Joyce</First_name>
                <Hours>20.0</Hours>
            </Worker>
        </Project>
    </Project>
    <Project>
        <Name>ProductY</Name>
        <Number>2</Number>
        <Location>Sugarland</Location>
        <Dept_no>5</Dept_no>
        <Worker>
            <Ssn>123456789</Ssn>
            <Hours>7.5</Hours>
        </Worker>
        <Worker>
            <Ssn>453453453</Ssn>
            <Hours>20.0</Hours>
        </Worker>
        <Worker>
            <Ssn>333445555</Ssn>
            <Hours>10.0</Hours>
        </Worker>
    </Project>
...
</Projects>
```

Figure . A complex XML element called <Projects>.

Complex elements are constructed from other elements hierarchically, whereas **simple elements** contain data values. A major difference between XML and HTML is that XML tag names are defined to describe the meaning of the data elements in the document, rather than to describe how the text is to be displayed. This makes it possible to process the data elements in the XML document automatically by computer programs. Also, the XML tag (element) names can be defined in another document, known as the *schema document*, to give a semantic meaning to the tag names that can be exchanged among multiple users. In HTML, all tag names are predefined and fixed; that is why they are not extendible.

It is straightforward to see the correspondence between the XML textual representation shown in Figure and the tree structure shown in Figure. In the tree representation, internal nodes represent complex elements, whereas leaf nodes rep-resent simple elements. That is why the XML model is called a **tree model** or a **hierarchical model**. In Figure, the simple elements are the ones with the tag names <Name>, <Number>, <Location>, <Dept_no>, <Ssn>, <Last_name>, <First_name>, and <Hours>. The complex elements are the ones with the tag names <Projects>, <Project>, and <Worker>. In general, there is no limit on the levels of nesting of elements.

It is possible to characterize three main types of XML documents:

Data-centric XML documents. These documents have many small data items that follow a specific structure and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them over or display them on the Web. These usually follow a *predefined schema* that defines the tag names.

Document-centric XML documents. These are documents with large amounts of text, such as news articles or books. There are few or no structured data elements in these documents.

Hybrid XML documents. These documents may have parts that contain structured data and other parts that are predominantly textual or unstructured. They may or may not have a predefined schema.

XML documents that do not follow a predefined schema of element names and corresponding tree structure are known as **schemaless XML documents**. It is important to note that data-centric XML documents can be considered either as semistructured data or as structured data as defined in Section 12.1. If an XML document conforms to a predefined XML schema or DTD (see Section 12.3), then the document can be considered as *structured data*. On the other hand, XML allows documents that do not conform to any schema; these would be considered as *semistructured data* and are *schemaless XML documents*. When the value of the standalone attribute in an XML document is yes, as in the first line in Figure, the document is standalone and schemaless.

XML attributes are generally used in a manner similar to how they are used in HTML to describe properties and characteristics of the elements (tags) within which they appear. It is also possible to use XML attributes to hold the values of simple data elements; however, this is generally not recommended. An exception to this rule is in cases that need to **reference** another element in another part of the XML document. To do this, it is common to use attribute values in one element as the references. This resembles the concept of foreign keys in relational databases, and is a way to get around the strict hierarchical model that the XML tree model implies.

5.13.XML DOCUMENTS, DTD, AND XML SCHEMA:

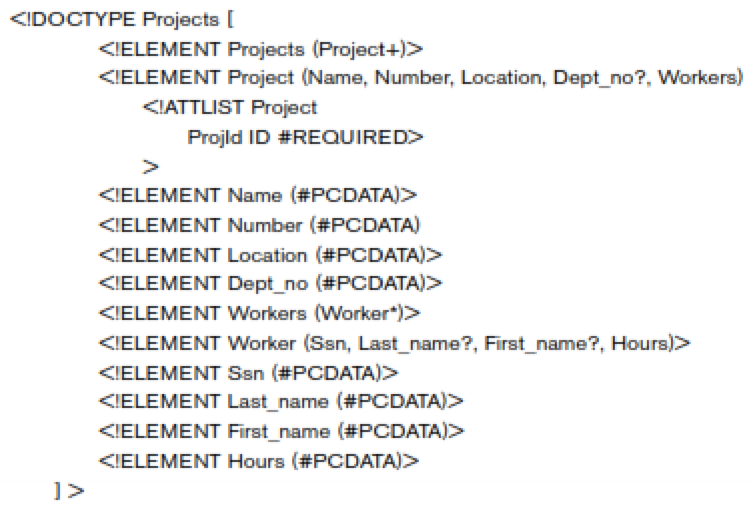
Well-Formed and Valid XML Documents and XML DTD:

An XML document is **well formed** if it follows a few conditions. In particular, it must start with an **XML declaration** to indicate the version of XML being used as well as any other relevant attributes, as shown in the first line in Figure. It must also follow the syntactic guidelines of the tree data model. This means that there should be a *single root element*, and every element must include a matching pair of start and end tags *within* the start and end tags *of the parent element*. This ensures that the nested elements specify a well-formed tree structure.

A well-formed XML document is syntactically correct. This allows it to be processed by generic processors that traverse the document and create an internal tree representation. A standard model with an associated set of API (application programming interface) functions called **DOM** (Document Object Model) allows programs to manipulate the resulting tree representation corresponding to a well-formed XML document. However, the whole document must be parsed beforehand when using DOM in order to convert the document to that standard DOM internal data structure representation. Another API called **SAX** (Simple API for XML) allows processing of XML documents on the fly by notifying the processing program through callbacks whenever a start or end tag is encountered. This makes it easier to process large documents and allows for processing of so-called **streaming XML documents**, where the processing program can process the tags as they are encountered. This is also known as **event-based processing**.

A well-formed XML document can be schemaless; that is, it can have any tag names for the elements within the document. In this case, there is no predefined set of elements (tag names) that a program processing the document knows to expect. This gives the document creator the freedom to specify new elements, but limits the pos-sibilities for automatically interpreting the meaning or semantics of the elements within the document.

A stronger criterion is for an XML document to be **valid**. In this case, the document must be well formed, and it must follow a particular schema. That is, the element names used in the start and end tag pairs must follow the structure specified in a separate XML **DTD** (**Document Type Definition**) file or XML schema file. Figure shows a simple XML DTD file, which specifies the elements (tag names) and their nested structures. Any valid documents conforming to this DTD should follow the specified structure. A special syntax exists for specifying DTD files. First, a name is given to the **root tag** of the document, which is called Projects in the first line in Figure. Then the elements and their nested structure are specified.



```
<!DOCTYPE Projects [
        <!ELEMENT Projects (Project+)>
        <!ELEMENT Project (Name, Number, Location, Dept_no?, Workers)
            <!ATTLIST Project
                ProjId ID #REQUIRED>
            >
        <!ELEMENT Name (#PCDATA)>
        <!ELEMENT Number (#PCDATA)
        <!ELEMENT Location (#PCDATA)>
        <!ELEMENT Dept_no (#PCDATA)>
        <!ELEMENT Workers (Worker*)>
        <!ELEMENT Worker (Ssn, Last_name?, First_name?, Hours)>
        <!ELEMENT Ssn (#PCDATA)>
        <!ELEMENT Last_name (#PCDATA)>
        <!ELEMENT First_name (#PCDATA)>
        <!ELEMENT Hours (#PCDATA)>
    ] >
```

Figure. An XML DTD file called *Projects*.

When specifying elements, the following notation is used:

A * following the element name means that the element can be repeated zero or more times in the document. This kind of element is known as an *optional multivalued (repeating) element.*

A + following the element name means that the element can be repeated one or more times in the document. This kind of element is a *required multivalued (repeating) element.*

A ? following the element name means that the element can be repeated zero or one times. This kind is an *optional single-valued (nonrepeating) element.*

An element appearing without any of the preceding three symbols must appear exactly once in the document. This kind is a *required single-valued (nonrepeating) element.*

The **type** of the element is specified via parentheses following the element. If the parentheses include names of other elements, these latter elements are the *children* of the element in the tree structure. If the parentheses include the keyword #PCDATA or one of the other data types available in XML DTD, the element is a leaf node. PCDATA stands for *parsed character data*, which is roughly similar to a string data type.

The list of attributes that can appear within an element can also be specified via the keyword !ATTLIST. In Figure 12.3, the Project element has an attribute ProjId. If the type of an attribute is ID, then it can be referenced from another attribute whose type is IDREF within another element. Notice that attributes can also be used to hold the values of simple data elements of type #PCDATA.

Parentheses can be nested when specifying elements.

A bar symbol (e_1 | e_2) specifies that either e_1 or e_2 can appear in the docu-ment.

We can see that the tree structure in Figure 12.1 and the XML document in Figure 12.3 conform to the XML DTD in Figure 12.4. To require that an XML document be checked for conformance to a DTD, we must specify this in the declaration of the document. For example, we could change the first line in Figure 12.3 to the following:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Projects SYSTEM "proj.dtd">
```

When the value of the standalone attribute in an XML document is "no", the docu-ment needs to be checked against a separate DTD document or XML schema docu-ment (see below). The DTD file shown in Figure 12.4 should be stored in the same file system as the XML document, and should be given the file name proj.dtd. Alternatively, we could include the DTD document text at the beginning of the XML document itself to allow the checking.

Although XML DTD is quite adequate for specifying tree structures with required, optional, and repeating elements, and with various types of attributes, it has several limitations. First, the data types in DTD are not very general. Second, DTD has its own special syntax and thus requires specialized processors. It would be advanta-geous to specify XML schema documents using the syntax rules of XML itself so that the same processors used for XML documents could process XML schema descriptions. Third, all DTD elements are always forced to follow the specified ordering of the document, so unordered elements are not permitted. These draw-backs led to the development of XML schema, a more general but also more com-plex language for specifying the structure and elements of XML documents.

5.14. XML SCHEMA

- An XML Schema describes the structure of an XML document, just like a DTD.
- The XML Schema language is also referred to as XML Schema Definition (XSD).
- An XML document is called "well-formed" if it contains the correct syntax. A well-formed and valid XML document is one which have been validated against Schema.

XML Schema Example
Create a schema file.
employee.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.javatpoint.com"
xmlns="http://www.javatpoint.com"
elementFormDefault="qualified">

<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:element name="email" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

xml file using XML schema or XSD file.
employee.xml

```
<?xml version="1.0"?>
<employee
xmlns="http://www.javatpoint.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.javatpoint.com employee.xsd">
```

```
  <firstname>vimal</firstname>
  <lastname>jaiswal</lastname>
  <email>vimal@javatpoint.com</email>
</employee>
```

Description of XML Schema

<xs:element name="employee"> :
It defines the element name employee.
<xs:complexType> :
It defines that the element 'employee' is complex type.
<xs:sequence> :
It defines that the complex type is a sequence of elements.
<xs:element name="firstname" type="xs:string"/> :
It defines that the element 'firstname' is of string/text type.
<xs:element name="lastname" type="xs:string"/> :
It defines that the element 'lastname' is of string/text type.
<xs:element name="email" type="xs:string"/> :
It defines that the element 'email' is of string/text type.

XML Schema Data types

There are two types of data types in XML schema.
1. simpleType
2. complexType

1) simpleType

The simpleType allows you to have text-based elements. It contains less attributes, child elements, and cannot be left empty.

2) complexType

The complexType allows you to hold multiple attributes and elements. It can contain additional sub elements and can be left empty.

DTD vs XSD

There are many differences between DTD (Document Type Definition) and XSD (XML Schema Definition). In short, DTD provides less control on XML structure whereas XSD (XML schema) provides more control.
The important differences are given below:

No.	DTD	XSD
1)	DTD stands for **Document Type Definition**.	XSD stands for XML Schema Definition.
2)	DTDs are derived from **SGML** syntax.	XSDs are written in XML.
3)	DTD **doesn't support datatypes**.	XSD **supports datatypes** for elements and attributes.
4)	DTD **doesn't support namespace**.	XSD **supports namespace**.

5)	DTD **doesn't define order** for child elements.	XSD **defines order** for child elements.
6)	DTD is **not extensible**.	XSD is **extensible**.
7)	DTD is **not simple to learn**.	XSD is **simple to learn** because you don't need to learn new language.
8)	DTD provides **less control** on XML structure.	XSD provides **more control** on XML structure.

5.15. XQuery: SPECIFYING QUERIES IN XML

- XQuery permits the specification of more general queries on one or more XML documents. The typical form of a query in XQuery is known as a **FLWR expression**, which stands for the four main clauses of XQuery and has the following form:

```
FOR <variable bindings to individual nodes (elements)>
LET <variable bindings to collections of nodes (elements)>
WHERE <qualifier conditions>
RETURN <query result specification>
```

- There can be zero or more instances of the FOR clause, as well as of the LET clause in a single XQuery. The WHERE clause is optional, but can appear at most once, and the RETURN clause must appear exactly once. Let us illustrate these clauses with the following simple example of an XQuery.

```
            LET $d := doc(www.company.com/info.xml)
            FOR $x IN $d/company/project[projectNumber = 5]/projectWorker, $y IN
$d/company/employee
            WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
            RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>
```

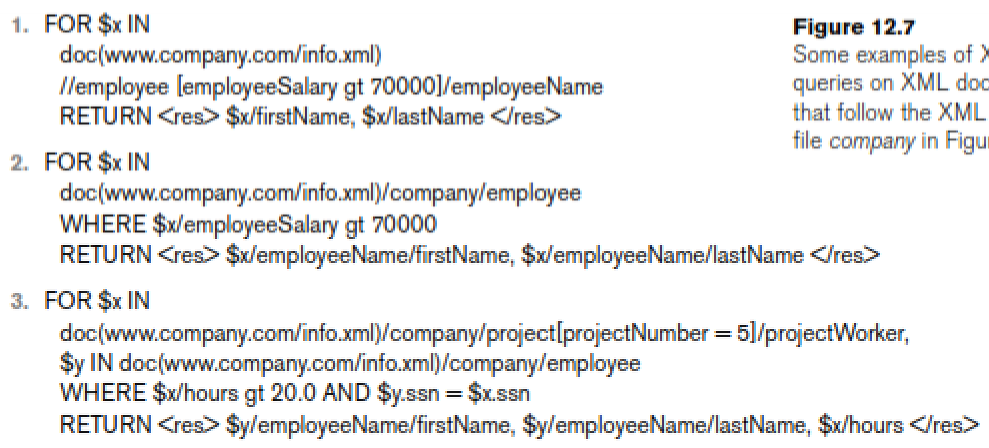
- Variables are prefixed with the $ sign. In the above example, $d, $x, and $y are variables.
- The LET clause assigns a variable to a particular expression for the rest of the query. In this example, $d is assigned to the document file name. It is possible to have a query that refers to multiple documents by assigning multiple variables in this way.
- The FOR clause assigns a variable to range over each of the individual items in a sequence. In our example, the sequences are specified by path expressions. The $x variable ranges over elements that satisfy the path expression $d/company/project[projectNumber = 5]/projectWorker. The $y variable ranges over elements that satisfy the path expression $d/company/employee. Hence, $x ranges over projectWorker elements, whereas $y ranges over employee elements.
- The WHERE clause specifies additional conditions on the selection of items. In this example, the first condition selects only those projectWorker elements that satisfy the condition (hours gt 20.0). The second condition specifies a join condition that combines an employee with a projectWorker only if they have the same ssn value.
- Finally, the RETURN clause specifies which elements or attributes should be retrieved from the items that satisfy the query conditions. In this example, I will return a sequence of elements each containing <firstName, lastName, hours> for employees who work more that 20 hours per week on project number 5.
- Figure includes some additional examples of queries in XQuery that can be specified on an XML instance documents that follow the XML schema document in Figure 12.5. The first query retrieves the first and last names of employees who earn more than $70,000. The variable $x is bound to each employeeName

element that is a child of an employee element, but only for employee elements that satisfy the qualifier that their employeeSalary value is greater than $70,000. The result retrieves the firstName and lastName child elements of the selected employeeName elements. The second query is an alternative way of retrieving the same elements retrieved by the first query.

- The third query illustrates how a join operation can be performed by using more than one variable. Here, the $x variable is bound to each projectWorker element that is a child of project number 5, whereas the $y variable is bound to each employee element. The join condition matches ssn values in order to retrieve the employee names. Notice that this is an alternative way of specifying the same query in our earlier example, but without the LET clause.

- XQuery has very powerful constructs to specify complex queries. In particular, it can specify universal and existential quantifiers in the conditions of a query, aggregate functions, ordering of query results, selection based on position in a sequence, and even conditional branching. Hence, in some ways, it qualifies as a fullfledged programming language.

- This concludes our brief introduction to XQuery. The interested reader is referred to www.w3.org, which contains documents describing the latest standards related to XML and XQuery.



```
1. FOR $x IN
     doc(www.company.com/info.xml)
     //employee [employeeSalary gt 70000]/employeeName
     RETURN <res> $x/firstName, $x/lastName </res>
2. FOR $x IN
     doc(www.company.com/info.xml)/company/employee
     WHERE $x/employeeSalary gt 70000
     RETURN <res> $x/employeeName/firstName, $x/employeeName/lastName </res>
3. FOR $x IN
     doc(www.company.com/info.xml)/company/project[projectNumber = 5]/projectWorker,
     $y IN doc(www.company.com/info.xml)/company/employee
     WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
     RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>
```

Figure 12.7
Some examples of XQuery queries on XML documents that follow the XML schema file *company* in Figure 12.5.

5.16. INFORMATION RETRIEVAL (IR) CONCEPTS

- **Information retrieval** is the process of retrieving documents from a collection in response to a query (or a search request) by a user.

Introduction to Information Retrieval

Consider a relation (or table) called HOUSES with the attributes:

HOUSES(Lot#, Address, Square_footage, Listed_price)

- This is an example of ***structured data.***
- We can compare this relation with home-buying contract documents, which are examples of ***unstructured data***. These types of documents can vary from city to city, and even county to county, within a given state in the United States. Typically, a contract document in a particular state will have a standard list of clauses described in paragraphs within sections of the document, with some predetermined (fixed) text and some variable areas whose content is to be supplied by the specific buyer and seller.
- Other variable information would include interest rate for financing, down-payment amount, closing dates, and so on. The documents could also possibly include some pictures taken during a home inspection. The information content in such documents can be considered *unstructured data* that can be

stored in a variety of possible arrangements and for-mats. By **unstructured information**, we generally mean information that does not have a well-defined formal model and corresponding formal language for representation and reasoning, but rather is based on understanding of natural language.

- With the advent of the World Wide Web (or Web, for short), the volume of unstructured information stored in messages and documents that contain textual and multimedia information has exploded. These documents are stored in a variety of standard formats, including HTML, XML, and several audio and video formatting standards. Information retrieval deals with the problems of storing, indexing, and retrieving (searching) such information to satisfy the needs of users. The problems that IR deals with are exacerbated by the fact that the number of Web pages and the number of social interaction events is already in the billions, and is growing at a phenomenal rate. All forms of unstructured data described above are being added at the rates of millions per day, expanding the searchable space on the Web at rapidly increasing rates.

- **Information retrieval** is "the discipline that deals with the structure, analysis, organization, storage, searching, and retrieval of information" .

- We can enhance the definition slightly to say that it applies in the context of unstructured documents to satisfy a user's information needs. This field has existed even longer than the database field, and was originally concerned with retrieval of cataloged information in libraries based on titles, authors, topics, and keywords. In academic programs, the field of IR has long been a part of Library and Information Science programs. Information in the context of IR does not require machine-understandable structures, such as in relational database systems. Examples of such information include written texts, abstracts, documents, books, Web pages, e-mails, instant messages, and collections from digital libraries. Therefore, all loosely represented (unstructured) or semistructured information is also part of the IR discipline.

- RDBMS vendors are providing modules to support many of these data types, as well as XML data, in the newer versions of their products, sometimes referred to as *extended RDBMSs*, or *object-relational database management systems*. The challenge of dealing with unstructured data is largely an information retrieval problem, although database researchers have been applying data-base indexing and search techniques to some of these problems.

- IR systems go beyond database systems in that they do not limit the user to a specific query language, nor do they expect the user to know the structure (schema) or content of a particular database. IR systems use a user's information need expressed as a **free-form search request** (sometimes called a **keyword search query**, or just **query**) for interpretation by the system. Whereas the IR field historically dealt with cataloging, processing, and accessing text in the form of documents for decades, in today's world the use of Web search engines is becoming the dominant way to find information. The traditional problems of text indexing and making collections of documents searchable have been transformed by making the Web itself into a quickly accessible repository of human knowledge.

An IR system can be characterized at different levels:

1) **Types of Users**. The user may be an *expert user* (for example, a curator or a librarian), who is searching for specific information that is clear in his/her mind and forms relevant queries for the task, or a *layperson user* with a generic information need. The latter cannot create highly relevant queries for search (for example, students trying to find information about a new topic, researchers try-ing to assimilate different points of view about a historical issue, a scientist verifying a claim by another scientist, or a person trying to shop for clothing).
2) **Types of Data**. Search systems can be tailored to specific types of data. For example, the problem of retrieving information about a specific topic may be handled more efficiently by customized search systems that are built to collect and retrieve only information related to that specific topic. The information repository could be hierarchically organized based on a concept or topic hierarchy. These

topical *domain-specific* or *vertical IR systems* are not as large as or as diverse as the generic World Wide Web, which contains information on all kinds of topics. Given that these domain-specific collections exist and may have been acquired through a specific process, they can be exploited much more efficiently by a specialized system.

3) **Types of Information Need.** In the context of Web search, users' information needs may be defined as navigational, informational, or transactional.[3] **Navigational search** refers to finding a particular piece of information (such as the Georgia Tech University Website) that a user needs quickly. The purpose of **informational search** is to find current information about a topic (such as research activities in the college of computing at Georgia Tech—this is the clas-sic IR system task). The goal of **transactional search** is to reach a site where fur-ther interaction happens (such as joining a social network, product shopping, online reservations, accessing databases, and so on).

Levels of Scale:

- This overabundance of information sources in effect creates a high noise-to-signal ratio in IR systems. Especially on the Web, where billions of pages are indexed, IR interfaces are built with efficient scalable algorithms for distributed searching, indexing, caching, merging, and fault tolerance. IR search engines can be limited in level to more specific collections of documents.
- **Enterprise search systems** offer IR solutions for searching different entities in an enterprise's **intranet**, which consists of the network of computers within that enterprise. The searchable entities include e-mails, corporate documents, manuals, charts, and presentations, as well as reports related to people, meetings, and projects. They still typically deal with hundreds of millions of entities in large global enterprises.
- On a smaller scale, there are personal information systems such as those on desktops and laptops, called **desktop search engines** (for example, Google Desktop), for retrieving files, folders, and different kinds of entities stored on the computer. There are peer-to-peer systems, such as BitTorrent, which allows sharing of music in the form of audio files, as well as specialized search engines for audio, such as Lycos and Yahoo! audio search.

Databases and IR Systems: A Comparison:

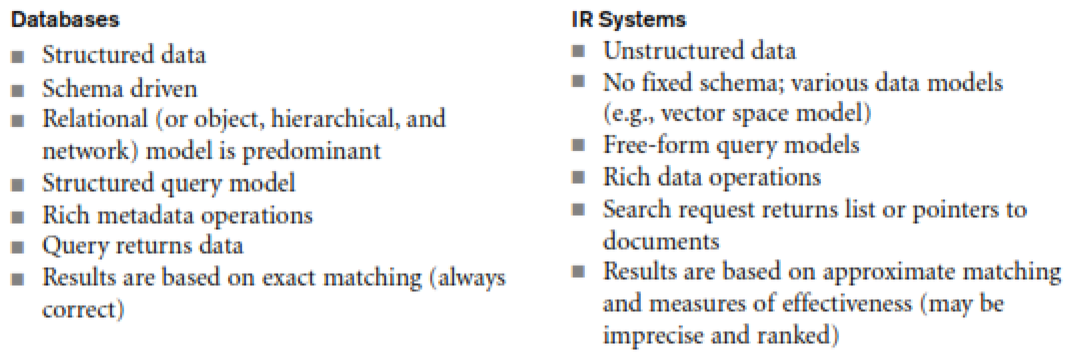
- Within the computer science discipline, databases and IR systems are closely related fields. Databases deal with structured information retrieval through well-defined formal languages for representation and manipulation based on the theoretically founded data models. Efficient algorithms have been developed for operators that allow rapid execution of complex queries. IR, on the other hand, deals with unstructured search with possibly vague query or search semantics and without a well-defined logical schematic representation. Some of the key differences between databases and IR systems are listed in Table.

- Whereas databases have fixed schemas defined in some data model such as the relational model, an IR system has no fixed data model; it views data or documents according to some scheme, such as the vector space model, to aid in query processing. Databases using the relational model employ SQL for queries and transactions. The queries are mapped into relational algebra operations and search algorithms and return a new relation (table) as the query result, providing an exact answer to the query for the current state of the database. In IR systems, there is no fixed language for defining the structure (schema) of the document or for operating on the document—queries tend to be a set of query terms (keywords) or a free-form natural language phrase. An IR query result is a list of document ids, or some pieces of text or multimedia objects (images, videos, and so on), or a list of links to Web pages.

- The result of a database query is an exact answer; if no matching records (tuples) are found in the relation, the result is empty (null). On the other hand, the answer to a user request in an IR query represents the IR system's best attempt at retrieving the information most relevant to that query.
- Whereas database systems maintain a large amount of metadata and allow their use in query optimization, the operations in IR systems rely on the data values themselves and their occurrence frequencies. Complex

statistical analysis is sometimes performed to determine the *relevance* of each document or parts of a document to the user request.

Table. A Comparison of Databases and IR Systems



Databases	IR Systems
■ Structured data	■ Unstructured data
■ Schema driven	■ No fixed schema; various data models (e.g., vector space model)
■ Relational (or object, hierarchical, and network) model is predominant	■ Free-form query models
■ Structured query model	■ Rich data operations
■ Rich metadata operations	■ Search request returns list or pointers to documents
■ Query returns data	■ Results are based on approximate matching and measures of effectiveness (may be imprecise and ranked)
■ Results are based on exact matching (always correct)	

- A **search engine** is a practical application of information retrieval to large-scale document collections. With significant advances in computers and communications technologies, people today have interactive access to enormous amounts of user-generated distributed content on the Web. This has spurred the rapid growth in search engine technology, where search engines are trying to discover different kinds of real-time content found on the Web.
- The part of a search engine responsible for discovering, analyzing, and indexing these new documents is known as a **crawler**. Other types of search engines exist for specific domains of knowledge.
- For example, the biomedical literature search database was started in the 1970s and is now supported by the PubMed search engine, which gives access to over 20 million abstracts.

- While continuous progress is being made to tailor search results to the needs of an end user, the challenge remains in providing high-quality, pertinent, and timely information that is precisely aligned to the information needs of individual users.

Modes of Interaction in IR Systems

- Information retrieval is the process of retrieving documents from a collection in response to a query (or a search request) by a user. Typically the collection is made up of documents containing unstructured data. Other kinds of documents include images, audio recordings, video strips, and maps. Data may be scattered non uniformly in these documents with no definitive structure.
- A **query** is a set of **terms** (also referred to as **keywords**) used by the searcher to specify an information need (for example, the terms 'databases' and 'operating systems' may be regarded as a query to a computer science bibliographic database). An informational request or a search query may also be a natural language phrase or a question (for example, "What is the currency of China?" or "Find Italian restaurants in Sarasota, Florida.").

There are two main modes of interaction with IR systems

1) retrieval
2) browsing

- **Retrieval** is concerned with the extraction of relevant information from a repository of documents through an IR query
- **Browsing** signifies the activity of a user visiting or navigating through similar or related documents based on the user's assessment of relevance. During browsing, a user's information need may not be defined *a priori* and is flexible.

- Consider the following browsing scenario: A user specifies ‘Atlanta’ as a keyword. The information retrieval system retrieves links to relevant result documents containing various aspects of Atlanta for the user. The user comes across the term ‘Georgia Tech’ in one of the returned documents, and uses some access technique (such as clicking on the phrase ‘Georgia Tech’ in a document, which has a built-in link) and visits documents about Georgia Tech in the same or a different Website (repository). There the user finds an entry for ‘Athletics’ that leads the user to information about various athletic programs at Georgia Tech. Eventually, the user ends his search at the Fall schedule for the Yellow Jackets foot-ball team, which he finds to be of great interest. This user activity is known as browsing. **Hyperlinks** are used to interconnect Web pages and are mainly used for browsing. **Anchor texts** are text phrases within documents used to label hyperlinks and are very relevant to browsing.

  - **Web search** combines both aspects—browsing and retrieval—and is one of the main applications of information retrieval today. Web pages are analogous to documents. Web search engines maintain an indexed repository of Web pages, usually using the technique of inverted indexing. They retrieve the most relevant Web pages for the user in response to the user’s search request with a possible ranking in descending order of relevance. The **rank of a Webpage** in a retrieved set is the measure of its relevance to the query that generated the result set.

<u>Generic IR Pipeline</u>

- Documents are made up of unstructured natural language text composed of character strings from English and other languages. Common examples of documents include newswire services, corporate manuals and reports, government notices, Web page articles, blogs, tweets, books, and journal papers. There are two main approaches to IR: statistical and semantic.
- In a **statistical approach**, documents are analyzed and broken down into chunks of text (words, phrases, or *n*-grams, which are all subsequences of length *n* characters in a text or document) and each word or phrase is counted, weighted, and measured for relevance or importance. These words and their properties are then compared with the query terms for potential degree of match to produce a ranked list of resulting documents that contain the words. Statistical approaches are further classified based on the method employed. The three main statistical approaches are Boolean, vector space, and probabilistic.
- **Semantic approaches** to IR use knowledge-based techniques of retrieval that broadly rely on the syntactic, lexical, sentential, discourse-based, and pragmatic lev-els of knowledge understanding. In practice, semantic approaches also apply some form of statistical analysis to improve the retrieval process.
- Figure shows the various stages involved in an IR processing system. The steps shown on the left in Figure are typically offline processes, which prepare a set of documents for efficient retrieval; these are document preprocessing, document modeling, and indexing. The steps involved in query formation, query processing, searching mechanism, document retrieval, and relevance feedback are shown on the right in Figure. In each box, we highlight the important concepts and issues. The rest of this chapter describes some of the concepts involved in the various tasks within the IR process shown in Figure.

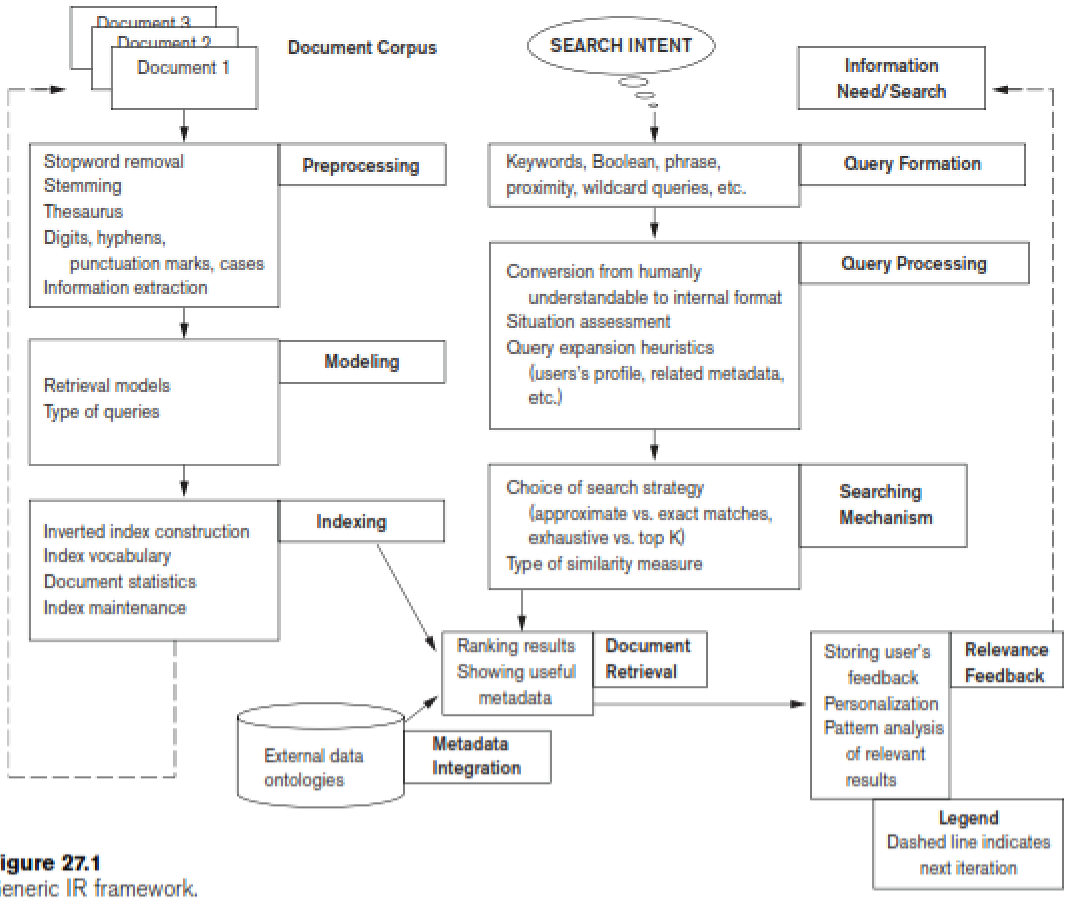


Figure 27.1
Generic IR framework.

- Figure shows a simplified IR processing pipeline. In order to perform retrieval on documents, the documents are first represented in a form suitable for retrieval. The significant terms and their properties are extracted from the documents and are represented in a document index where the words/terms and their properties are stored in a matrix that contains these terms and the references to the documents that contain them.
- This index is then converted into an inverted index of a word/term vs. document matrix. Given the query words, the documents containing these words—and the document properties, such as date of creation, author, and type of document—are fetched from the inverted index and compared with the query. This comparison results in a ranked list shown to the user.
- The user can then provide feedback on the results that triggers implicit or explicit query expansion to fetch results that are more relevant for the user. Most IR systems allow for an interactive search where the query and the results are successively refined.

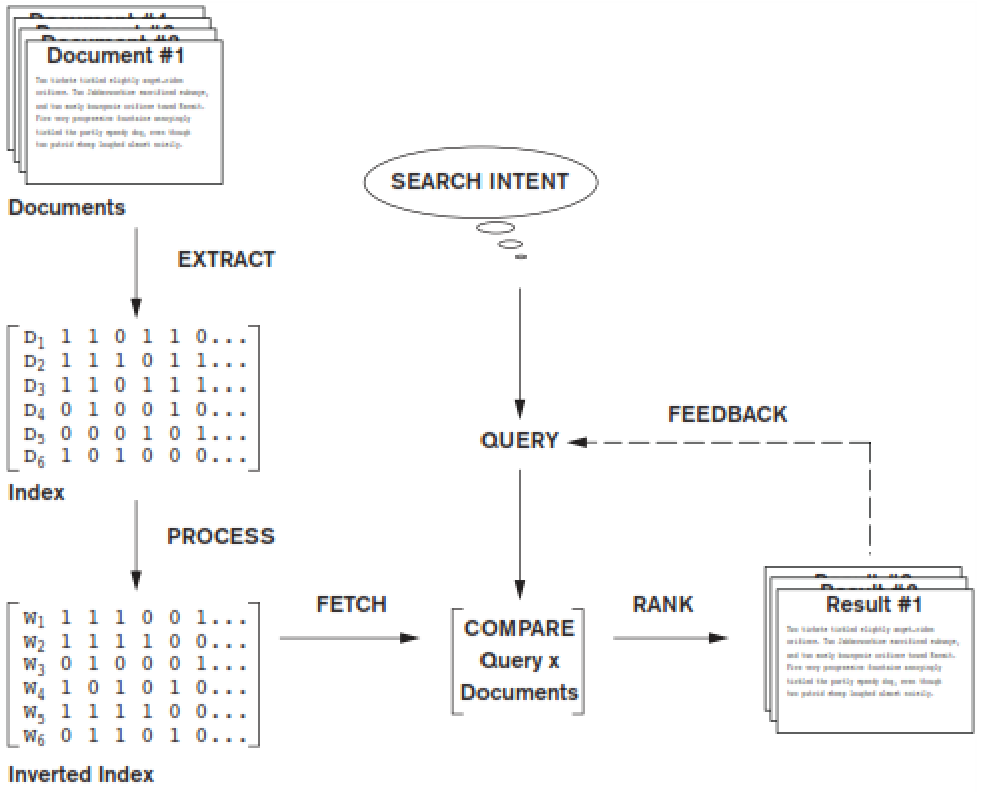


Figure Simplified IR process pipeline.

<u>5.17. RETRIEVAL MODELS:</u>

Three main statistical models—Boolean, vector space, and probabilistic—and the semantic model.

1) Boolean Model

- In this model, documents are represented as a set of *terms*. Queries are formulated as a combination of terms using the standard Boolean logic set-theoretic operators such as AND, OR and NOT. Retrieval and relevance are considered as binary concepts in this model, so the retrieved elements are an “exact match” retrieval of relevant documents. There is no notion of ranking of resulting documents.
- All retrieved documents are considered equally important—a major simplification that does not consider frequencies of document terms or their proximity to other terms com-pared against the query terms.
- Boolean retrieval models lack sophisticated ranking algorithms and are among the earliest and simplest information retrieval models. These models make it easy to associate metadata information and write queries that match the contents of the documents as well as other properties of documents, such as date of creation, author, and type of document.

2) Vector Space Model

- The vector space model provides a framework in which term weighting, ranking of retrieved documents, and relevance feedback are possible. Documents are represented as *features* and *weights* of term features in an *n*-dimensional vector space of terms.

- **Features** are a subset of the terms in a *set of documents* that are deemed most relevant to an IR search for this particular set of documents. The process of selecting these important terms (features) and their properties as a sparse (limited) list out of the very large number of available terms (the vocabulary can contain hundreds of thousands of terms) is independent of the model specification. The query is also specified as a terms vector (vector of features), and this is compared to the document vectors for similarity/relevance assessment.
- The similarity assessment function that compares two vectors is not inherent to the model—different similarity functions can be used. However, the cosine of the angle between the query and document vector is a commonly used function for similarity assessment. As the angle between the vectors decreases, the cosine of the angle approaches one, meaning that the similarity of the query with a document vector increases. Terms (features) are weighted proportional to their frequency counts to reflect the importance of terms in the calculation of relevance measure. This is different from the Boolean model, which does not take into account the frequency of words in the document for relevance match.
- In the vector model, the *document term weight* w_{ij} (for term *i* in document *j*) is represented based on some variation of the TF (term frequency) or TF-IDF (term frequency-inverse document frequency) scheme (as we will describe below). **TF-IDF** is a statistical weight measure that is used to evaluate the importance of a document word in a collection of documents. The following formula is typically used:

$$\text{cosine}(d_j, q) = \frac{\langle d_j \times q \rangle}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{|V|} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{|V|} w_{ij}^2} \times \sqrt{\sum_{i=1}^{|V|} w_{iq}^2}}$$

In the formula given above, we use the following symbols:

d_j is the document vector.
q is the query vector.
w_{ij} is the weight of term *i* in document *j*.
w_{iq} is the weight of term *i* in query vector *q*.
$|V|$ is the number of dimensions in the vector that is the total number of important keywords (or features).

TF-IDF uses the product of normalized frequency of a term *i* (TF_{ij}) in document D_j and the inverse document frequency of the term *i* (IDF_i) to weight a term in a document. The idea is that terms that capture the essence of a document occur fre-quently in the document (that is, their TF is high), but if such a term were to be a good term that discriminates the document from others, it must occur in only a few documents in the general population (that is, its IDF should be high as well).

IDF values can be easily computed for a fixed collection of documents. In case of Web search engines, taking a representative sample of documents approximates IDF computation. The following formulas can be used:

$$TF_{ij} = f_{ij} \Big/ \sum_{i=1 \text{ to } |V|} f_{ij}$$

$$IDF_i = \log\left(N / n_i\right)$$

In these formulas, the meaning of the symbols is:

TF_{ij} is the normalized term frequency of term *i* in document D_j.
f_{ij} is the number of occurrences of term *i* in document D_j.
IDF_i is the inverse document frequency weight for term *i*.
N is the number of documents in the collection.
n_i is the number of documents in which term *i* occurs.

Note that if a term i occurs in all documents, then $n_i = N$ and hence $IDF_i = \log(1)$ becomes zero, nullifying its importance and creating a situation where division by zero can occur. The weight of term i in document j, w_{ij} is computed based on its TF-IDF value in some techniques. To prevent division by zero, it is common to add a 1 to the denominator in the formulae such as the cosine formula above.

Sometimes, the relevance of the document with respect to a query (rel(D_j,Q)) is directly measured as the sum of the TF-IDF values of the terms in the Query Q:

$$\text{rel}(D_j,Q)=\sum_{i\in Q} TF_{ij}\times IDF_i$$

The normalization factor (similar to the denominator of the cosine formula) is incorporated into the TF-IDF formula itself, thereby measuring relevance of a document to the query by the computation of the dot product of the query and document vectors.

The Rocchio algorithm is a well-known relevance feedback algorithm based on the vector space model that modifies the initial query vector and its weights in response to user-identified relevant documents. It expands the original query vector q to a new vector q_e as follows:

$$q_e=\alpha q+\frac{\beta}{|D_r|}\sum_{d_r\in D_r} d_r-\frac{\gamma}{|D_{ir}|}\sum_{d_{ir}\in D_{ir}} d_{ir},$$

Here, D_r and D_{ir} are relevant and non relevant document sets and α, β, and γ are parameters of the equation. The values of these parameters determine how the feed-back affects the original query, and these may be determined after a number of trial-and-error experiments.

3) Probabilistic Model

The similarity measures in the vector space model are somewhat ad hoc. For example, the model assumes that those documents closer to the query in cosine space are more relevant to the query vector. In the probabilistic model, a more concrete and definitive approach is taken: ranking documents by their estimated probability of relevance with respect to the query and the document. This is the basis of the *Probability Ranking Principle* developed by Robertson:[11]

In the probabilistic framework, the IR system has to decide whether the documents belong to the **relevant set** or the **nonrelevant** set for a query. To make this decision, it is assumed that a predefined relevant set and nonrelevant set exist for the query, and the task is to calculate the probability that the document belongs to the relevant set and compare that with the probability that the document belongs to the nonrelevant set.

Given the document representation D of a document, estimating the relevance R and nonrelevance NR of that document involves computation of conditional probability $P(R|D)$ and $P(NR|D)$. These conditional probabilities can be calculated using Bayes' Rule:

$P(R|D) = P(D|R) \times P(R)/P(D)$

$P(NR|D) = P(D|NR) \times P(NR)/P(D)$

A document D is classified as relevant if $P(R|D) > P(NR|D)$. Discarding the constant
$P(D)$, this is equivalent to saying that a document is relevant if:

$P(D|R) \times P(R) > P(D|NR) \times P(NR)$

The likelihood ratio $P(D|R)/P(D|NR)$ is used as a score to determine the likelihood of the document with representation D belonging to the relevant set.

The *term independence* or *Naïve Bayes* assumption is used to estimate $P(D|R)$ using computation of $P(t_i|R)$ for term t_i. The likelihood ratios $P(D|R)/P(D|NR)$ of docu-ments are used as a proxy for ranking based on the assumption that highly ranked documents will have a high likelihood of belonging to the relevant set.[13]

With some reasonable assumptions and estimates about the probabilistic model along with extensions for incorporating query term weights and document term weights in the model, a probabilistic ranking algorithm called **BM25** (Best Match 25) is quite popular. This weighting scheme has evolved from several versions of the **Okapi** system.
The Okapi weight for Document d_j and query q is computed by the formula below.
Additional notations are as follows:

t_i is a term.
f_{ij} is the raw frequency count of term t_i in document d_j.
f_{iq} is the raw frequency count of term t_i in query q.
N is the total number of documents in the collection.
df_i is the number of documents that contain the term t_i.
dl_j is the document length (in bytes) of d_j.
$avdl$ is the average document length of the collection.

The Okapi relevance score of a document d_j for a query q is given by the equation below, where k_1 (between 1.0–2.0), b (usually 0.75) ,and k_2 (between 1–1000) are parameters:

$$\text{okapi}(d_j, q) = \sum_{t_i \in q, d_j} \ln \frac{N - df_i + 0.5}{df_i + 0.5} \times \frac{(k_1+1) f_{ij}}{k_1\left(1 - b + b\frac{dl_j}{avdl}\right) + f_{ij}} \times \frac{(k_2+1) f_{iq}}{k_2 + f_{iq}},$$

4) Semantic Model

However sophisticated the above statistical models become, they can miss many relevant documents because those models do not capture the complete meaning or information need conveyed by a user’s query. In semantic models, the process of matching documents to a given query is based on concept level and semantic matching instead of index term (keyword) matching. This allows retrieval of relevant documents that share meaningful associations with other documents in the query result, even when these associations are not inherently observed or statistically captured.

Semantic approaches include different levels of analysis, such as morphological, syntactic, and semantic analysis, to retrieve documents more effectively. In **morphological analysis**, roots and affixes are analyzed to determine the parts of speech (nouns, verbs, adjectives, and so on) of the words. Following morphological analysis, **syntactic analysis** follows to parse and analyze complete phrases in documents. Finally, the semantic methods have to resolve word ambiguities and/or generate relevant synonyms based on the **semantic relationships** between levels of structural entities in documents (words, paragraphs, pages, or entire documents).

The development of a sophisticated semantic system requires complex knowledge bases of semantic information as well as retrieval heuristics. These systems often require techniques from artificial intelligence and expert systems. Knowledge bases like Cyc and WordNet have been developed for use in *knowledge-based IR systems* based on semantic models. The Cyc knowledge base, for example, is a representation of a vast quantity of commonsense knowledge about assertions (over 2.5 million facts and rules) interrelating more than 155,000 concepts for reasoning about the objects and events of everyday life. WordNet is an extensive thesaurus (over 115,000 concepts) that is very popular and is used by many systems and is under continuous development.

5.18. TYPES OF QUERIES IN IR SYSTEMS

Different keywords are associated with the document set during the process of indexing. These keywords generally consist of words, phrases, and other characterizations of documents such as date created, author names, and type of document. They are used by an IR system to build an inverted index, which is then consulted during the search. The queries formulated by users are compared to the set of index keywords. Most IR systems also allow the use of Boolean and other operators to build a complex query. The query language with these operators enriches the expressiveness of a user's information need.

1) Keyword Queries:

Keyword-based queries are the simplest and most commonly used forms of IR queries: the user just enters keyword combinations to retrieve documents. The query keyword terms are implicitly connected by a logical AND operator. A query such as 'database concepts' retrieves documents that contain both the words 'data-base' and 'concepts' at the top of the retrieved results. In addition, most systems also retrieve documents that contain only 'database' or only 'concepts' in their text. Some systems remove most commonly occurring words (such as *a*, *the*, *of*, and so on, called **stopwords**) as a preprocessing step before sending the filtered query keywords to the IR engine. Most IR systems do not pay attention to the ordering of these words in the query. All retrieval models provide support for keyword queries.

2) Boolean Queries:

Some IR systems allow using the AND, OR, NOT, ( ), + , and – Boolean operators in combinations of keyword formulations. AND requires that both terms be found. OR lets either term be found. NOT means any record containing the second term will be excluded. '( )' means the Boolean operators can be nested using parentheses. '+' is equivalent to AND, requiring the term; the '+' should be placed directly in front of the search term. '–' is equivalent to AND NOT and means to exclude the term; the '–' should be placed directly in front of the search term not wanted. Complex Boolean queries can be built out of these operators and their combinations, and they are evaluated according to the classical rules of Boolean algebra. No ranking is possible, because a document either satisfies such a query (is "relevant") or does not satisfy it (is "nonrelevant"). A document is retrieved for a Boolean query if the query is logically true as an exact match in the document. Users generally do not use combinations of these complex Boolean operators, and IR systems support a restricted version of these set operators. Boolean retrieval models can directly sup-port different Boolean operator implementations for these kinds of queries.

3) Phrase Queries

When documents are represented using an inverted keyword index for searching, the relative order of the terms in the document is lost. In order to perform exact phrase retrieval, these phrases should be encoded in the inverted index or implemented differently (with relative positions of word occurrences in documents). A phrase query consists of a sequence of words that makes up a phrase. The phrase is generally enclosed within double quotes. Each retrieved document must contain at least one instance of the exact phrase. Phrase searching is a more restricted and specific version of proximity searching. For example, a phrase searching query could be 'conceptual database design'. If phrases are indexed by the retrieval model, any retrieval model can be used for these query types. A phrase thesaurus may also be used in semantic models for fast dictionary searching for phrases.

4) Proximity Queries:

Proximity search refers to a search that accounts for how close within a record multiple terms should be to each other. The most commonly used proximity search option is a phrase search that requires terms to be in the exact order. Other proximity operators can specify how close terms should be to each other. Some will also specify the order of the search terms. Each search engine can define proximity operators differently, and the search engines use various operator names such as NEAR, ADJ(adjacent), or AFTER. In some cases, a sequence of single words is given, together with a maximum allowed distance between them. Vector space models that also maintain information about positions and offsets of tokens (words) have robust implementations for this query type. However, providing support for complex proximity operators becomes computationally expensive because it requires the time-consuming preprocessing of documents, and is thus suitable for smaller document collections rather than for the Web.

5) Wildcard Queries:

Wildcard searching is generally meant to support regular expressions and pattern matching-based searching in text. In IR systems, certain kinds of wildcard search support may be implemented—usually words with any trailing characters (for example, 'data*' would retrieve *data*, *database*, *datapoint*, *dataset*, and so on). Providing support for wildcard searches in IR systems involves preprocessing over-head and is not considered worth the cost by many Web search engines today. Retrieval models do not directly provide support for this query type.

6) Natural Language Queries

There are a few natural language search engines that aim to understand the structure and meaning of queries written in natural language text, generally as a question or narrative. This is an active area of research that employs techniques like shallow semantic parsing of text, or query reformulations based on natural language under-standing. The system tries to formulate answers for such queries from retrieved results. Some search systems are starting to provide natural language interfaces to provide answers to specific types of questions, such as definition and factoid questions, which ask for definitions of technical terms or common facts that can be retrieved from specialized databases. Such questions are usually easier to answer because there are strong linguistic patterns giving clues to specific types of sentences—for example, 'defined as' or 'refers to'. Semantic models can provide support for this query type.

CS8492 / DATABASE MANAGEMENT SYSTEMS

UNIT-1

PART-A

1. **What is the purpose of Database Management System? Nov/ Dec 2014**

A DBMS is a software for creating and managing databases. It provides users with a systematic way to create, retrieve, update and manage data. It is a middleware between the database which store all the data and the users or applications which need to interact with that stored database. A DBMS can limit what data the end user sees, as well as how that end user can view the data, providing many views of a single database schema

2. **What are the characteristics that distinguish the database approach with the File – based approach? April/ May 2015**

In File System, files are used to store data while, collections of databases are utilized for the storage of data in DBMS. Although File System and DBMS are two ways of managing data, DBMS clearly has many advantages over File Systems. Typically when using a File System, most tasks such as storage, retrieval and search are done manually and it is quite tedious whereas a DBMS will provide automated methods to complete these tasks. Because of this reason, using a File System will lead to problems like data integrity, data inconsistency and data security, but these problems could be avoided by using a DBMS. Unlike File System, DBMS are efficient because reading line by line is not required and certain control mechanisms are in place.

3. **Is it possible for several attributes to have the same domain? Illustrate your answer with suitable example. Nov/ Dec 2015**

A domain is a pool of values from which the values of specific attributes of specific relations are taken. For example, the domain dept is a set of all possible dept names and the domain emp_name is a set of all employee names. Thus each and every attribute has its own domain. Hence it is not possible for several attributes to have the same domain.

4. **What are the disadvantages of file processing system? May/ June 2016**

The disadvantages of file processing systems are

a) Data redundancy and inconsistency b) Difficulty in accessing data c) Data isolation d) Integrity problems e) Atomicity problems f) Concurrent access anomalies

5. **Differentiate File System with Database Management system. Nov/ Dec 2016**

S. No	File System	Database Management system
1	files are used to store data	collections of databases are utilized for the storage of data
2	most tasks such as storage, retrieval and search are done manually and it is quite tedious	provide automated methods to complete these tasks
3	File System will lead to problems like data integrity, data inconsistency and data security	these problems could be avoided by using a DBMS
4	Each application has its own private files resulting in considerable amount of redundancy of the stored data. Thus storage space is wasted	It has centralized control over the database. Hence, data is shared and redundancy is avoided
5	There is no centralized control over the database. Hence concurrent access results in data inconsistency	It has centralized control over the data. Hence concurrent access to the database doesn't result in data inconsistency

6. **Distinguish key and super key. Nov Dec 2017**

   Minimal column which are sufficient to identify row is **primary key**. **Super key** also use for identify row but one **super key** may be contain more than 1 **primary key** or combination of **primary keys** known as **super key**. Both used for uniquely identify of row. Table contain more than 1 candidate **key** but only 1 **primary key**

7. **What are the advantages of using a DBMS?**

a) Controlling redundancy b) Restricting unauthorized access c) Providing multiple user interfaces d) Enforcing integrity constraints. e) Providing backup and recovery

8. **Define instance and schema?**

Instance: Collection of data stored in the data base at a particular moment is called an Instance of the database.

Schema: The overall design of the data base is called the data base schema.

9. **Define the terms 1) Physical schema 2) logical schema.**

Physical schema: The physical schema describes the database design at the physical level, which is the lowest level of abstraction describing how the data are actually stored.

Logical schema: The logical schema describes the database design at the logical level, which describes what data are stored in the database and what relationship exists among the data.

10. **What is storage manager? List the components.**

A storage manager is a program module that provides the interface between the low level data

Stored in a database and the application programs and queries submitted to the system.

The storage manager components include

a) Authorization and integrity manager b) Transaction manager c) File manager d) Buffer manager

11. **What is a data dictionary?**

A data dictionary is a data structure which stores Meta data about the structure of the database ie. The schema of the database.

12. **What are attributes? Give examples.**

An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set.

Example: possible attributes of customer entity are customer name, customer id, Customer Street, customer city.

13. **What is relationship? Give examples**

A relationship is an association among several entities.

Example: A depositor relationship associates a customer with each account that he/she has.

14. **Define the term Relationship set**.

Relationship set: The set of all relationships of the same type is termed as a relationship set.

15. Define null values

In some cases a particular entity may not have an applicable value for an attribute or if we do not know the value of an attribute for a particular entity. In these cases null value is used.

16. List the role of DBA.

The person who has central control over the system is called database administrator. The functions of the DBA include the following:

- Schema definition
- Storage structure and access-method definition
- Schema and physical-organization modification
- Granting of authorization for data access
- Integrity-constraint specification

17. Differentiate Static SQL and Dynamic SQL. Nov/ Dec 2014, April/ May 2015, Nov/ Dec 2015, Nov/ Dec 2016

Static SQL: Static SQL statements are SQL instructions that are a part of the language syntax. It can be used directly in the source code as normal procedural instructions.

Dynamic SQL: It is a programming technique that enables to build SQL statements dynamically at runtime.

18. Give a brief description in DCL commands. NOV/DEC 2014

It is a computer language and a subset of SQL, used to control access to data in a database.
Examples of DCL commands include:

GRANT: used to allow specified users to perform specified tasks.
REVOKE: used to cancel previously granted permission

19. Why does SQL allow duplicate tuples in a table or in a query result? Nov/ Dec 2015

SQL usually treats a table not as a set but rather as a multiset. Duplicate tuples can appear more than once in a table and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries for the following reasons.

i. Duplicate elimination is an expensive operation one way to implement it, is to sort the tuples first and then eliminate duplicates.

ii. The users may want to see duplicate tuples in the result of a query

When an aggregate function is applied to tuples in most cases we do not want to eliminate duplicates

20. Name the categories of SQL Command. May/ June 2016

a. data definition language
b. data manipulation language

c. data control language
d. transaction control language

21. What is Data Definition Language? Give example. Nov/ Dec 2016

It is used to define relational database of a system. It creates, changes and removes a table structure. Ex. CREATE, ALTER, DROP, RENAME and TRUNCATE.

22. Define Data Manipulation Language.

DML: A data manipulation language is a language that enables users to insert, modify and delete the data in the database.

Ex. Insert, delete, modify

23. List the SQL statements used for Transaction Control.

a. Commit
b. Rollback
c. Save point
d. Set Transaction

24. Define database objects.

A database object is any defined object in the database that is used to store or reference data. Some examples include tables, views, clusters, sequences, indexes and synonyms.

25. Name the different types of joins supported in SQL.

a. Inner join
b. Outer join
    1. Left Outer Join
    2. Right Outer Join
    3. Full Outer Join
c. Natural join

26. What is a trigger in SQL?

A trigger is a statement that the system executes automatically as a side effect of a modification to the database.

27. What are primary keys?

A primary key is one or more columns in a table used to uniquely identify each row in the table. Its values must not be null and must be unique across the column.

28. Explain the basic structure of an SQL expression.

An SQL Expression has three clauses: select, from and where.

α. Select- used to list the attributes desired in the result of a query
β. From- lists the relations to be scanned
χ. Where- predicate involving attributes in the from clause

select A1, A2….An from r1, r2,….rn where p;

29. Write a SQL Statement to find the names & loan numbers of all customers who have a loan at Chennai branch from the following relations.

i). Loan (Loan _ no, Branch _ name, amount)
ii). Branch (Branch _ name, Branch _city, Assets)

select Loan_no from Loan, Branch where Loan.Branch_name = Branch.Branch_name and Branch_city ='Chennai';

Part B

1. Consider a student registration database comprising of the below given table schema .

student file

student number , student name , address , telephone

course file

course number , description , hours , professor number

professor file

professor number , name , office

registration file

student number , course number , date

consider a suitable example of tuples/records for the above mentioned tables and write DML , statements (SQL) to answer for the queries listed below
i) Which courses does a specific professor teach ?
ii) What courses are taught by two specific professors?
iii) Who teaches a specific courses and where is his/her office?
iv)For specific student number in which courses is the student registered and what is his/her name?
v) who are the professors for a specific student ?
vi) Who are the students registered in specific courses?

2) Explain the following with examples:
i) DDL
ii) DML
ii) Embedded SQL

3) Assume the following table :
degree(degcode,name,subject)
candidate(seat no, degcode,semesrer,month,year,result)
Marks(seatno,degcode,semester,month,year,papcode,marks)
Degcode-degree code.Name-name of the degree(MSC.MCOM)
SUBJECT_subject of courses Eg. Phy , pap code –paper code eg.A1
Solve the following queries using SQL
i) Write a SELECT statement to display all the degree codes are there in the candidate table but not present in degree table in the order of degcode.
ii) Write a SELECT statement to display the name of all the candidates who have got less than 40 marks in exactly 2 subjects .
iii) Write a SELECT statement to display the name, subject and number of the candidate for all degrees in which there are less than 5 candidates.
iv) Write a SELECT statement to display the names of all the candidates who have got highest total marks in MSc., (Maths).

4) Describe the GRANT functions and explain how it relates to security. what types of privileges may be granted? How rights could be revoked ?

5) With the help of a neat block diagram , explain the basic architecture of a data base management system?
6) Describe the six clauses in the syntax of an sql query and show what type of constructs can be specified in each of the six clauses.which of the six clauses are required and which are optional?

7) List the operations of relational algebra and the purpose of each with example.
8) consider the relation schema given in figure 1.design and draw an ER diagram that capture the information of this schema.

Employee(empno,name,office,age)
Books(isbn,tittle ,authors,publisher)
Loan(empno,isbn,date)

write the following queries in relational algebra and SQL.
i)find the name of employees who have borrowed a book published by McGraw-Hill.
ii) find the name of employees who have borrowed all books published by McGraw-Hill.

9) Write the DDL,DML,DCL commands for the students database.Which contains
Student details :name,id,DOB,branch,DOJ.
Course details:Course name,Course id,Stud.id,Faculty name,id,marks.

10) Differentiate between foreign key constraints and referential integrity constraints with suitable examples?

11) Justify the need of embedded SQL .consider the relation student(studentno,name,mark and grade).Write embedded dynamic SQL statements in C language to retrieve all the student's records whose marks more than 90.

12) Explain about functions and procedures in SQL.

13) Discuss about triggers.

CS8492 / DATABASE MANAGEMENT SYSTEMS

UNIT-2

PART –A

1. Why 4NF in Normal Form is more desirable than BCNF? Nov/ Dec 2014

4NF is an improvement over BCNF since it eliminates another form of undesirable structure MVD by taking two projections.

R.A -> R.B

A relation R is in fourth normal form if whenever there exists a MVD in relation R.A ->-> B then all attributes of R are also functionally dependent on A

4NF is stronger than BCNF, ie, any 4NF relation is necessary in BCNF

2. Define functional dependency. April/ May 2015

Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. If R is a relation with attributes X and Y, a functional dependency between the attributes is represented as X->Y, which specifies Y is functionally dependent on X.

3. State the anomalies of 1NF. Nov/ Dec 2015

INSERT. Certain student with SID 5 got admission in a different campus (say) Karachi cannot be added until the student registers for a course.

DELETE. If student graduates and his/her corresponding record is deleted, then all information about that student is lost.

UPDATE. If student migrates from Islamabad campus to Lahore campus (say) SID = 1, then six rows would have to be updated with this new information

4. Explain entity relationship model. May/ June 2016

The entity relationship model is a collection of basic objects called entities and relationship among those objects. An entity is a thing or object in the real world that is distinguishable from other objects.

5. What is a weak entity? Give example. Nov/ Dec 2016

An entity set may not have sufficient attributes to form a primary key, and its primary key compromises of its partial key and primary key of its parent entity, then it is said to be Weak Entity set

6. What are the desirable properties of decomposition? Nov/ Dec 2017

a. Lossless Join: this property ensures that any instance of the original relation can be identified from corresponding instances in the smaller relation.

b. Dependency Preservation: this property ensures that a constraint on the original relation can be maintained by simply enforcing some constraint on each of the smaller relation.

7. What is meant by normalization of data?
~~It is~~ a process of analyzing the given relation schemas based on their Functional Dependencies (FDs) and primary key to achieve the properties

   i) Minimizing redundancy
   ii) Minimizing insertion, deletion and updating anomalies

8. Distinguish key and super key. Nov Dec 2017

Minimal column which are sufficient to identify row is **primary key**. **Super key** also use for identify row but one **super key** may be contain more than 1 **primary key** or combination of **primary keys** known as **super key**. Both used for uniquely identify of row. Table contain more than 1 candidate **key** but only 1 **primary key**

9. Define instance and schema?

Instance: Collection of data stored in the data base at a particular moment is called an Instance of the database.

Schema: The overall design of the data base is called the data base schema.

10. Define the terms 1) Physical schema 2) logical schema.

Physical schema: The physical schema describes the database design at the physical level, which is the lowest level of abstraction describing how the data are actually stored.

Logical schema: The logical schema describes the database design at the logical level, which describes what data are stored in the database and what relationship exists among the data.

11. What are attributes? Give examples.

An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set.

Example: possible attributes of customer entity are customer name, customer id, Customer Street, customer city.

12. What is relationship? Give examples

A relationship is an association among several entities.

Example: A depositor relationship associates a customer with each account that he/she has.

13. **. Define the term Relationship set**.

Relationship set: The set of all relationships of the same type is termed as a relationship set.

14. Define null values

In some cases a particular entity may not have an applicable value for an attribute or if we do not know the value of an attribute for a particular entity. In these cases null value is used.

15. Consider the following relation :
    R (A, B, C, D, E)
    The primary key of the relation is AB. The following functional dependencies hold :
    $A \rightarrow C$
    $B \rightarrow D$
    $AB \rightarrow E$.
    Is the above relation in second normal form?

16. Consider the following relation :
    R(A, B, C, D)
    The primary key of the relation is A. The following functional dependencies hold :
    $A \rightarrow B,C$
    $B \rightarrow D$
    Is the above relation in third normal form?

17. What is the need for normalization?

Normalization is the process of removing redundant data from your tables in order to improve storage efficiency, data integrity and scalability. This improvement is balanced against an increase in complexity and potential performance losses from the joining of the normalized tables at query-time. There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a table). Both of these are worthy goals as they reduce the amount of space a database consumes and ensure that data is logically stored.

18. Give an example of a relation schema R and a set of dependencies such that R is in BCNF, but not in 4NF.
19. Why are certain functional dependencies called as trivial functional dependencies?
20. A relation R={A,B,C,D} has FD's F={ $A \rightarrow D$, $D \rightarrow C$, $C \rightarrow AB$} is R is in 3NF?

Part-B

1. Distinguish between lossless-join decomposition and dependency preserving decomposition.
2. Discuss the correspondence between the ER model construct and the relational model constructs. Show how each ER model construct can be mapped to the relation model. discuss the option for mapping EER model construct.
3. Consider the relation schema given in figure. Design and draw an ER diagram that capture the information of this schema.
    Employee(empno,name,office,age)
    Books(isbn,tittle ,authors,publisher)
    Loan(empno,isbn,date)

4. A car rental company maintains a data base for all vehicles in its current fleet. for all vehicles, it includes the vehicle identification number license number, manufacturer, model, data of purchase and color. Special data are included for certain types of vehicles?
    Trucks: cargo capacity
    Sports cars: horse power, renter age requirement

    Vans: number of passengers
    Off-road vehicles : ground clearance, drivetrain(four or two wheel drive)

    Construct an ER model for the car rental company database.

5. Draw E-R diagram for banking system.
6. Draw E-R diagram for the "Restaurant Menu ordering system", which will facilitate the food item ordering and services with in a restaurant. the entire restaurant scenario is detailed as follows. The customer is able to view the food items menu, call the waiter, place orders and obtain the final bill through the computer kept in their table. The waiters through their wireless tablet PC are able to initialize a table for customers, control the table functions to assist customers, orders, send order to food preparation staff(chef) and finalize the customers bill. The food preparation staffs(chefs),with their touch-display interfaces to the system, are able to view orders sent to kitchen by the waiters. during preparation, they are able to let the waiter know the status of each

item, and can send notifications when items are completed. The system should have fill accountability and logging facilities, and should support supervisor actions to account for exceptional circumstances, such as a meal being refunded or walked out on.

7. What are normal forms? Explain the type of normal forms with suitable example. **Nov/ Dec 2014**

   (or)

   State the need for Normalization of a database and explain the various Normal Forms (1st, 2nd, and 3rd, BCNF, 4th, 5th and domain-key) with suitable examples. **May/ June 2015**
8. Explain non loss decomposition and functional dependencies with suitable example.
9. Explain the boyce-code normal form with an example. Also state how it differs from that of 3NF.
10. Discuss join dependencies and fifth normal form, and explain why 5NF?
11. Construct an ER diagram for car insurance company that has a set of customers each of whom owns one or more cars. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars, and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the premium was made. **(7) Nov/ Dec 2016**
12. Describe about the multi_valued dependencies and fourth normal form with suitable example.
13. Explain boyce codd normal form and fourth norml forms with suitable example.
14. Explain the principles of
    (i) Loss less join decomposition
    (ii) Join dependencies
    (iii) Fifth normal form.

CS8492 / DATABASE MANAGEMENT SYSTEMS

UNIT-III

PART-A

1. What is meant by concurrency control? Nov/ Dec 2015

In a multiprogramming environment, multiple transactions can be executed simultaneously. The system must control the interaction among the concurrent transactions. This control is achieved through one of concurrency control schemes. The concurrency control schemes are based on the serializability property.

2. Give an example of two phase commit protocol. Nov/ Dec 2015

```
lock-X(A);      // Growing Phase
read(A);
A=A-50;
Write(A);
lock-X(B);      // Growing Phase
read(B);
B=B+50;
Wtite(B);
unlock(A);  // Shrinking Phase
unlock(B);    // Shrinking Phase
```

3. What are the properties of transaction? April/ May 2016, Nov/ Dec 2014, April/ May 2015 Collection of operations that form a single logical unit of work are called transaction.

ACID Properties

1) Atomicity.

Either all operations of the transaction are properly reflected in the database or none are.

2) Consistency.

Execution of a transaction in isolation preserves the consistency of the database.

3) Isolation.

Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

4) Durability.

After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

4. **Differentiate Strict Two Phase Locking Protocol and Rigorous Two Phase Locking Protocol. April/ May 2016**

   Strict Two Phase Locking Protocol

   All exclusive locks are held by transaction T and are released when T commits and not before.

   Rigorous Two Phase Locking Protocol

   All locks both exclusive and shared locks held by transaction T are released when T commits and not before.

5. **What is Serializability (Serializable Schedule)? How is it tested? Nov/ Dec 2014, Nov/ Dec 2016, April/ May 2017**

   Serializability is a schedule that has the same effect on the database as a serial schedule. It is tested using two techniques: View Serializability and Conflict Serializability.

6. **List the four conditions for deadlock. Nov/ Dec 2016**
   i. mutual exclusion
   ii. hold and wait
   iii. no pre-emption
   iv. circular wait

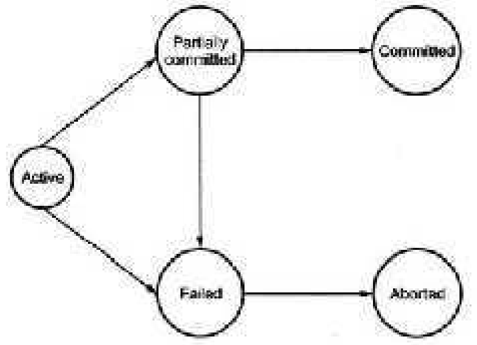
7. **What type of locking is needed for insert and delete operations? April/ May 2017**

   Shared lock is obtained when Data item can only be read. But data item can be both read as well as written when exclusive lock is obtained. So exclusive lock is needed for insert and delete operations

8. **Brief about cascading rollback.**

   The Phenomenon in which a single transaction failure leads to a series of transaction roll backs, is called cascading rollback.

9. **With neat diagram show the states of a transaction.**



10. **What is transaction-management component?**

    Ensuring atomicity is the responsibility of the database system itself specifically; it is handled by a component called the transaction-management component.

11. **When two operations in schedule are said to be conflict?**
    i) Two operation belong to different transaction
    ii) Two operation access the same item x
    iii) At least one of the operation is write-item(x)

12. **Define cascading rollback**

A **cascading rollback** occurs in database systems when a transaction (T1) causes a failure and a **rollback** must be performed. Other transactions dependent on T1's actions must also be rollbacked due to T1's failure, thus causing a **cascading** effect. That is, one transaction's failure causes many to fail

13. List out the two-phase locking.

1) Growing phase: A transaction may obtain locks but may not replace any lock.
2) Shrinking phase: A transition may release lock but may not obtain any new locks.

14. Define lock.

Lock is variable associated with a data item. Lock is used as a means of synchronizing the access by concurrent transaction to the database item.

15. Define timestamp

Timestamp are typically based on the order in which transition are stared.

16. Define timestamp based protocol.

Timestamp based protocol ensures serializability. It selects an ordering among transactions in advance using time stamps.

17. If deadlock is avoided by deadlock avoidance scheme, is starvation still possible? Explain your answer.

A transaction may become the victim of deadlock-prevention rollback arbitrarily many times, thus creating a potential starvation situation.

18. When are two scheduled conflict equivalent?

If a schedule S can be transformed into a schedule S` by a series of swaps of non-conflicting instructions, we say that S and S` are conflict equivalent.

19. Why is it necessary to have control of concurrent execution of transactions? How is it made possible?

The system must control the interaction among the concurrent transactions. This control is achieved through one of the concurrency control schemes. The concurrency control are based on the serializability property.

20. What are the two operations that access data in transaction?

Read(x)- transfer data item x from database.
Write(x)- transfer data item x from the local buffer.

21. List the two commonly used Concurrency Control techniques.

1) Lock based protocols
2) Time stamp based protocol

22. What are the pitfalls of lock-based protocols?

23. List the SQL statements used for transaction control.

24. List down the SQL facilities for concurrency.

Part B

1. What is concurrency? Explain it in terms of locking mechanisms and two phase commit protocol. ***Nov/ Dec 2014***
2. Write short notes on Transaction concept and schedules. ***Nov/ Dec 2014***
3. Write short notes on Deadlock. ***Nov/ Dec 2014, (16) Nov/ Dec 2015, (6) Nov/ Dec 2016***
4. What is concurrency control? How is it implemented in DBMS? Illustrate with a suitable example. ***Nov/ Dec 2015***
5. Discuss View Serializability and conflict Serializability. ***Nov/ Dec 2015***
6. Explain briefly about two phase commit. ***May/ June 2016, (8) May/ June 2015, (6) Nov/ Dec 2016***
7. Explain about locking protocols. ***May/ June 2016***

8. Consider the following schedules. The actions are listed in the order they are scheduled and prefixed with the transaction name. ***May/ June 2015***

   S1: T1: R(X), T2: R(X), T1: W(Y), T2: W(Y), T1: R(Y), T2: R(Y)
   S2: T3: W(X), T1: R(X), T1: W(Y), T2: R(X), T2: W(Z), T3: R(Z) For each of the schedules answer the following questions:

(i) What is the precedence graph for the schedule?
(ii) Is the schedule conflict serializable? If so, what are all the conflict equivalent serial schedule?
(iii) Is the schedule view serializable? If so, what are all the view equivalent serial schedule?

9. Discuss the violations caused by each of the following: Dirty read, non-repeatable read and phantoms with suitable example. ***April/ May 2017***

10. Consider the following two transactions:
T1: read(A);
Read(B);
If A=0 then B=B+1;
Write(B);

T2: read(B);
Read(A);
If B=0 then A=A+1;
Write(A);

Add lock and unlock instructions to transactions T1 and T2 so that they observe the two phase locking protocol. Can the execution of these transactions result in a deadlock? ***Nov/ Dec 2016***

11. Consider the following extension to the tree locking protocol, which allows both shared and exclusive locks:

- A transaction can be either a read only transaction in which case it can request only shared locks or an update transaction, in which case it can request only exclusive locks.

- Each transaction must follow the rules of the tree protocol. Read-only transactions may lock any data item first, whereas update transaction must lock the root first.

Show that the protocol ensures serializability and deadlock freedom ***Nov/ Dec 2016***
12. Elobarate the SQL facilities for concurrency and recovery.
13. Discuss about transaction recovery.
14. Explain various isolation levels in concurreuncy.

Part-A

1. **Differentiate Static and Dynamic Hashing. Nov/ Dec 2014, May/ June 2015, Nov/ Dec 2015**

S.No	Static Hashing	Dynamic Hashing
1	Number of buckets is fixed	Number of buckets is not fixed
2	Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows. If database shrinks, again space will be wasted	Good for database that grows and shrinks in size
3	One option is periodic re-organization of the file with a new hash function, but it is very expensive	Allows the hash function to be modified dynamically

2. **Give an example of a join that is not a simple equi join for which partitioned parallelism can be used. Nov/ Dec 2015**
   r, s -> relations
   A, B, C -> attributes
   (r.A=s.B)^(r.a<s.c)s

Here we have extra conditions which can be checked after the join. Hence partitioned parallelism is used.

3. **What is meant by garbage collection? May/ June 2016**

It is a process of destroying objects that are no longer referenced and freeing the resources those objects used.

4. **Define software and hardware RAID system. May/ June 2016**

RAID has an arrangement of several independent disks that are organized to improve reliability through parity schemes and error correcting codes and at the same time increase performance through data striping.

5. **List out the mechanisms to avoid collisions during hashing. Nov/ Dec 2016**
   a. Open Hashing (Separate chaining)

Open Hashing, is a technique in which the data is not directly stored at the hash key index (*k)* of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored.

b. Closed hashing (open Addressing)

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found.

6. What are the disadvantages of B tree over B+ tree? Nov/ Dec 2016

1) In B+-Trees, all records are stored at the leaf level while keys and child pointers are stored in interior nodes (non-leaf nodes). This maximizes the branching factor of the internal nodes.
2) Because B+ trees don't have data associated with interior nodes, more keys can fit on a page of memory. Therefore, it will require fewer cache misses in order to access data that is on a leaf node.
3) The leaf nodes of B+ trees are linked, so doing a full scan of all objects in a tree requires just one linear pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree. This full-tree traversal will likely involve more cache misses than the linear traversal of B+ leaves.

7. Define replication transparency. May/ June 2017

A **transparency** is some aspect of the **distributed system** that is hidden from the user. A **transparency** is provided by including some set of mechanisms in the **distributed system** at a layer below the interface where the **transparency** is required. **Replication transparency** is the term used to describe the fact that the user should be unaware that data is **replicated**.
on.

8. What are two approaches to store a relation in the distributed database? May '04

a. Replications ii. Fragmentation

9. What are the advantages of distributed database? Dec'04, May'08

i. Sharing data
ii. Autonomy
iii. Availability

10. List out the reasons for the development of distributed database. May'06

i. In the centralized system, data is stored on a single computer. If that computer fails, complete system fails.
ii. In client-server system also the data is stored on server. If server fails, the complete system fails.
iii. Thus, there was need of a system in which data is distributed and replicated among several computers.

11. Give the measures of quality of a disk.

1. Capacity
2. Access time
3. Seek time
4. Data transfer rate
5. Reliability
6. Rotational latency time

12. What are the types of storage devices?

Primary storage, Secondary storage, Tertiary storage

13. Define access time and seek time.

Access time is the time from when a read or write request is issued to when data transfer begins. The time for repositioning the arm is called the seek time and it increases with the distance that the arm is called the **seek time.**

14. Define rotational latency time.

The time spent waiting for the sector to be accessed to appear under the head is called the rotational latency time.

15. What is the use of a slotted-page structure and what is the information present in the header?

The slotted-page structure is used for organizing records within a single block.
The header contains the following information.

i. The number of record entries in the header.
ii. The end of free space.

An array whose entries contain the location and size of each record.

16. **What are the advantages of distributed system?**
    1) Sharing data
    2) Autonomy
    3) Availability

17. Which are the factors to be considered for the evaluation of indexing and hashing techniques?

18. What are the advantages and disadvantages of indexed sequential file?

Part-A

1. What is a B+ tree index file in DBMS?
2. Examine the need for query Optimization.
3. Explain ―Query Optimization‖ with your own database.
4. Point out the methods for implementing JOINs.
5. Define software and hardware RAID systems.
6. Illustrate the need for RAID.
7. Distinguish between fixed length records and variable length records?
8. When is it preferable to use a dense index rather than a sparse index? Explain your answer.
9. List the different Hashing techniques.
10. Give the procedure to reduce the occurrences of bucket overflows in a hash file organization?
11. What are ordered indices with example?
12. Contrast sparse index and dense index
13. Outline the steps involved in query processing.
14. Point out the disadvantages of B Tree over B+ Tree
15. Differentiate between Static and Dynamic Hashing
16. List out the mechanisms to avoid collision during hashing.
17. What are select operations?
18. Assess why we need to go for cost estimation in query optimization.
19. What is hash function? Give example.

20. Prepare the factors to be considered for the evaluation of indexing and hashing techniques?

PART-B

1. Explain about RAID system. How does it improve performance and reliability.

2. Discuss the level 3 and level 4 of RAID.

3.(i) Describe the index schemas used in databases.(07)

(ii) Since indices speed query processing, why might they not be kept on several search keys? List as many reasons as possible.(06)

4. Describe the different types of file organization? Explain using a sketch of each of them with their advantages and disadvantages.(13)

5.(i) Describe the ordered indices with example.(10)

(ii)Describe the different methods of implementing variable length records. (03)

6. Give a detailed description about Query Processing and Optimization. Explain the cost estimation of Query Optimization.

7.(i)Discuss briefly about B+ tree index file with example. (07)

(ii) How does a B-tree differ from a B+ - tree? why is a B+-tree usually preferred as an access structure to a data file?(06)

8. (i) Illustrate indexing techniques with suitable examples (07)

(ii) Write notes on Hashing.(06)

9. Illustrate the Join order optimization and Heuristic optimization algorithms.(13)

10. What is meant by semantic query optimization? How does it differ from other

query optimization technique? Give example. (13)

11. Examine the algorithms for SELECT and JOIN

.

12. Examine the catalog information for cost estimation for selection and sorting

operation in database. (13)

13. Describe about B tree index file with example.(13)

14.Explain the distinction between static and dynamic hashing. Discuss the relative

merits of each technique in database applications. (13)

15. Develop a B+ tree to insert the following

5,3,4,9,7,15,14,21,22,23. (13)

16. Construct B tree and B^+ tree to insert the following key values(the order of the tree is three) 32,11,15,13,7,22,15,44,67,4.(15)

17.The following key values are organized in an extendable hashing technique. 1 3 5 8 9 12 17 28 Show the extendable hash structure for this file if the hash function is h(x)=x mod 8 and buckets can hold three records.

Unit-5
Part-A

1. **What is crawling and indexing the Web? Nov/ Dec 2014**
   Web crawling is the process by which we gather pages from the web, in order to index them and support a search engine. The crawler begins with one or more URLs that constitute a seed set. It picks a URL from this seed set, then fetches the web page at that URL. **Indexing** is the process of adding webpages into the search
2. **What is Relevance Ranking? Nov/ Dec 2014**
   Relevancy ranking is the process of sorting the document results so that those documents which are most likely to be relevant to your query are shown at the top.
3. **List the types of privileges used in database access control. Nov/ Dec 2015**
   i. ***System privileges:*** these privileges are related to the access of the database
   ii. ***Object privileges:*** these privileges are focused on a particular database object
4. **Define a distributed database management System. Nov/ Dec 2016**
   A DDBMS (distributed database management system) is a centralized application that manages a distributed database as if it were all stored on the same computer. The DDBMS synchronizes all the data periodically, and in cases where multiple users must access the same data, ensures that updates and deletes performed on the data at one location will be automatically reflected in the data stored elsewhere.
5. **How does the concept of an object in the object oriented model differ from the concept of an entity in the entity relationship model? Nov/ Dec 2016**
   An **entity** is something that exists in itself, actually or potentially, concretely or abstractly, physically or not. It needs not be of material existence.
   In Object Oriented Model, an **object** is a location in memory having a value and possibly referenced by an identifier.
   An entity is an abstract concept that's typically represented by a table in a database schema. The term object usually refers to in-memory data structures.
   A database object is represented as database, schema, table, column, primary key, and foreign key while a database Entity is a concept or object of importance about which data must be captured.
6. **Explain the three phase commit protocol.**
   It is an extension of two phase commit protocol for avoiding blocking problem of 2PC protocol under certain assumptions:
   1. No network partition occurs in the system
   2. Not more than k sites fail
7. **Write the difference between XML Schema and XML DTD**
   i. XML schemas are written in XML while DTD are derived from SGML syntax.
   ii. XML schemas define data types for elements and attributes while DTD doesn't support data types.
   iii. XML schemas allow support for namespaces while DTD does not.
   iv. XML schemas define number and order of child elements, while DTD does not.
   v. XML schemas can be manipulated on your own with XML DOM but it is not possible in case of DTD.
   vi. Using XML schema user need not to learn a new language but working with DTD is difficult for a user.

vii. XML schema provides secure data communication i.e sender can describe the data in a way that receiver will understand, but in case of DTD data can be misunderstood by the receiver.

viii. XML schemas are extensible while DTD is not extensible.

Part-A

1. Compare information retrieval Vs DBMS.

2. Give the architecture models in distributed database.
3. Show how are transaction performed in Object oriented database?
4. List the Operations performed in transaction?

5. Define Information Retrieval system. Prepare how it is differs from database

system.
6. Define distributed database management system.
7. Demonstrate the meaning of homogenous and heterogeneous DDBMS

8. What are ODL and OQL.
9. List out the IR models.
10. Tell how spatial databases are more helpful than active database?

11. Differentiate XML schema and DTD.

12. Discuss Relevance Ranking.
13. State the function of XML schema.
14. List the features of object relational.
15. How does the concept of an object in the object oriented model differ from the

concept of an entity in the entity relationship model?

16. Can we have more than one constructor in a class? If yes, explain the need for

such a situation.

17. List Explain the need of object oriented database.

18. Give the general syntax of XML file.

PART-B

1. Explain the structure of XML databases. (13)

2. Describe the important models of information retrieval. (13)

3.
Describe about Distributed Databases and their characteristics,

advantages and disadvantages.(13)
4.Explain the necessary characteristics a system must satisfy to be considered as an

object oriented database management system. (13)

5.Differentiate between Document Type Definition and XML schema. (13)

6.i) Discuss about Distributed Transactions (07)

ii) Show the challenges in object relational database.(6)

7.i) Compare and contrast between object oriented and XML databases. (7)
ii)Give XML representation of bank management system and also explain about DTD and XML schema (6)
8.Discuss briefly about object database concepts. (13)

9.Illustrate the concepts for information retrieval. (13)

10.Illustrate the hierarchical data model in XML. (13)
11.Point out the types of queries in IR systems. (13)
12.Describe in detail about Object Model of ODMG. (13)
13.i) Explain the features of object relational. (7)
ii) Examine the process of Querying XML data (6)

14. Give the DTD or XML Schema for an XML representation of the following nested-relational schema : (15)

Emp=(ename,ChildrenSet setoff(Children),SkillsSet setoff (Skills))

Children=(name,Birthday)

Birthday=(day,month,year)

Skills=(type,ExamsSet setoff(Exams))

Exams=(year,city)

Explain with diagrammatic illustration the architecture of a distributed database management system. (15)

PRATHYUSHA ENGINEERING COLLEGE

Poonamallee – Tiruvallur Road, Chennai – 602025.

CS8602

Compiler Design

(Anna University - Regulation)

Ms.K.P.Revathi

UNIT 1 – INTRODUCTION TO COMPILERS

Topics to be Covered

Translators-Compilation and Interpretation-Language processors -The Phases of Compiler-Errors Encountered in Different Phases-The Grouping of Phases-Compiler Construction Tools - Programming Language basics.

1.1 Translators:

A **translator** is a computer program that performs the translation of a program written in a given programming language into a functionally equivalent program in a different computer language, without losing the functional or logical structure of the original code (the "essence" of each program).

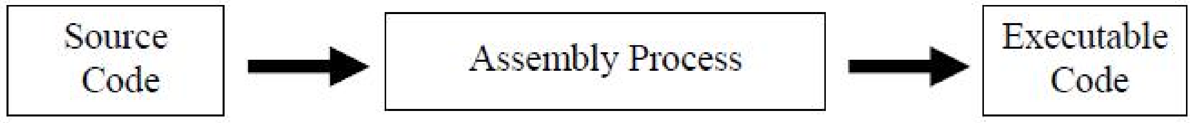
Types of Computer Language Translators:

The widely used translators that translate the code of a computer program into a machine code are:

1. *Assemblers*
2. *Interpreters*
3. *Compilers*

Assembler:

An Assembler converts an assembly program into machine code.



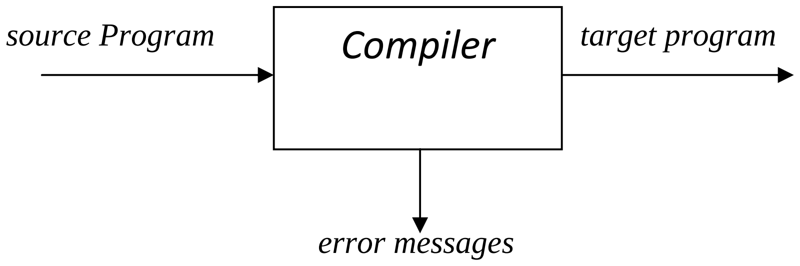
1.2 Compilation and Interpretation:

1.2.1 Compilation:

Compilation is the conceptual process of translating source code into a CPU-executable binary target code.

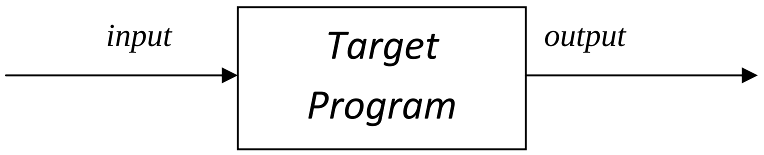
Compiler:

A compiler is a program that reads a program written in one language – the *source* language – and translates it into an equivalent program in another language – the *target language.*



As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.



Advantages of Compiler:

1. Fast in execution
2. The object/executable code produced by a compiler can be distributed or executed without having to have the compiler present.
3. The object program can be used whenever required without the need to of recompilation.

Disadvantages of Compiler:

1. Debugging a program is much harder. Therefore not so good at finding errors.
2. When an error is found, the whole program has to be re-compiled.

History of Compiler:

- Until 1952 most of the programs were written in assembly language
- In 1952 Grace Hopper writes the first compiler for the A-0 programming language
- Between 1957 – 58 John Backus writes the first Fortran compiler. Optimization of the code was the integral component of the compiler.

Applications of Compiler Technology:

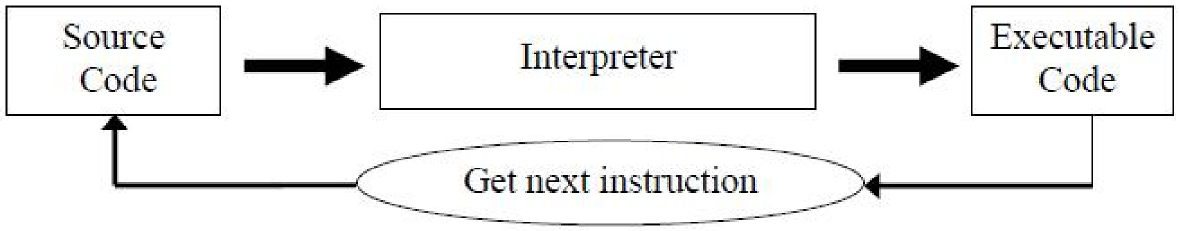
- Implementation of High Level Programming Languages
- Optimizations for Computer Architectures (both *parallelism and memory hierarchies* improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance of an application)
- Design of a new computer architecture
- Program Translations (Program Translation techniques are: Binary Translation, Hardware Synthesis, Database Query Interpreters, Compiled Simulation)
- Software Productivity Tools (Ex. Structure editors, type checking, bound checking, memory management tools, etc)

1.2.2 Interpretation:

Interpretation is the conceptual process of translating a high level source code into executable code.

Interpreter:

An Interpreter is also a program that translates high-level source code into executable code. However the difference between a compiler and an interpreter is that **an interpreter translates one line at a time and then executes it**: no object code is produced, and so the program has to be interpreted each time it is to be run. If the program performs a section code 1000 times, then the section is translated into machine code 1000 times since each line is interpreted and then executed.



Advantages of an Interpreter:

1. Good at locating errors in programs
2. Debugging is easier since the interpreter stops when it encounters an error.
3. If an error is deducted there is no need to retranslate the whole program

Disadvantages of an Interpreter:

1. Rather slow
2. No object code is produced, so a translation has to be done every time the program is running.
3. For the program to run, the Interpreter must be present

Difference between Compiler and Interpreter:

S.No.	***Compiler***	***Interpreter***
1.	Compiler works on the complete program at once. It takes the entire program as input.	Interpreter Program works line by line. It takes one statement at a time as input.
2.	Compiler generates intermediate code, called the object code or machine code.	Interpreter does not generate intermediate object code or machine code.
3.	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs faster than interpreter.	Interpreter executes conditional control statements at a much slower speed.
4.	Compiled program take more memory because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, interpreted programs are more memory efficient.

S.No.	***Compiler***	***Interpreter***
5.	Compile once and run any time. Compiled program does not need to be compiled every time.	Interpreted programs are interpreted line by line every time they are executed.
6.	Errors are reported after the entire program is checked for syntactical and other errors.	Error is reported as soon as the first error is encountered. Rest of the program will be checked until the existing error is removed.
7.	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and report errors as it encounters them.
8.	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from the first line and stops execution only if it encounters an error.
9.	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug.
10.	*Examples:* *C, C++, COBOL*	*Examples:* *BASIC, VISUAL BASIC, Python, Ruby, PERL, MATLAB, Lisp*

Hybrid Compiler:

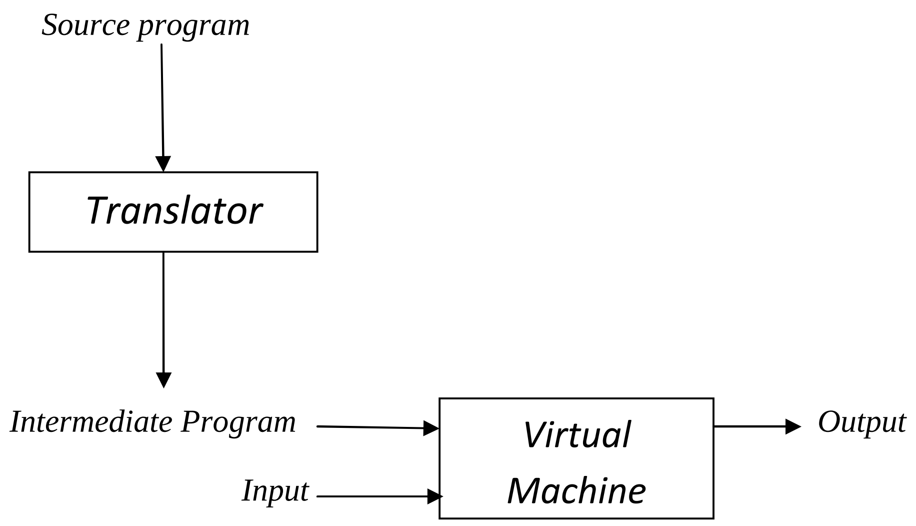
Hybrid compiler is a compiler which translates a human readable source code to an intermediate byte code for later interpretation. So these languages do have both features of a compiler and an interpreter. These types of compilers are commonly known as Just In-time Compilers (JIT).

Example of a Hybrid Compiler:

Java is one good example for these types of compilers. Java language processors combine compilation and interpretation. A Java Source program may be first compiled into an intermediate form called *byte codes.* The byte codes are then interpreted by a virtual machine.

A benefit of this arrangement is that the byte codes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers called *just-in-time* compilers, translate the byte codes into machine language immediately before they run the intermediate program to process the input.



Compilers are not only used to translate a source language into the assembly or machine language but also used in other places.

Example:

1. ***Text Formatters:*** A text formatter takes input that is stream of characters, most of which is text, some of which includes commands to indicate paragraphs, figures, or mathematical structures like subscripts and superscripts.
2. ***Silicon compilers:*** A silicon compiler has a source language that is similar or identical to a conventional programming language. The variable of the language represent logical signals (0 or 1) or groups of signals in a switching circuit. The output is a circuit design in an appropriate language.
3. ***Query Interpreters:*** A query interpreter translates a predicate containing relational and Boolean operators into commands to search a database for records satisfying that predicate.

1.3 Language Processors:

A language processor is a program that processes the programs written in programming language (source language). A part of a language processor is a language translator, which translates the program from the source language into machine code, assembly language or other language.

An integrated software developmental environment includes many different kinds of language processors. They are:

1. Pre Processor
2. Compiler
3. Assembler
4. Linker
5. Loader

1. Pre Processor

The Pre Processor is the system software which is used to process the source program before fed into the compiler. They may perform the following functions:

1. *Macro Processing:* A preprocessor may allow a user to define macros that are shorthand for longer constructs.
2. *File Inclusion:* A preprocessor may include header files into the program text. For example, the C pre-processor causes the contents of the file <global.h> to replace the statement #include <global.h> when it processes a file containing this statement.
3. *Rational Preprocessors:* These processors provides the user with built-in macros for constructs like while-statements or if-statements etc.,
4. *Language Extensions:* It provides features similar to built-in macros. For example, the language Equel is a database query language embedded in C.

2. Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

3. Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

4. Linker

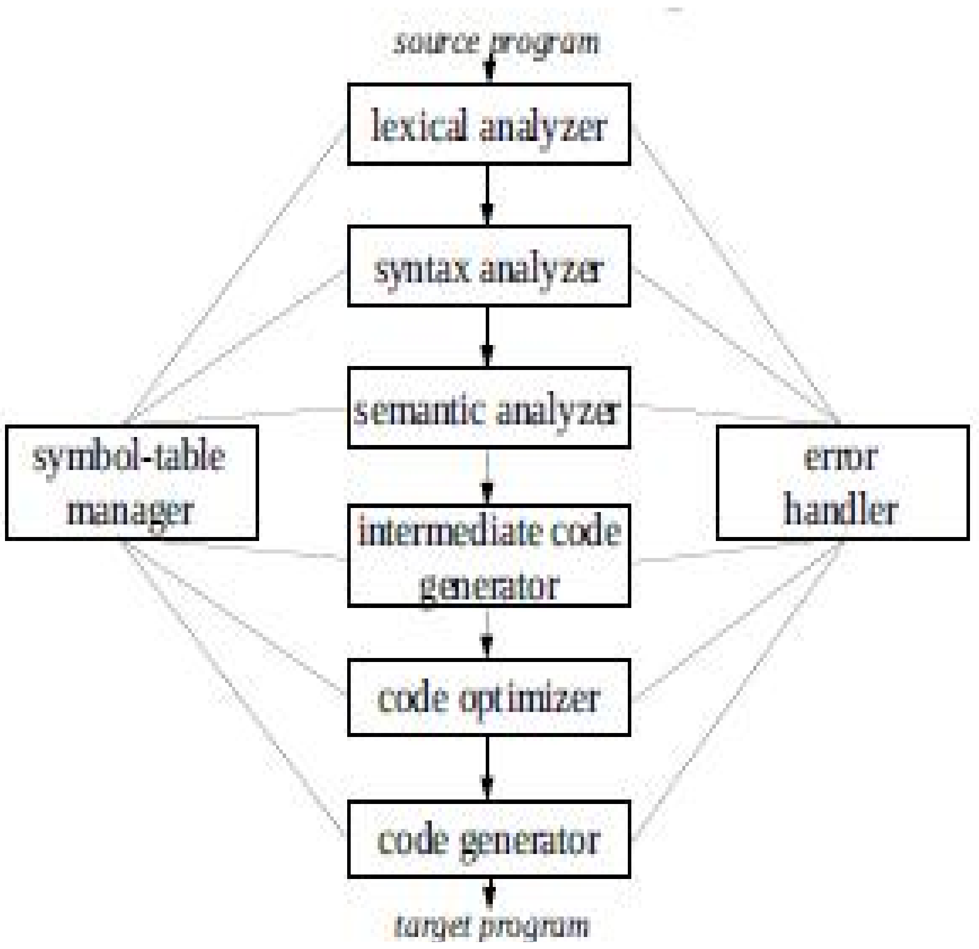
Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

5. Loader

Loader is a part of operating system and is responsible for loading executable files into memory and executes them. It calculates the size of a program *instructions and data* and creates memory space for it. It initializes various registers to initiate execution.

1.4 Phases of Compiler:

A compiler operates in phases, each of which transforms the source program from one representation to another.



The Analysis – Synthesis Model of Compilation:

There are two parts to compilation:

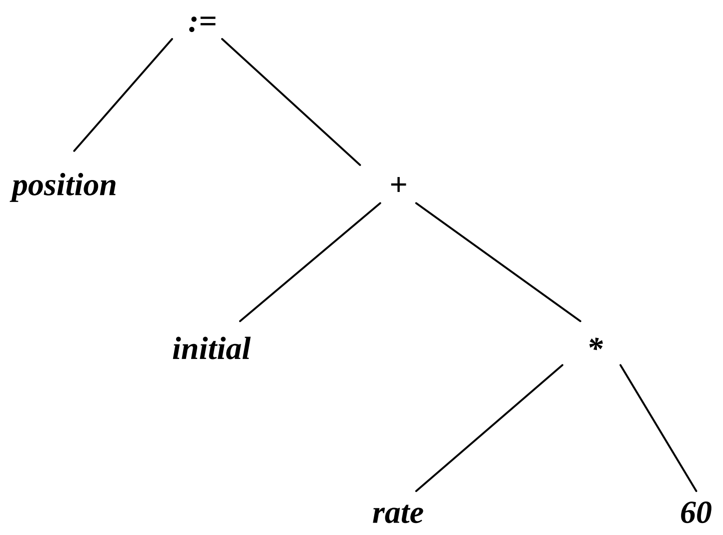
- Analysis and
- Synthesis

1. Analysis:

The first three phases forms the bulk of the analysis portion of a compiler. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a syntax tree, in which each node represents an operation and the children of a node represent the arguments of the operation.

Example:

*Syntax tree for position := initial + rate * 60*



2. Synthesis Part:

The synthesis part constructs the desired target program from the intermediate representation. This part requires most specialized techniques.

<u>The Analysis Phase:</u>

Lexical Analysis: The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically sequence of characters, such as identifier, a keyword (if, while, etc), a punctuation character, or a multi-character operator work like :=. The character sequence forming a token is called the *lexeme* for the token.

Certain tokens will be augmented by a "lexical value". Ex. When an identifier ***rate*** is found, the lexical analyzer generates the token ***id*** and also enters ***rate*** into the symbol table, if it is not already exist. The lexical value associated with this ***id*** then points to the symbol-table entry for ***rate.***

*Example: position := initial + rate * 60*

Tokens:

1. *position, initial and rate* - ***id***

2. *:=, + and * are* ***signs***

3. *60 is a* ***number***

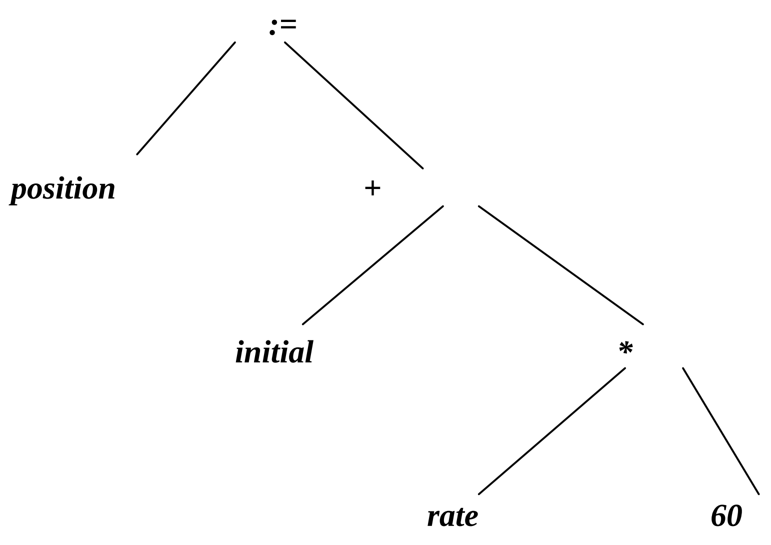
Thus the lexical analyzer will give the output as:

Id_1 := id_2 + id_3 * 60

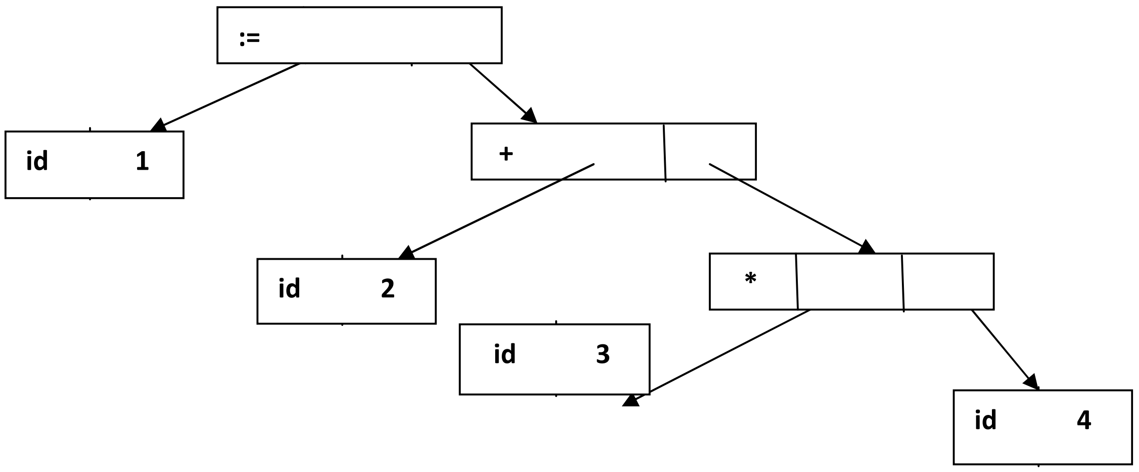
Syntax Analysis:

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree *or syntax tree*. In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.
It imposes a hierarchical structure of the token stream in the form of parse tree or syntax tree. The syntax tree can be represented by using suitable data structure.

*Example: position := initial + rate * 60*



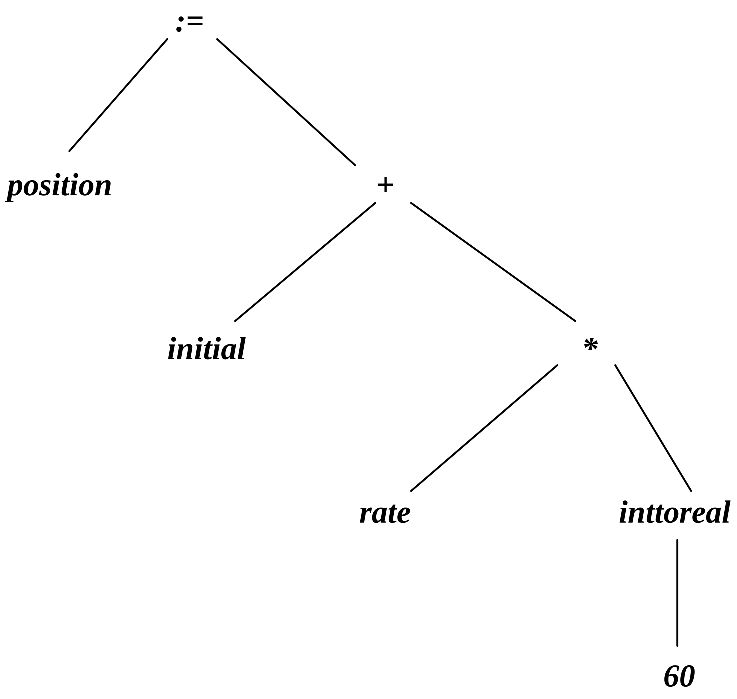
Data structure of the above tree:



Semantic Analysis:

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

This analysis inserts a conversion from integer to real in the above syntax tree.



<u>Synthesis Phase:</u>

Intermediate Code Generation:

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.
Intermediate code have two properties: easy to produce and easy to translate into the target program. An intermediate code representation can have many forms. One of the form is *three-address code,* which is like the assembly language for a machine in which every memory location can act like a register and *three-address code* have at most three operands.

Example: The output of the semantic analysis can be represented in the following intermediate form:

temp1 := inttoreal (60)

*temp2 := id3 * temp1*

temp3 := id2 + temp2

id1 := temp3

Code Optimization:

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources *CPU, memory*. In the following example the natural algorithm is used for optimizing the code.

Example:

The output of intermediate code can be optimized as:

*temp1 := id3 * 60.0*

id1 := id2 + temp1

The compiler that do most code optimization are called *“optimizing compilers”*.

Code Generation:

This is the final phase of the compiler which generates the target code, consisting normally of relocatable machine code or assembly code. Variables are assigned to the registers.

Example:

The output of above optimized code can be generated as:

MOVF id3, R2

MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id3

The first and the second operands of each instruction specify a source and destination respectively. The F in each instruction denotes the floating point numbers. The # signifies that 60.0 is to be treated as constant.

Activities of Compiler:

Symbol table manager and error handler are the other two activities in the compiler which is also referred as phases. These two activities interact with all the six phases of a compiler.

Symbol Table Manager:

The symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

The attributes of the identifiers may provide the information about the storage allocated for an identifier, its type, its scope (where in the program it is valid), and in the case of procedure names the attributes provide information about the number and types of its arguments, the method of passing each argument (eg. by reference), and the type returned, if any.

The symbol table allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. Attributes of the identifiers cannot be determined during lexical analysis phase. But it can be determined during the syntax and semantic analysis phases. The other phase like code generators uses the symbol table to retrieve the details about the identifiers.

Error Handler: (Error Detection and Reporting)

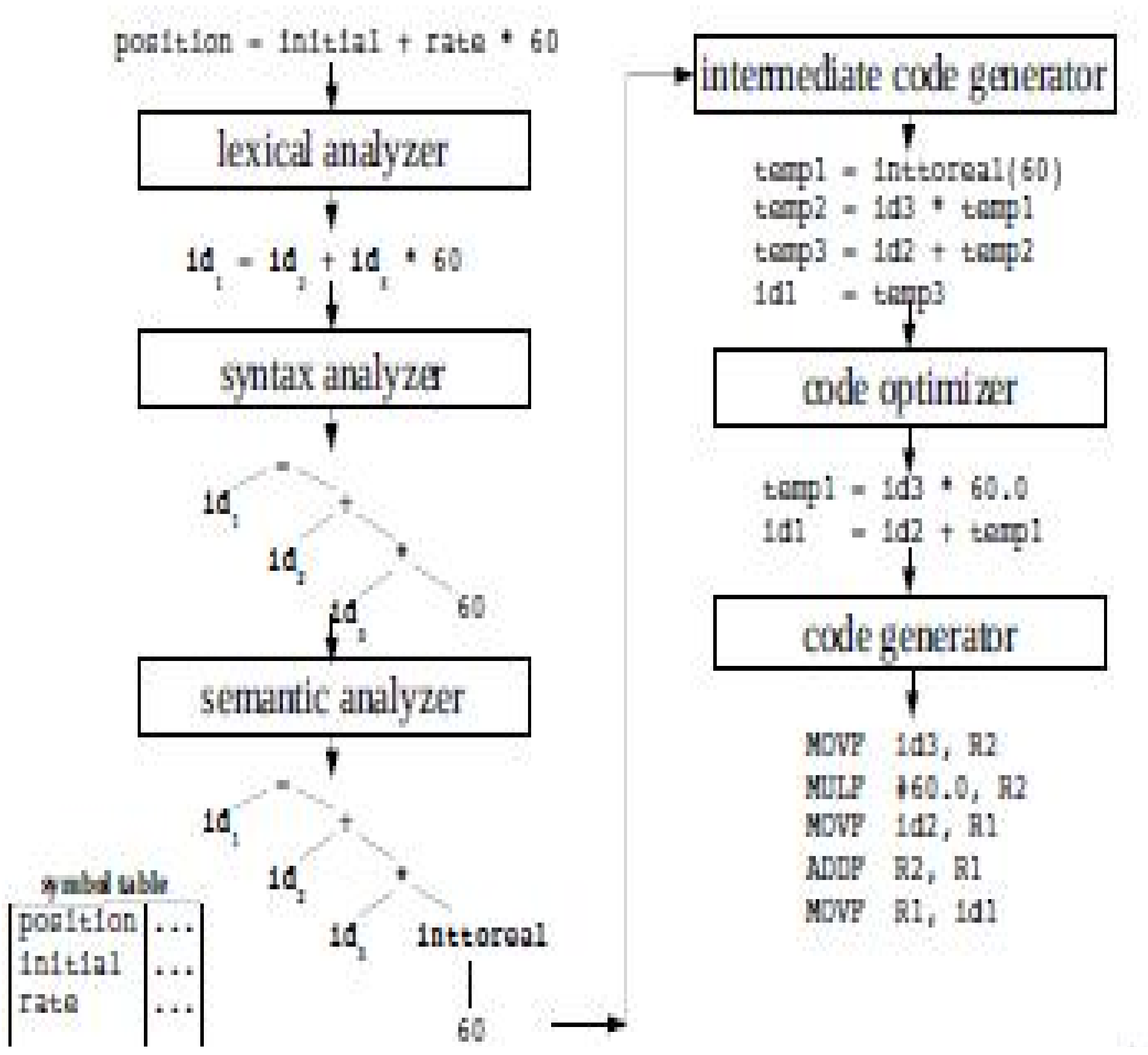
Each phase can encounter errors. After the deduction of an error, a phase must somehow deal with that error, so that the compilation can proceed, allowing further errors in the source program to be detected.

Lexical Analysis Phase: If the characters remaining in the input do not form any token of the language, then the lexical analysis phase detect the error.

Syntax Analysis Phase: The large fraction of errors is handled by syntax and semantic analysis phases. If the token stream violates the structure rules (syntax) of the language, then this phase detects the error.

Semantic Analysis Phase: If the constructs have right syntactic structure but no meaning to the operation involved, then this phase detects the error. Ex. Adding two identifiers, one of which is the name of the array, and the other the name of a procedure.

Translation of statement



1.5 Errors Encountered in Different Phases:

Program submitted to a compiler often have errors of various kinds. So, good compiler should be able to detect as many errors as possible in various ways and also recover from them.

Each phase can encounter errors. After the deduction of an error, a phase must somehow deal with that error, so that the compilation can proceed, allowing further errors in the source program to be detected.

Errors during Lexical Analysis:

If the characters remaining in the input do not form any token of the language, then the lexical analysis phase detect the error.

There are relatively few errors which can be detected during lexical analysis.

i. Strange characters

Some programming languages do not use all possible characters, so any strange ones which appear can be reported. However almost any character is allowed within a quoted string.

ii. Long quoted strings (1)

Many programming languages do not allow quoted strings to extend over more than one line; in such cases a missing quote can be detected.

iii. Long quoted strings (2)

If quoted strings can extend over multiple lines then a missing quote can cause quite a lot of text to be 'swallowed up' before an error is detected.

iv. Invalid numbers

A number such as 123.45.67 could be detected as invalid during lexical analysis (provided the language does not allow a full stop to appear immediately after a number). Some compiler writers prefer to treat this as two consecutive numbers 123.45 and .67 as far as lexical analysis is concerned and leave it to the syntax analyser to report an error. Some languages do not allow a number to start with a full stop/decimal point, in which case the lexical analyzer can easily detect this situation.

Error Recovery Actions:

The possible error-recovery actions are:

i) Deleting an extraneous character
ii) Inserting a missing character
iii) Replacing an incorrect character by correct character
iv) Transposing two adjacent characters

For example:

fi (a == 1) ....

Here ***fi*** is a valid identifier. But the open parentheses followed by the identifier may tell ***fi*** is misspelling of the keyword if or an undeclared function identifier.

Errors in Syntax Analysis:

The large fraction of errors is handled by syntax and semantic analysis phases. If the token stream violates the structure rules (syntax) of the language, then this phase detects the error.

The errors detected in this phase include misplaced semicolons or extra or missing braces; that is, "{" or " } . " As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error. (However, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code). Unbalanced parenthesis in expressions is handled

During syntax analysis, the compiler is usually trying to decide what to do next on the basis of expecting one of a small number of tokens. Hence in most cases it is possible to automatically generate a useful error message just by listing the tokens which would be acceptable at that point.

```
Source:  A + * B
Error:       | Found '*', expect one of: Identifier, Constant, '('
```

More specific hand-tailored error messages may be needed in cases of bracket mismatch.

```
Source:  C := ( A + B * 3 ;
Error:                  | Missing ')' or earlier surplus '('
```

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc.. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Errors during Semantic Analysis

Semantic errors are mistakes concerning the meaning of a program construct; they may be either type errors, logical errors or run-time errors:

(i) **Type errors** occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.

(ii) **Logical errors** occur when a badly conceived program is executed, for example: while x = y do ... when x and y initially have the same value and the body of loop need not change the value of either x or y.

(iii)**Run-time errors** are errors that can be detected only when the program is executed, for example:

```
var x : real; readln(x); writeln(1/x)
```

which would produce a run time error if the user input 0.

Syntax errors must be detected by a compiler and at least reported to the user (in a helpful way). If possible, the compiler should make the appropriate correction(s). Semantic errors are much harder and sometimes impossible for a computer to detect.

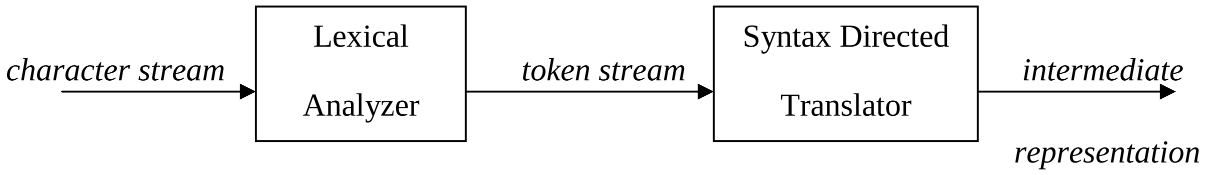
1.6 The Grouping of Phases:

Depending on the relationship between phases, the phases are grouped together as front end and a back end.

Front End:

The front end consists of phases that depend primarily on the source language and are largely independent of the target machine. The phases of front end are:

- Lexical Analysis
- Syntactic Analysis
- Creation of the symbol table
- Semantic Analysis
- Generation of the intermediate code
- A part of code optimization
- Error Handling that goes along with the above said phases



Back End:

The back end includes the phases of the compiler that depend on the target machine, and these phases do not depend on the source language, but depend on the intermediate language. The phases of back end are:

- Code Optimization
- Code Generation
- Necessary Symbol table and error handling operations

Categories of Compiler Design:

Based on the grouping of phases there are two types of compiler design is possible:

1. A Single Compiler for different Machine - It is possible to produce a single compiler for the same source language on a different machine by taking the front end of a compiler as common and redo its associated back end.
2. Several Compiler for One Machine – It is possible to produce several compilers for one machine by using a common back end for the different front ends.

1.7 Compiler Construction Tools:

In order to atomize the development of compilers some general tools have been created. These tools use specialized languages for specifying and implementing the component. The most successful tool should hide the details of the generation algorithm and produce components which can be easily integrated into the remainder of the compiler. These tools are often referred as *compiler – compilers, compiler – generators, or translator-writing systems.*

Some of the compiler-construction tools are:

Parser generators: Automatically produce syntax analyzers from a grammatical description of a programming language.

Scanner generators: Produce lexical analyzers from a regular-expression description of the tokens of a language.

Syntax-directed translation engines: Produce collections of routines for walking a parse tree and generating intermediate code.

Code-generator generators: Produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
Data-flow analysis engines: Facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

Compiler-construction toolkits: Provide an integrated set of routines for constructing various phases of a compiler.

1.8 Programming Language Basics:

The important terminology and distinctions that appear in the programming languages are:

1. The Static / Dynamic Distinction:

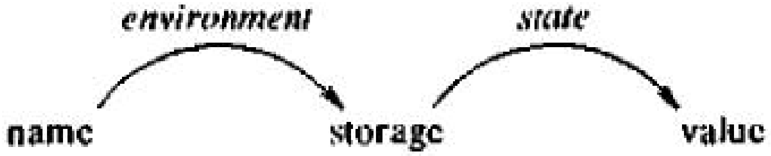
- A programming language can have *static policy* and *dynamic policy*.
- *Static Policy:* The issues that can be decided at compile time by compiler is called *static policy.*
- *Dynamic Policy:* The issues that can be decided at run time of the program is called *dynamic policy.*
- One of the issue decision policy in the language is the scope of declarations.
- *Scope Rules:* The *scope* of a declaration of *x* is the context in which uses of *x* refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program and can be determined by compiler. Otherwise, the language uses *dynamic scope.*
- *Example in Java:*

  public static int x;

  The compiler can determine the location of integer *x* in memory.

2. Environments and States:

The association of names with locations in memory (the *store)* and then with values can be described by two state mappings that change as the program runs.



Two-State Mapping from Names to Values

The *environment* is a mapping from names to locations in the store.

The *state* is a mapping from locations in store to their values. That is, the state maps *l*-values to their corresponding *r*-values, in the terminology of C.

Example:

The storage address 100, associated with variable *pi,* holds *0.* After the assignment *pi := 3.14*, the same storage is associated with *pi,* but the value held there is *3.14.*

3. Static Scope and Block Structure:

Scope Rules: The *scope* of a declaration of *x* is the context in which uses of *x* refer to this declaration. . A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program and can be determined by compiler. Otherwise, the language uses *dynamic scope.*

- *Example in Java:*

  public static int x;

The compiler can determine the location of integer *x* in memory.

The static-scope policy is as follows:

1. A C program consists of a sequence of top-level declarations of variables and functions.
2. Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.
3. The scope of a top-level declaration of a name *x* consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of *x.*

Block Structures:

Languages that allow blocks to be nested are said to have *block structure.* A name a: in a nested block *B* is in the scope of a declaration *D* of *x* in an enclosing block if there is no other declaration of *x* in an intervening block.

4. Explicit Access Control:

- Classes and structures introduce a new scope for their members.
- The use of keywords like **public, private,** and **protected,** object oriented languages such as C + + or Java provide explicit control over access to member names in a super class.
- These keywords support *encapsulation* by restricting access.
- Thus,
    - Private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C + + term).
    - Protected names are accessible to subclasses.
    - Public names are accessible from outside the class.

5. Dynamic Scope:

- *Scope Rules:* The *scope* of a declaration of *x* is the context in which uses of *x* refer to this declaration.
- A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program and can be determined by compiler.
- *Example in Java:*

  public static int x;

  The compiler can determine the location of integer *x* in memory.
- The language uses *dynamic scope* if it is not possible to determine the scope of a declaration during compile time.
- *Example in Java:*

  public int x;
- With dynamic scope, as the program runs, the same use of *x* could refer to any of several different declarations of *x*.

6. Parameter Passing Mechanism: Parameters are passed from a calling procedure to the callee either by value (call by value) or by reference (call by reference). Depending on the procedure call, the actual parameters associated with formal parameters will differ.

Call-By-Value: In *call-by-value,* the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure.

Call-By-Reference:

In *call-by-reference,* the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

Call-By-Name:

A third mechanism — call-by-name — was used in the early programming language Algol **60.** It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct).

When large objects are passed by value, the values passed are really references to the objects themselves, resulting in an effective call-by-reference.

7. Aliasing: When parameters are (effectively) passed by reference, two formal parameters can refer to the same object, called aliasing. This possibility allows a change in one variable to change another.

UNIT 2 – LEXICAL ANALYSIS

Topics to be Covered

Need and Role of Lexical Analyzer-Lexical Errors-Expressing Tokens by Regular Expressions-Converting Regular Expression to DFA- Minimization of DFA-Language for Specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.

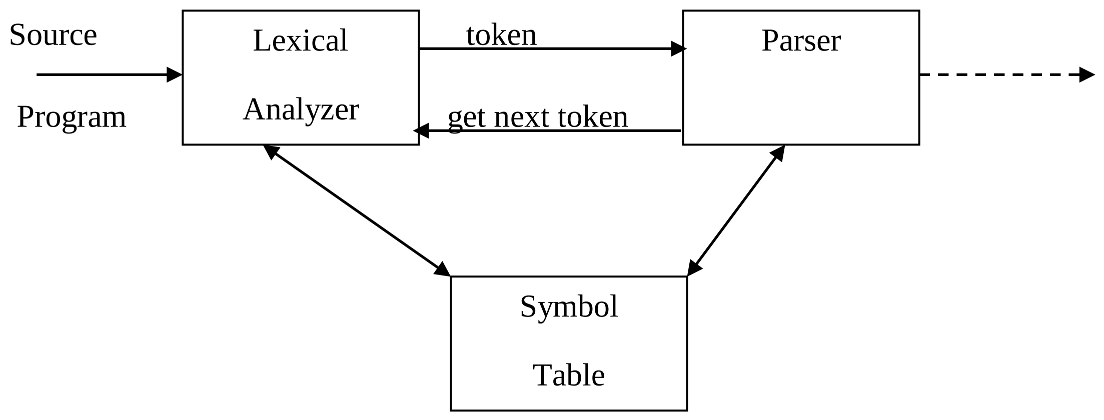
Lexical Analysis

The Role of the Lexical Analyzer

The lexical analyzer is the first phase of a compiler.

Main Task of Lexical Analyzer:

Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



The above diagram illustrates that the lexical analyzer is a subroutine or a co routine of the parser. Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Secondary Tasks of Lexical Analyzer:

Since Lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface.

1. Stripping out from the source program comments and white space in the form of blank, tab and newline characters.
2. Correlating error messages from the compiler with the source program. Example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message.

Phases of Lexical Analyzer:

Lexical analyzers are divided into a cascade of two phases:

Scanning – the scanner is responsible for doing simple tasks (Example – Fortran compiler use a scanner to eliminate blanks from the input)

Lexical analysis – the lexical analyzer does the more complex operations.

Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:

1. To make the design simpler. The separation of lexical analysis from syntax analysis allows the other phases to be simpler. For example, parsing a document with comments and white spaces is more complex than it is removed in the previous phase itself.
2. To improve the efficiency of the compiler. A separate lexical analyzer allows to construct an efficient processor. A large amount of time is spent in reading the source program and partitioning it into tokens. Specialized buffering techniques speed up the performance.
3. To enhance the compiler portability. Input alphabets and device specific anomalies can be restricted to the lexical analyzer.

Tokens, Patterns and Lexemes:

Token: A token is an atomic unit represents a logically cohesive sequence of characters such as an identifier, a keyword, an operator, constants, literal strings, punctuation symbols such as parentheses, commas and semicolons.

Eg.	rate	-	identifier
	+, -	-	operator
	if	-	keyword

Pattern: A pattern is a rule used to describe lexeme. It is a set of strings in the input for which the same token is produced as output.

Lexeme: A lexeme is a sequence of characters in the source program which is matched by the pattern for a token. i.e. lexemes represents tokens.

Token	Sample Lexemes	Informal Description of Pattern
Const	Const	const
If	If	if
Relation	<, <=, =, < >, >, >=	< or <= or = or < > or > or >=
Id	pi, count, A2	Letter followed by letters and digits
Num	3.1416, 0, 6.02E23	any numeric constant
Literal	"garbage collection"	any characters between " and " except "

Attributes for Tokens:

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.

For example, the pattern **relation** matches the operators like <, <=, >, >=, =, < >. It is necessary to identify operator which is matched with the pattern.

The lexical analyzer collects other information about tokens as its attributes. A token has only a single attribute, a pointer to the symbol-table entry in which the information about the token is kept.

For example: The tokens and associated attribute-values for the Fortran statement

X   = Y * Z ** 4

are written below as a sequence of pairs:

<**id,** pointer to symbol-table entry for X>

<**assign_op,**>

<**id,** pointer to symbol-table entry for Y>

<**mult_op,**>

<**id,** pointer to symbol-table entry for Z>

<**exp_op,**>

<**num,** integer value 4>

For certain attribute pairs, there is no need for an attribute value.

Eg. <**assign_op,**>

For others, the compiler stores the character string that forms a value in a symbol table.

Lexical Errors:

A lexical analyzer has a very localized view of a source programs.

The possible error-recovery actions are:

i) Deleting an extraneous character
ii) Inserting a missing character
iii) Replacing an incorrect character by correct character
iv) Transposing two adjacent characters

For example:

```
fi ( a == 1) ....
```

Here ***fi*** is a valid identifier. But the open parentheses followed by the identifier may tell ***fi*** is misspelling of the keyword if or an undeclared function identifier.

INPUT BUFFERING:

Input buffering is a method used to read the source program and to identify the tokens efficiently. There are three general approaches to the implementation of a lexical analyzer.

1. Use a lexical-analyzer generator to produce the lexical analyzer from a regular-expression based specification. In this case, the generator provides routines for reading and buffering the input. Example – Lex Compiler
2. Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and explicitly manage the reading of input.

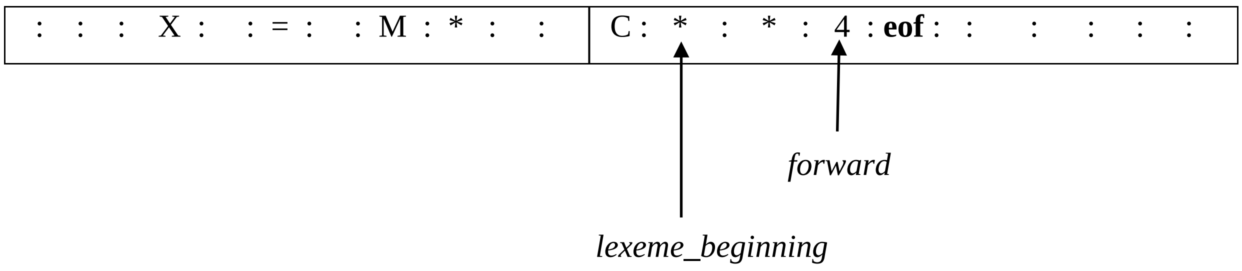
Since the lexical analyzer is the only phase of the compiler that reads the source program character-by-character, it is possible to spend a considerably amount of time in the lexical analysis phase. Thus the speed of lexical analysis is a concern in compiler design.

The following technique uses two-buffer input scheme to identify the tokens. The speed of the lexical .analyzer can be improved by using the sentinels to mark the buffer end.

Buffer Pairs:

The lexical analyzer needs to look-ahead many characters beyond the lexeme for finding the pattern. The lexical analyzer uses a function ungetc( ) to push the look-ahead characters back into the input stream. In order to reduce the amount of overhead required to process an input character, specialized buffering techniques have been developed.

A buffer is divided into N-character halves where N is the number of characters on one disk block. Example 1024 or 4096


: : : X : : = : : M : * : :
C : * : * : 4 : **eof** : : : : : :
forward
lexeme_beginning

Input Buffer with two halves

The processing of buffer pair is as follows:

1. Read N input character into each half of the buffer using one system read command instead of reading each input character
2. If fewer than N characters remain in the input, then eof marker is read into the buffer after the input characters.
3. Two pointers to the input buffer are maintained. Initially both pointers point to the first character of the next lexeme to be found.
    a. Begin pointer points the s tart of the lexeme
    b. The forward pointer is set to the character at its right end
4. Once the lexeme is identified, both pointers are set to the character immediately past the lexeme.

If the forward pointer is reaching the halfway mark, the right half is filled with N new input characters. If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer. The number of tests to be required is very large.

Code to advance forward pointer:

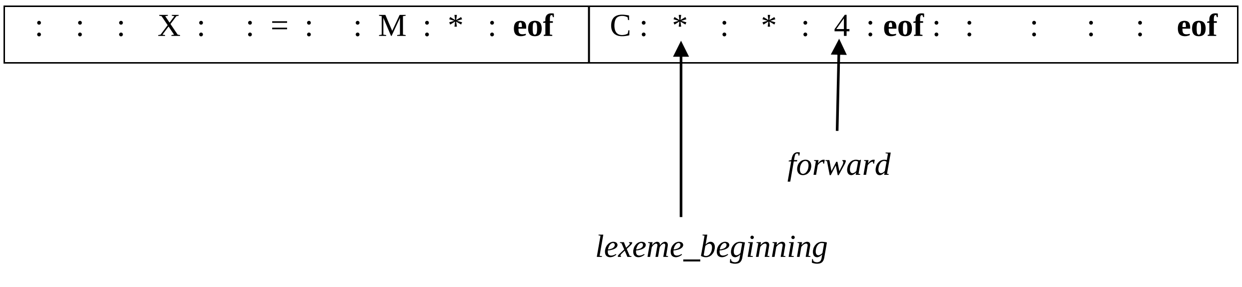
```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1
end

else if forward at end of second half then begin
       reload first half;
       move forward to beginning of first half
end
else
       forward := forward + 1;
```

Sentinels:

In the previous scheme mentioned a check should be made each time when the *forward* pointer is moved that we have not moved off one half of the buffer. i.e. only one **eof** marker at the end.

A sentinel is a special character which is not a part of the source program used to represent the end of file. (eof)

Instead of testing the forward pointer each time by two tests, extend each buffer half to hold a sentinel character at the end and reduce the number of tests to one.



: : : X : : = : : M : * : eof
C : * : * : 4 : eof : : : : : eof
forward
lexeme_beginning

Sentinels at end of each buffer half

For most of the cases, the code performs only one test to see whether forward point to an eof. If it reaches the end of a buffer or the end of the file, then we performs more tests for checking each half and to reload other half of the buffer.

Look ahead code with sentinels:

```
forward := forward + 1;
if forward $\uparrow$ eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1
    end
    else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
    end
    else /* eof within a buffer signifying end of input */
        terminate lexical analysis
end
```

SPECIFICATION OF TOKENS:

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for set of strings.

Strings and Languages:

Alphabet: An alphabet or character class denotes any finite set of symbols. For example, Letters, Characters, ASCII characters, EBCDIC characters

String: A string over some alphabet is a finite sequence of symbols drawn from that alphabet. For example, 1 0 1 0 1 1 is a string over $\{0, 1\}^*$, ϵ is a empty string over $\{0, 1\}^*$

Length of the String : The length of the string 1 0 1 is denoted as | 1 0 1 | = 3 i.e. the number of occurrences of symbol is S.

Language: A language denotes any set of strings over some fixed alphabet Σ.

Example Language L = $\{0^n 1^n \mid n > 0\}$

Some common terms associated with parts of a string are as follows:

Let *s* be the string where S = “regular”.

TERM	**DEFINITION**
prefix of *s*	A string obtained by removing zero or more trailing symbols of string *s*; eg. *ban* is a prefix of *banana*
suffix of *s*	A string formed by deleting zero or more of the leading symbols of *s;* eg. *nana* is a suffix of *banana*
substring of *s*	A string obtained by deleting a prefix and a suffix from *s; eg. nan* is a substring of *banana.*
proper prefix, suffix or substring of *s*	Any nonempty string *x* that is, respectively, a prefix, suffix, or substring of *s* such that $s \neq x$
subsequence of *s*	Any string formed by deleting zero or more not necessarily contiguous symbols from *s;* eg. *baaa* is a subsequence of *banana.*

Operations on Languages:

There are several important operations that ca be applied to languages. For lexical analysis the following operations are applied:

OPERATION	**DEFINITION**
union of *L* and *M* written *L U M*	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of *L* and *M* written *LM*	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of *L* written *L**	$L^* = \bigcup_{i=0}^{\infty} L^i$ *L** denotes “zero or more concatenations of” *L*
positive closure of *L* written	∞

L^+	$L^+ = U L^i$ $i=1$ L^+ denotes "one or more concatenations of" L

Example:

Let L = {A, B, . . . , Z, a, b, . . . , z} and

D = {0, 1, . . . , 9}

By applying operators defined above on these languages L and D we get the following new languages:

1. $L \; U \; D$ is the set of letters and digits

   i.e. $L \; U \; D$ = {A,B, . . . ,Z, a, b, . . . , z, 0, 1, . . . , 9}
2. LD is the set of strings consisting of a letter followed by a digit

   i.e. LD = {0A, 0B, . . . , 0Z, 0a, 0b, . . . , 0z, 1A, 1B, . . ,1Z, 1a, 1b, . . ,1z, ... }
3. L^4 is the set of all four-letter strings i.e. L^4 = { aBAC, MNop, ... }
4. L^* is the set of all strings of letters, including ε, the empty string

   *i.e. L** = { ε, A, B, . . . , Z, a, b, . . . , z, AB, BA, aB, ... }
5. *L(L U D)** is the set of all strings of letters and digits beginning with a letter
6. D^+ is the set of all strings of one or more digits

Regular Expressions:

A regular expression is built out of simple regular expressions using a set of defining rules. Each regular expression *r* denotes a language *L(r).*

Rules that define the regular expressions:

Basis:

i) ε is a regular expression denotes the language { ε }.

ii) If *a* is a symbol in Σ, then *a* is a regular expression denotes the language { *a* }

Induction:

iii) Suppose *r* and *s* are regular expressions denoting the language *L(r)* and *L(s).* Then,

   a. *(r) | (s)* is a regular expression denoting *L (r) U L (s).*

   b. *(r) (s)* is a regular expression denoting *L (r) L (s).*

c. *(r)** is a regular expression denoting *(L (r))*.*

d. *(r)* is a regular expression denoting *L (r).*

A language denoted by a regular expression is said to be a *regular set.*

The precedence and associativity of operators are as follows:

1. the unary operator * has the highest precedence and is left associative.
2. concatenation has the second highest precedence and is left associative.
3. | has the lowest precedence and is left associative.

Unnecessary parentheses can be avoided in the regular expression if the above precedence is adopted. For example the regular expression: *(a) | ((b)* (c))* is equivalent to *a | b*c.*

Example:

Let Σ = {a,b}

1. The regular expression *a | b* denotes the set { *a, b }*
2. The regular expression *(a | b) (a | b)* denotes *{aa, ab, ba, bb},* the set of all strings of *a's* and *b's* of length two. Another regular expression for this same set is *aa | ab | ba | bb.*
3. The regular expression *a** denotes the set of all strings of zero or more *a's* i.e. *{ ε, a, aa, aaa, . . . }*
4. The regular expression *(a | b)** denotes the set of all strings containing zero or more instances of an *a* or *b,* that is, the set of all strings of *a's* and *b's.* An equivalent regular expression for this set is *(a*b*)**
5. The regular expression *a | a*b* denotes the set containing the string *a* and all strings consisting of zero or more *a's* followed by a *b.*

If two regular expressions *r* and *s* denote the same language, then we say *r* and *s* are equivalent and write *r = s.* For example, *(a | b) = (b | a).*

There are number of algebraic laws obeyed by regular expressions and these laws can be used to manipulate regular expressions into equivalent forms.

Let *r, s* and *t* be the regular expression. The following are the algebraic laws for these regular expressions:

AXIOM	**DESCRIPTION**
$r \mid s = s \mid r$	| is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	| is associative
$(rs) t = r (st)$	Concatenation is associative
$r (s \mid t) = rs \mid rt$ $(s \mid t) r = sr \mid st$	Concatenation distributes over |
$\epsilon r = r$ $r\varepsilon = r$	ε is the identity element for concatenation
$r^* = (r \mid \varepsilon)^*$	relation between * and ε
$r^{**} = r^*$	* is idempotent

Regular Definitions:

The regular expressions can be given names and defining regular expressions using these names is called regular definition. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

d_1 -> r_1

d_2 -> r_2

. . . .

d_n -> r_n

where each d_i is a distinct name, and each r_i is a regular expression over the symbols in $\Sigma \cup \{ d_1, d_2, \ldots , d_{i-1} \}$, i.e., the basic symbols and the previously defined names.

Example:

1. *Regular Definition for identifiers:*
   letter → A | B | . . . | Z | a | b | . . . | z
   digit → 0 | 1 | . . . | 9
   id → **letter (letter | digit)***

2. *Regular Definition for num:*

digit → 0 | 1 | . . . | 9

digits → **digit digit***

optional_fraction → **. digits** | ϵ

optional_exponent → (E (+ | - | ϵ) **digits**) | ϵ

num → **digits optional_fraction optional_exponent**

Notational Shorthands:

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances(+):* The unary postfix operator + means "one or more instances of". Example – $(r)^+$ - Set of all strings of one or more occurrences of *r*.
2. *Zero or One Instance (?):* The unary postfix operator ? means " zero or one instance of". Example – *(r)?* – One or zero occurrence of *r*.

   The regular definition for num can be written by using unary + and unary ? operator as follows:

   digit → 0 | 1 | . . . | 9

   digits → $\textbf{digit}^+$

   optional_fraction → **(. digits) ?**

   optional_exponent → (E (+ | -)? **digits**) ?

   num → **digits optional_fraction optional_exponent**

3. *Character Classes:* The notation where *a*, *b* and *c* are alphabet symbols denotes the regular expression *a* | *b* | *c*. An abbreviated character class such as [a – z] denotes the regular expression *a* | *b* | . . . | z.

   Using character classes the identifiers can be described as strings generated by regular expression: [A – Za – z] [A – Z a – z 0 – 9]*

Recognition of Tokens:

The tokens are recognized by following the grammatical specification of tokens.

Example:

Consider the following grammar fragment:

stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| ϵ

expr → *term* **relop** *term*

| *term*

term → **id**

| **num**

where the terminals **if, then, else, relop, id** and **num** generate sets of strings given by the following regular definitions:

if → if

then → then

else → else

relop → < | <= | = | < > | > | >=

id → **letter (letter | digit)***

num → **digit**$^+$ (**. digit** $^+$)? (E (**+ | -**)? **digit**$^+$) ?

letter → A | B | . . . | Z | a | b | . . . | z

digit → 0 | 1 | . . . | 9

Regular definition for White Space (ws) is:

delim → blank | tab | newline

ws → delim^{+}

The goal of the lexical analyzer is to isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute value using the table given below:

Regular Expression	**Token**	**Attribute - Value**
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
< >	relop	NE
>	relop	GT
>=	relop	GE

Regular Expression Patterns for Tokens

Transition Diagrams:

As an intermediate step in the construction of a lexical analyzer, a stylized flowchart called a *transition diagram.* Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about characters that are seen as the forward pointer scans the input.

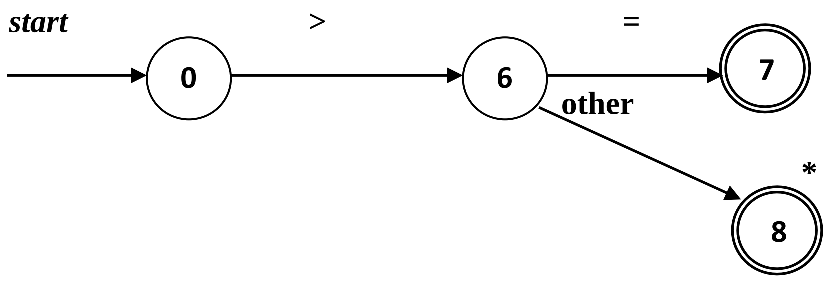
Positions in a transition diagram are drawn as circles and are called *states.* The states are connected by arrows, called *edges.* Edges leaving state *s* have labels indicating the input characters that can next appear after the transition diagram has reached state *s.* The label **other** refers to any character that is not indicated by any of the other edges leaving *s.*

One state is labeled as *start* state; it is the initial state of the transition diagram where control resides when we begin to recognize token. Certain states may have actions that are executed when the flow of control reaches that state. On entering a state we read the next input character. If there is an edge from the current state whose label matches this input character, then we go to the state pointed by the edge. Otherwise, we indicate failure.

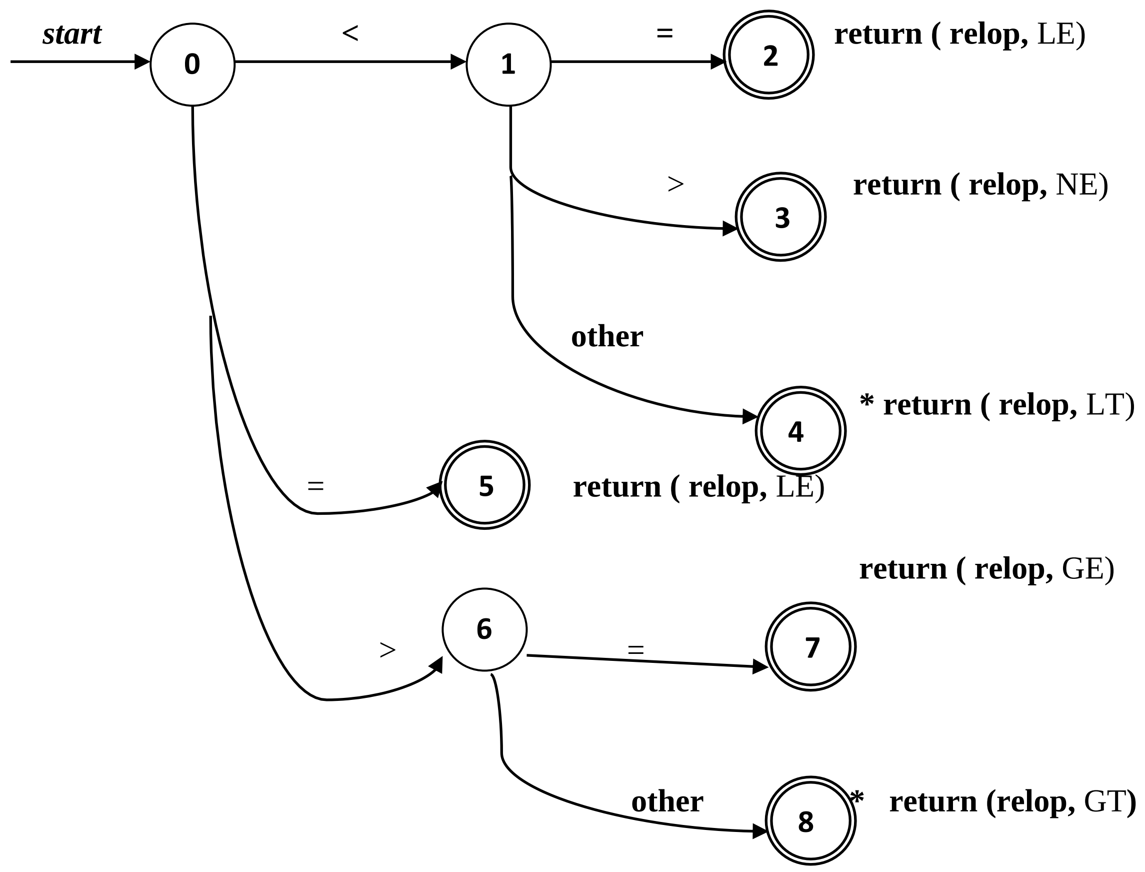
The symbol * is used to indicate states on which the input retraction must take place.

There may be several transition diagram, each specifying a group of tokens. If failure occurs in one transition diagram, then the forward pointer is retracted to where it was in the start state of this diagram, and activate the next transition diagram. Since the lexeme beginning and forward pointers marked the same position in the start state of the diagram, the forward pointer is retracted to the position marked by the lexeme_begining pointer. If failure occurs in all transition diagrams, then a lexical error has been detected and an error-recovery routine is invoked.

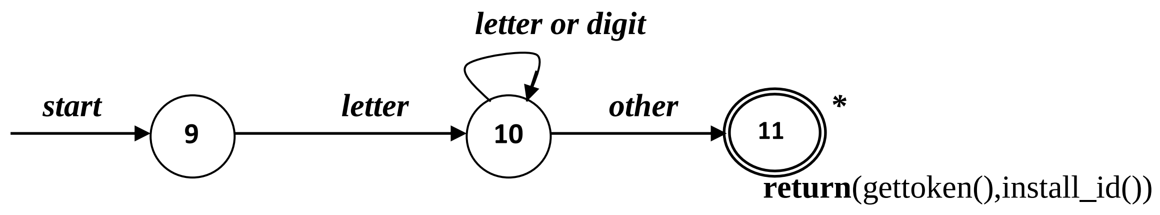
Transition Diagram for >=:



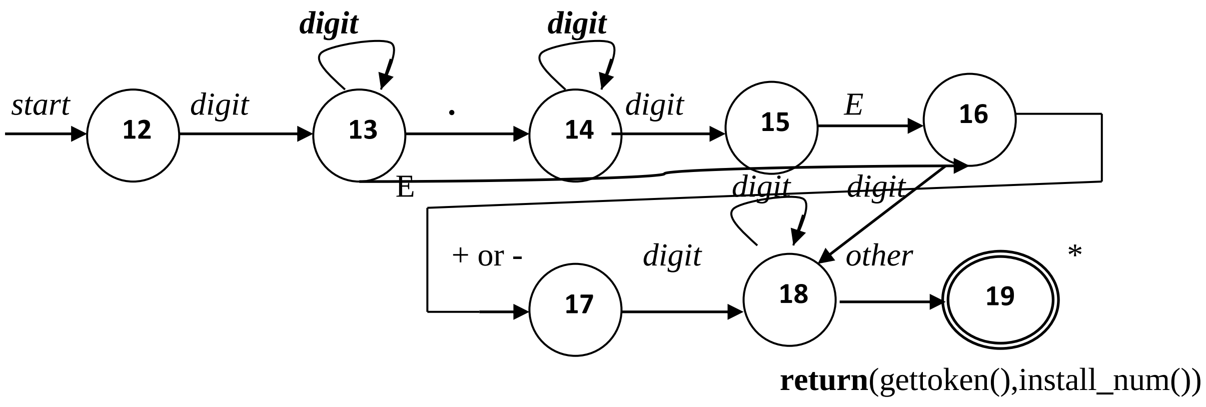
Transition Diagram for Relational Operators:

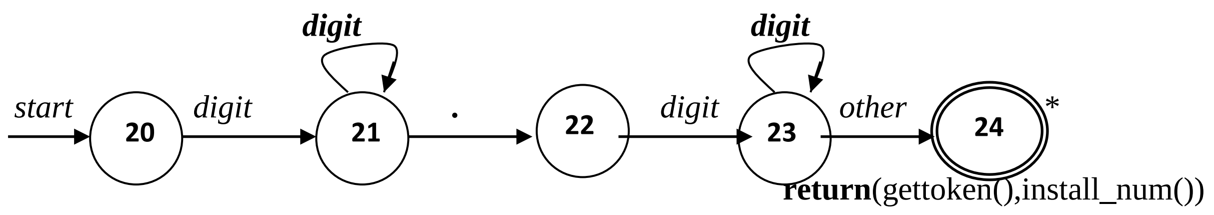


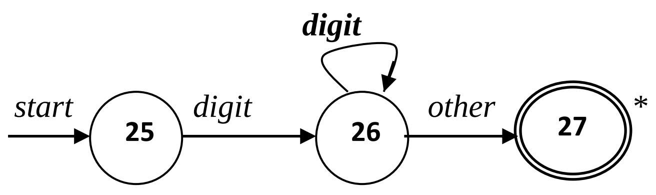
Transition Diagram for identifiers and keywords:



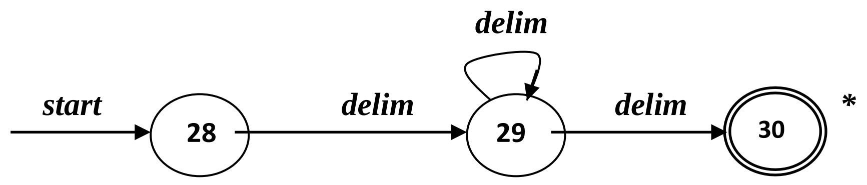
Transition Diagram for Unsigned Numbers in Pascal:







Transition Diagram for white space:



Convert Regular Expression to DFA -

Regular expression is used to represent the language (lexeme) of finite automata (lexical analyzer).

Finite automata

A recognizer for a language is a program that takes as input a string *x* and answers *yes* if *x* is a sentence of the language and *no* otherwise.

A regular expression is compiled into a recognizer by constructing a generalized transition diagram called a Finite Automaton (FA).

Finite automata can be Non-deterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA).

It is given by M = (Q, Σ, *qo,* F, δ*).*

Where Q - Set of states

Σ - Set of input symbols

qo - Start state

F - set of final states

δ - Transition function (mapping states to input symbol).

$\delta : Q \times \Sigma \rightarrow Q$

• Non-deterministic Finite Automata (NFA)

  o More than one transition occurs for any input symbol from a state.

  o Transition can occur even on empty string (Ɛ).

• Deterministic Finite Automata (DFA)

  o For each state and for each input symbol, exactly one transition occurs from that state.

Regular expression can be converted into DFA by the following methods:

(i) Thompson's subset construction

  • Given regular expression is converted into NFA

  • Resultant NFA is converted into DFA

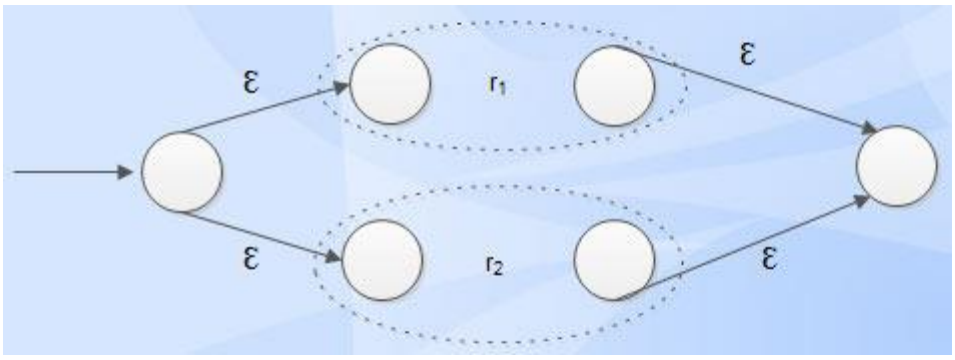
(ii) Direct Method

  • In direct method, given regular expression is converted directly into DFA.

Rules for Conversion of Regular Expression to NFA

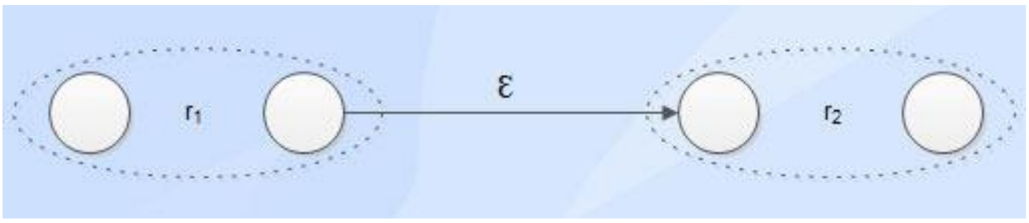
• Union

$$\mathbf{r = r_1 + r_2}$$



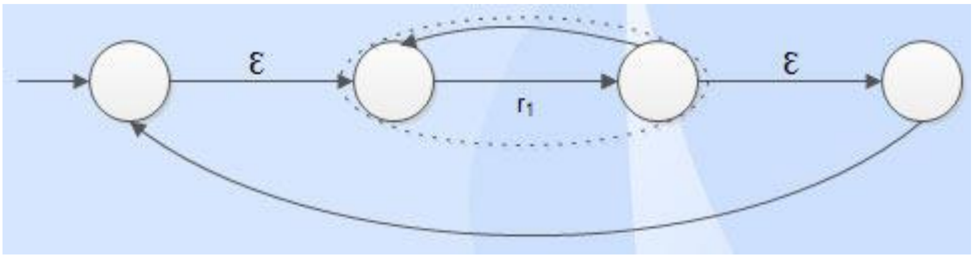
Concatenation

$$\mathbf{r = r_1 \, r_2}$$



Closure

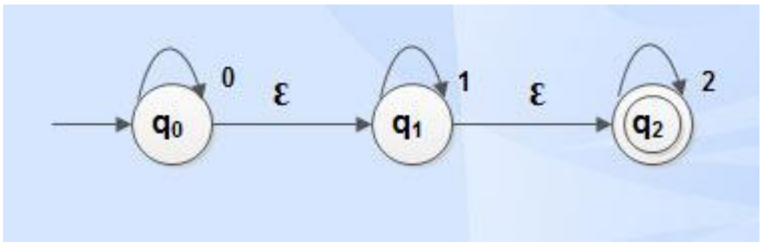
r = r$_1^*$



Ɛ –closure

Ɛ - Closure is the set of states that are reachable from the state concerned on taking empty string as input. It describes the path that consumes empty string (Ɛ) to reach some states of NFA.

Example 1

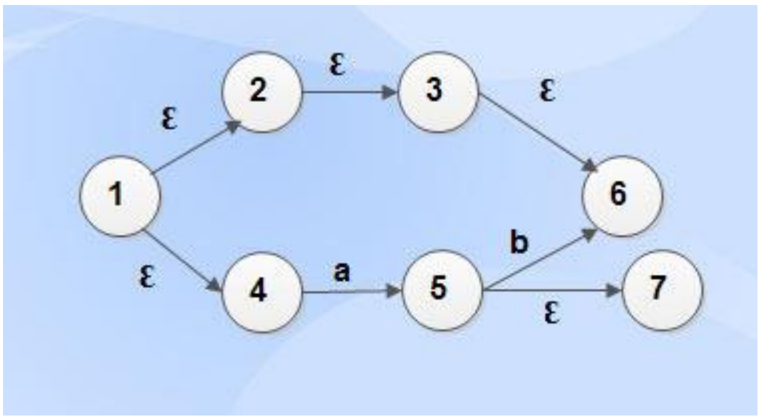


Ɛ -closure(q_0) = { q_0, q_1, q_2}

Ɛ –closure(q_1) = {q_1, q_2}

Ɛ -closure(q_2) = { q_0}

Example 2



Ɛ -closure (l) = {l, 2, 3, 4, 6}

Ɛ-closure (2) = {2, 3, 6}

Ɛ-closure (3) = {3, 6}

Ɛ-closure (4) = {4}

Ɛ-closure (5) = {5, 7}

Ɛ -closure (6) = {6}

Ɛ-closure (7) = {7}

Sub-set Construction

• Given regular expression is converted into NFA.

• Then, NFA is converted into DFA.

Steps

l. Convert into NFA using above rules for operators (union, concatenation and closure) and precedence.

2. Find Ɛ -closure of all states.

3. Start with epsilon closure of start state of NFA.

4. Apply the input symbols and find its epsilon closure.

Dtran [state, input symbol] = Ɛ -closure (move (state, input symbol))

where Dtran transition function of DFA

5. Analyze the output state to find whether it is a new state.

6. If new state is found, repeat step 4 and step 5 until no more new states are found.

7. Construct the transition table for Dtran function.

8. Draw the transition diagram with start state as the Ɛ -closure (start state of NFA) and final state is the state that contains final state of NFA drawn.

Direct Method

Direct method is used to convert given regular expression directly into DFA.

1. Uses augmented regular expression r#.
2. Important states of NFA correspond to positions in regular expression that hold symbols of the alphabet.
3. Regular expression is represented as syntax tree where interior nodes correspond to operators representing union, concatenation and closure operations.
4. Leaf nodes corresponds to the input symbols
5. Construct DFA directly from a regular expression by computing the functions nullable(n), firstpos(n), lastpos(n) andfollowpos(i) from the syntax tree.
6. nullable (n): Is true for * node and node labeled with Ɛ. For other nodes it is false.
7. firstpos (n): Set of positions at node ti that corresponds to the first symbol of the sub-expression rooted at n.
8. lastpos (n): Set of positions at node ti that corresponds to the last symbol of the sub-expression rooted at n.
9. followpos (i): Set of positions that follows given position by matching the first or last symbol of a string generated by sub-expression of the given regular expression.

Rules for computing nullable, firstpos and lastpos

Node n	**nullable** *(n)*	**firstpos** *(n)*	**lastpos** *(n)*
A leaf labeled Ɛ	True	Ø	Ø
A leaf with position i	False	{i}	{i}
An *or* node *n* = c_1| c_2	Nullable (c_1) or Nullable (c_2)	firstpos (c_1) U firstpos (c_2)	Iastpos (c_1) U Iastpos (c_2)
A cat node *n* = c_1c_2	Nullable (c_1) and Nullable (c_2)	If (Nullable (c_1)) firstpos (c_1) U firstpos (c_2) else firstpos (c_1)	If (Nullable (c_2)) lastpos (c_1) U Iastpos (c_2) else lastpos (c_1)
A star node *n* = c_1*	True	firstpos (c_1)	lastpos (c_1)

Computation of followpos

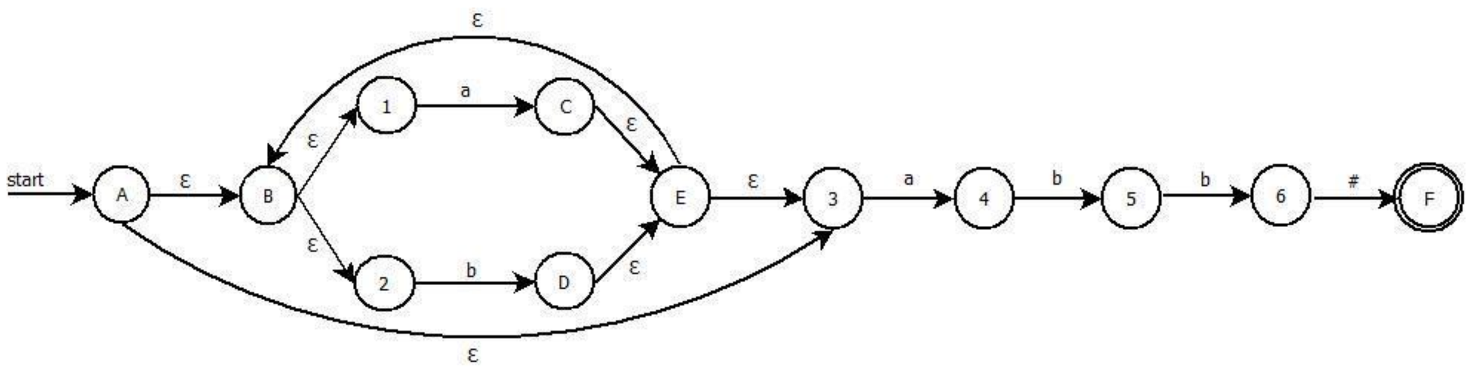
The position of regular expression can follow another in the following ways:

- If *n* is a cat node with left child c_1 and right child c_2, then for every position i in *lastpos(*c_1*),* all positions in *firstpos(*c_2*)* are in *followpos(i).*
- For cat node, for each position i in *lastpos* of its *left child,* the *firstpos* of its *right child* will be in f*ollowpos(i).*
- If *n* is a star node and i is a position in *lastpos(n),* then all positions in *firstpos(n)* are in *followpos(i).*
- For star node, the *firstpos* of that node is in *f ollowpos* of all positions in *lastpos* of that node.

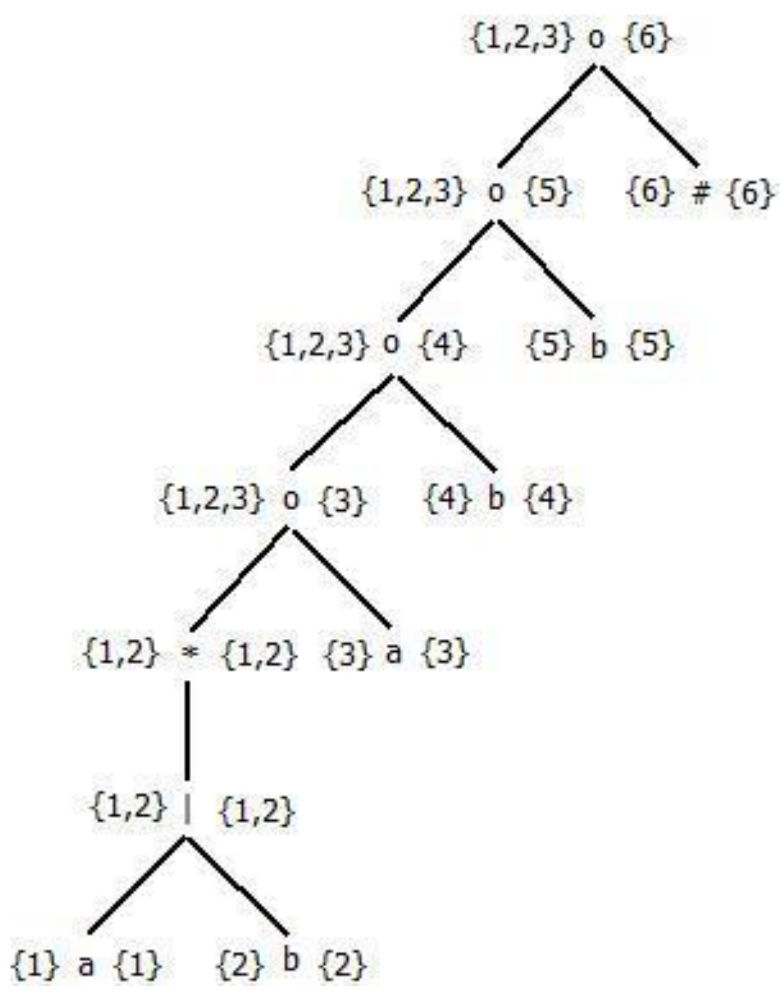
Example:

Thompson's subset construction for

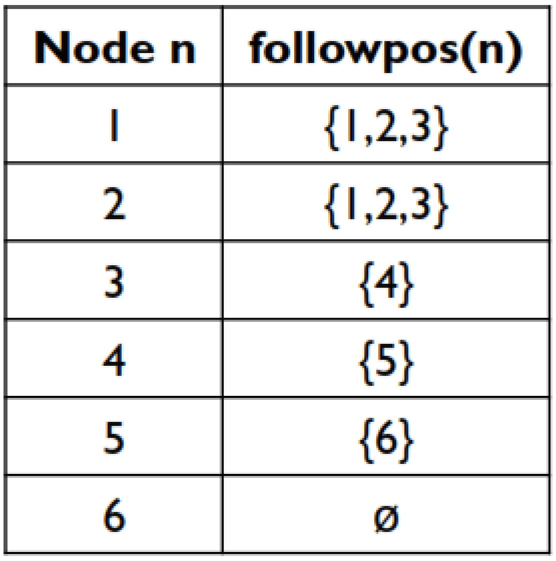
(a+b)*abb



Direct Method for (a+b)*abb #



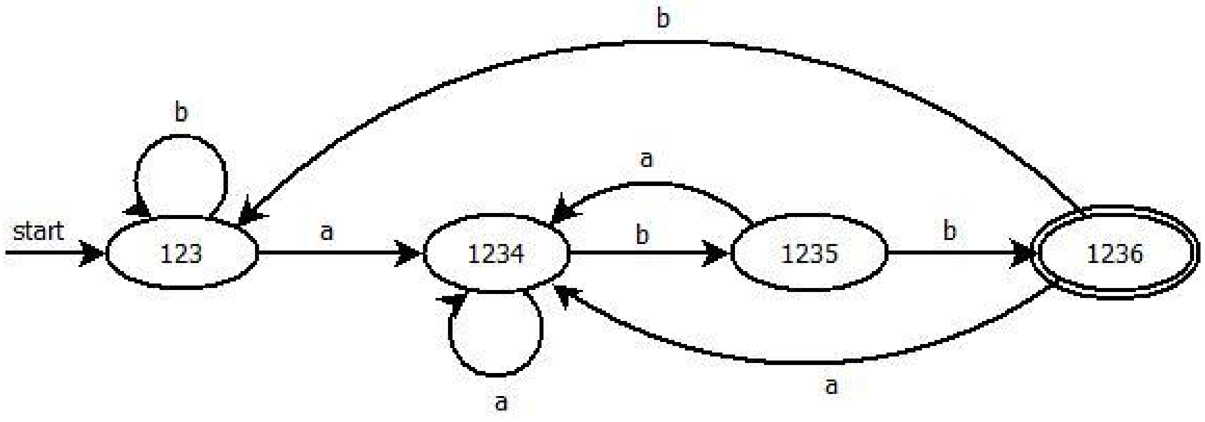
FollowPos



Node n	followpos(n)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	ø

A=firstpos(n_0)={1,2,3}
Dtran[A,a]=
followpos(1) U followpos(3)= {1,2,3,4}=B
Dtran[A,b]=
followpos(2)={1,2,3}=A
Dtran[B,a]=
followpos(1) U followpos(3)=B
Dtran[B,b]=
followpos(2) U followpos(4)={1,2,3,5}=C

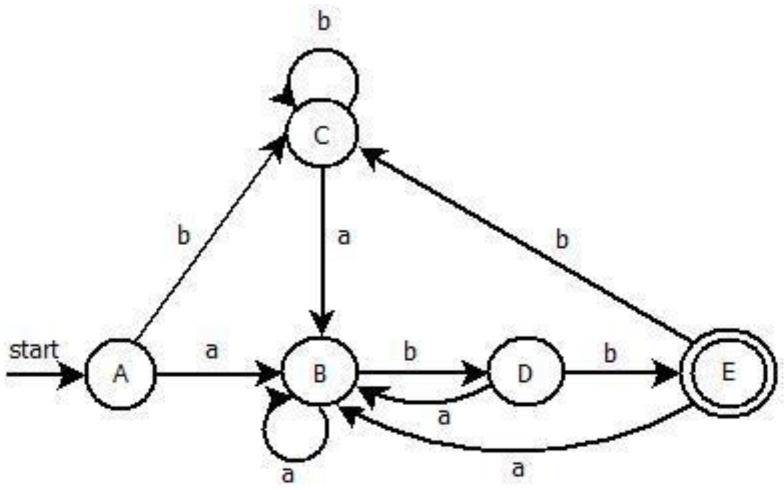
. . . .

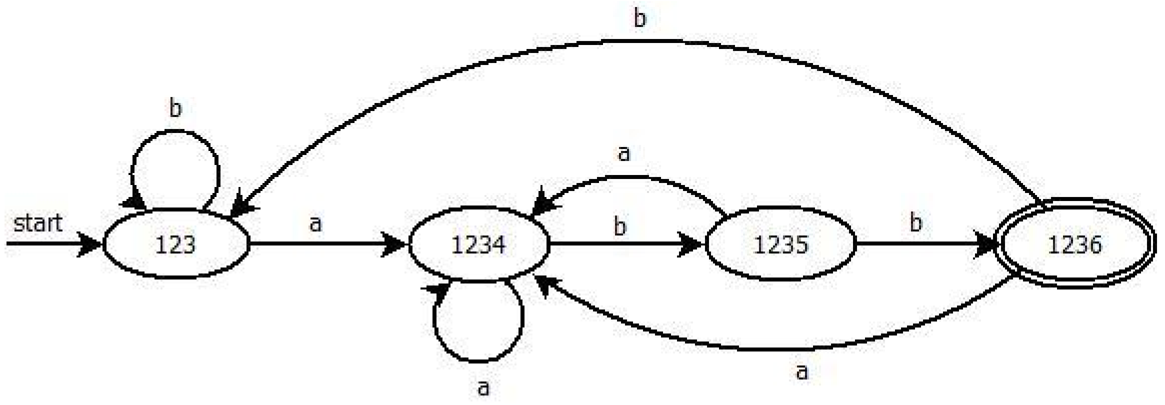


$$\begin{tikzcd}
\text{start} \arrow[r] & 123 \arrow[loop above, "b"] \arrow[r, "a"] & 1234 \arrow[loop below, "a"] \arrow[r, "b"] & 1235 \arrow[l, bend right, "a"'] \arrow[r, "b"] & \boxed{1236} \arrow[lll, bend right=40, "b"'] \arrow[ll, bend left, "a"]
\end{tikzcd}$$

Minimizing the Number of States of a DFA

Equivalent automata
{A, C}=123
{B}=1234
{D}=1235
{E}=1236
Exists a minimum state DFA





A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

There is a wide range of tools for constructing lexical analyzers.

- LEX
- YACC

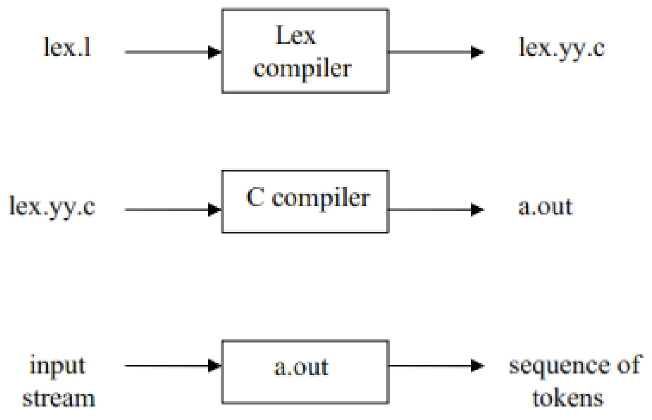
LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

Creating a lexical analyzer

First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

Finally, lex.yy.c is run through the C compiler to produce an object progra m a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specification

A Lex program consists of three parts:

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

- **Definitions** include declarations of variables, constants, and regular definitions
- **Rules** are statements of the form
  p_1 {action$_1$}
  p_2 {action$_2$}
  …
  p_n {action$_n$}
  where p_i is regular expression and action$_i$ describes what action the lexical analyzer should take when pattern p_i matches a lexeme. Actions are written in C code.
- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

Example:

```
%{
 int v=0,c=0;
%}
%%
[aeiouAEIOU] v++;
[a-zA-Z] c++;
%%
main()
{
printf("ENTER INTPUT : \n");
yylex();
printf("VOWELS=%d\nCONSONANTS=%d\n",v,c);
}
```

UNIT-III

SYNTAX ANALYSIS

Need and Role of the Parser-Context Free Grammars -Top Down Parsing -General Strategies-Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item-Construction of SLR Parsing Table -Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC-Design of a syntax Analyzer for a Sample Language .

SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

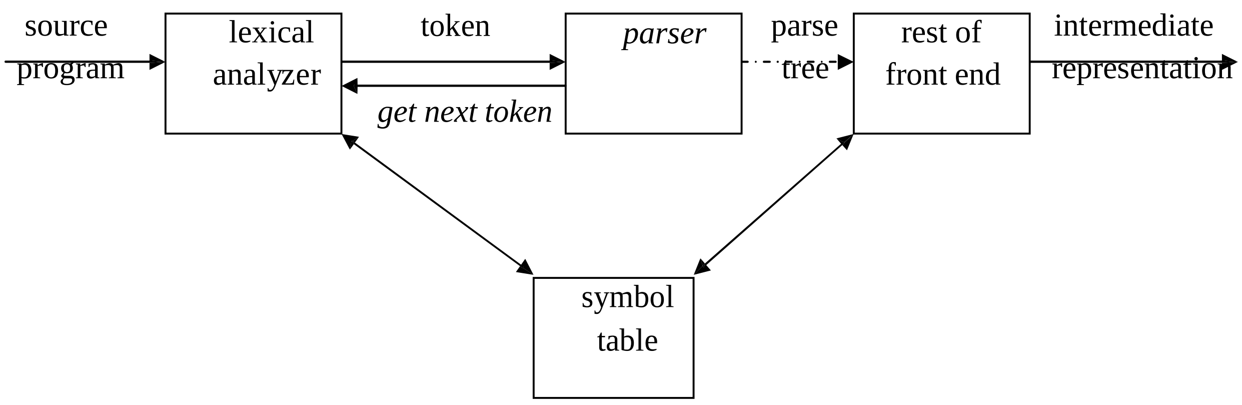
Advantages of grammar for syntactic specification:

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Position of parser in compiler model



Functions of the parser:

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues:

Parser cannot detect errors such as:
1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

Terminals : These are the basic symbols from which strings are formed.

Non-Terminals : These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol : One non-terminal in the grammar is denoted as the “Start-symbol” and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Example of context-free grammar: The following grammar defines simple arithmetic expressions:

$$
\begin{aligned}
expr &\rightarrow expr \ op \ expr \\
expr &\rightarrow (expr) \\
expr &\rightarrow -\, expr \\
expr &\rightarrow \textbf{id} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow / \\
op &\rightarrow \uparrow
\end{aligned}
$$

In this grammar,

- **id** + - * / ↑ ( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

Derivations:

Two basic requirements for a grammar are :
1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example : Consider the following grammar for arithmetic expressions :

$$E \rightarrow E+E \mid E*E \mid (E) \mid - E \mid id$$

To generate a valid string - (id+id) from the grammar the steps are

1. $E \rightarrow - E$
2. $E \rightarrow - (E)$
3. $E \rightarrow - (E+E)$
4. $E \rightarrow - (id+E)$
5. $E \rightarrow - (id+id)$

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.

- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar G : E → E+E | E*E | (E) | - E | id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	**RIGHTMOST DERIVATION** E
→ - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E) E
→ - (id+E)	E → - (E+id) E
→ - (id+id)	E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

Sentinels:

Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

Ambiguity:

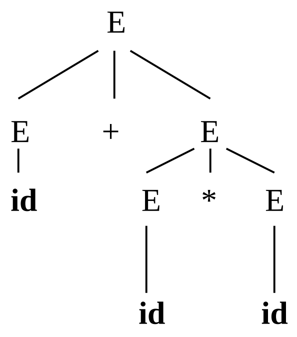
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

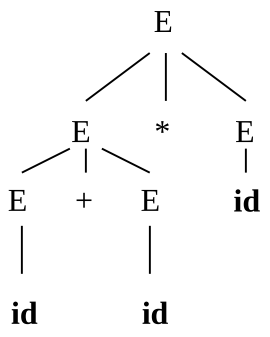
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are :





WRITING A GRAMMAR

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

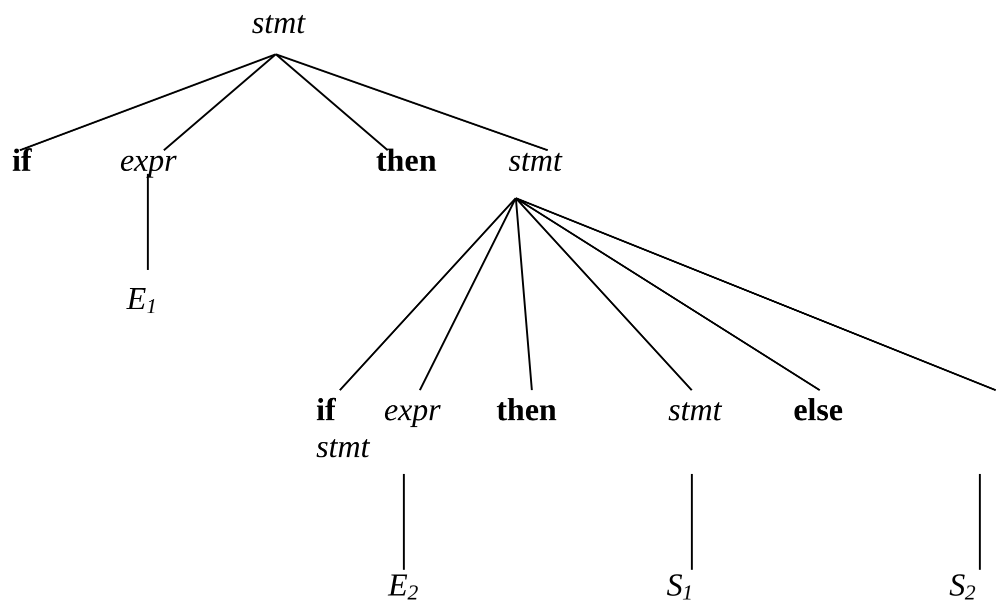
Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

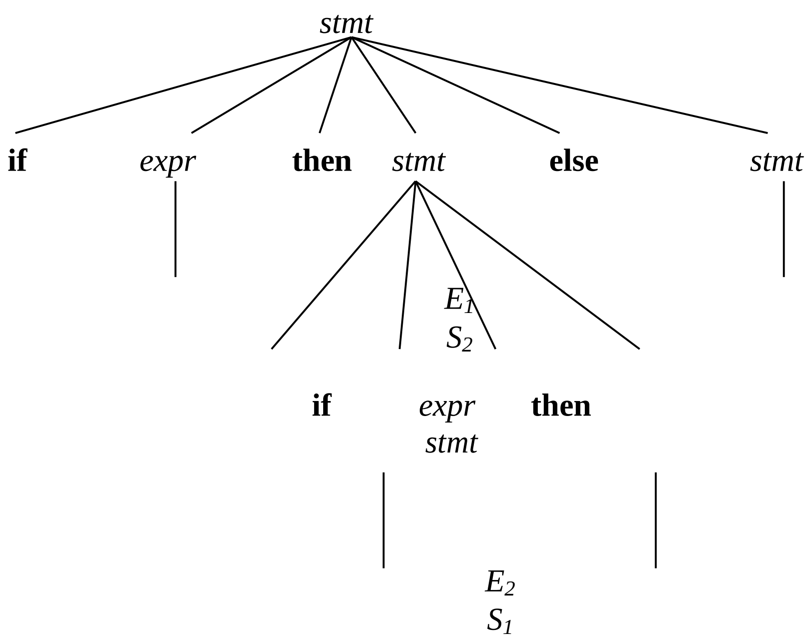
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* **|** **if** *expr* **then** *stmt* **else** *stmt* **|** **other**

This grammar is ambiguous since the string **if E_1 then if E_2 then S_1 else S_2** has the following two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence of two productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

without changing the set of strings derivable from A.

Algorithm to eliminate left recursion:

```
1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
2. for i := 1 to n do begin
       for j := 1 to i-1 do begin
            replace each production of the form $A_i \rightarrow A_j \gamma$
                 by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$
                 where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current $A_j$-productions;
       end
       eliminate the immediate left recursion among the $A_i$-productions
   end
```

Example : Consider the following grammar for arithmetic expressions: E

→ E+T | T

T → T*F | F F

→ (E) | id

First eliminate the left recursion for E as

E → TE’

E’ → +TE’ | ε

Then eliminate for T as

T → FT’ T’→

*FT’ | ε

Thus the obtained grammar after eliminating left recursion is

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A → $\alpha\beta_1$ | $\alpha\beta_2$, it can be rewritten as

A → αA’

A’ → β_1 | β_2

Consider the grammar , G : S → iEtS | iEtSeS | a
E → b

Left factored, this grammar becomes

S → iEtSS’ | a
S’ → eS | ε
E → b

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

- Top–down parsing : A parser can start with the start symbol and try to transform it to the input string.
  Example : LL Parsers.
- Bottom–up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
  Example : LR Parsers.

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

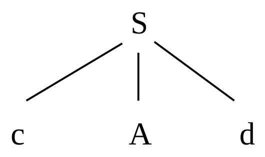
Example for backtracking :

Consider the grammar G : S → cAd
A → ab | a
and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

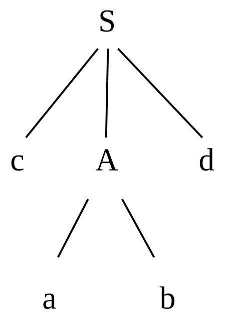
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



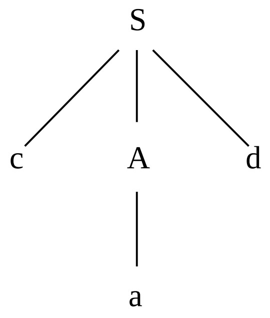
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

<u>**Example for recursive decent parsing**</u>**:**

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating the left-recursion the grammar becomes,

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Now we can write the procedure for grammar as follows:

<u>**Recursive procedure**:</u>

```
Procedure E()
begin
        T( );
        EPRIME( );
end
```

```
Procedure EPRIME( )
begin
        If input_symbol='+' then
        ADVANCE( );
        T( );
        EPRIME( );
end

Procedure T( )
begin
        F( );
        TPRIME( );
end

Procedure TPRIME( )
begin
        If input_symbol='*' then
        ADVANCE( );
        F( );
        TPRIME( );
end

Procedure  F( )
begin
        If input-symbol='id' then
        ADVANCE( );
        else if input-symbol='(' then
        ADVANCE( );
        E( );
        else if input-symbol=')' then
        ADVANCE( );
end

else  ERROR( );
```

Stack implementation:

To recognize input **id+id*id :**

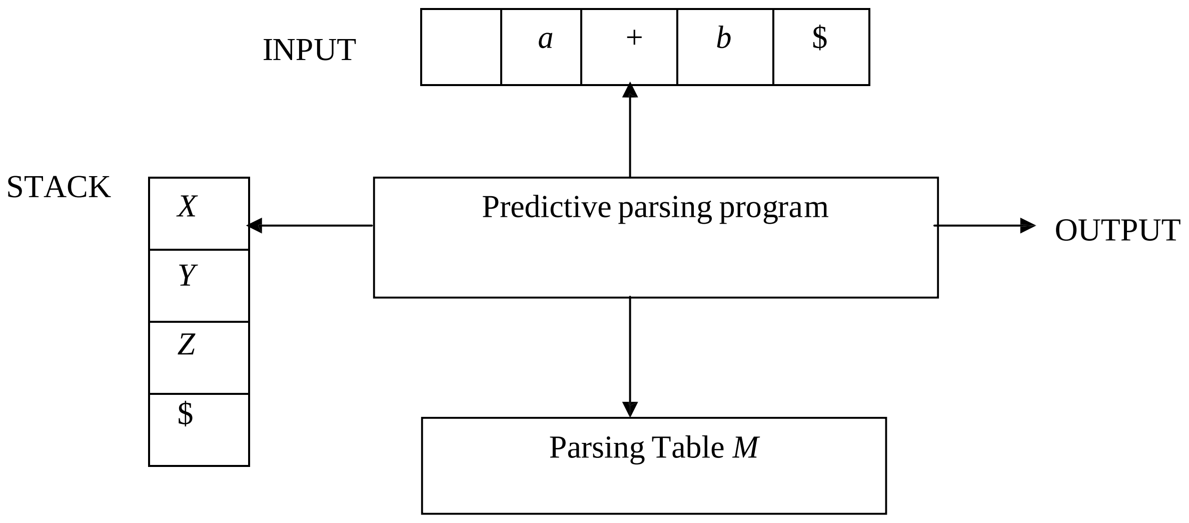
PROCEDURE	**INPUT STRING**
E( )	**id**+id*id
T( )	**id**+id*id
F( )	**id**+id*id
ADVANCE( )	id+id*id

TPRIME( )	id**+**id*id
EPRIME( )	id**+**id*id
ADVANCE( )	id+**id***id
T( )	id+**id***id
F( )	id+**id*** id
ADVANCE( )	id+id*****id
TPRIME( )	id+id*****id
ADVANCE( )	id+id***id**
F( )	id+id***id**
ADVANCE( )	id+id***id**
TPRIME( )	id+id***id**

2. PREDICTIVE PARSING

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by \$ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by \$ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of \$.

Parsing table:

It is a two-dimensional array $M[A, a]$, where **'*A*'** is a non-terminal and **'*a*'** is a terminal.

<u>Predictive parsing program</u>:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
   If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by WVU.
   If $M[X, a] =$ **error**, the parser calls an error recovery routine.

<u>Algorithm for nonrecursive predictive parsing:</u>

Input : A string *w* and a parsing table *M* for grammar *G*.

Output : If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.

Method : Initially, the parser has \$*S* on the stack with *S*, the start symbol of *G* on top, and *w*\$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else            /* X is a non-terminal */
        if $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ then begin
```

```
                pop $X$ from the stack;
                push $Y_k$, $Y_{k-1}$, … ,$Y_1$ onto the stack, with $Y_1$ on top;
                output the production $X \rightarrow Y_1\, Y_2 \ldots Y_k$
            end
            else error()
    until $X$ = $             /* stack is empty */
```

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST

2. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

3. If X is non-terminal and X → aα is a production then add a to FIRST(X).

4. If X is non-terminal and $X \rightarrow Y_1\, Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1),…,FIRST(Y_{i-1}); that is, $Y_1,\ldots.Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST($Y_j$) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.

2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in follow(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.

2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.

3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add $A \rightarrow \alpha$ to $M[A, \$]$.

4. Make each undefined entry of M be **error**.

Example:

Consider the following grammar :

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating left-recursion the grammar is

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

First( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +, $,) }

FOLLOW(T') = { +, $,) }

FOLLOW(F) = {+, * , $,) }

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

<u>LL(1) grammar</u>:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have

S → iEtSS' | a
S'→ eS | ε
E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }

FIRST(S') = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S’) = {

$,e }

FOLLOW(E) =

{t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS’		
S’			S’ → eS S’ → ε			S’ → ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the
grammar: S →
aABe
A → Abc | b
B → d
The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)	**RIGHTMOST DERIVATION**
a**b**bcde (A → b)	**S** → aA**B**e
a**Abc**de (A → Abc)	→ a**A**de
aA**d**e (B → d)	→ a**A**bcde
aABe (S → aABe)	→ abbcde
S	

The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

$E \rightarrow \underline{\mathbf{E+E}}$
$\rightarrow E+\underline{\mathbf{E*E}}$
$\rightarrow E+E\mathbf{*}\underline{\mathbf{id_3}}$
$\rightarrow E+\underline{\mathbf{id_2}}\mathbf{*}id_3$
$\rightarrow \underline{\mathbf{id_1}}+id_2\mathbf{*}id_3$

In the above derivation the underlined substrings are called **handles.**

Handle pruning:

A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if w is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the n^{th} right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	**Input**	**Action**
\$	$id_1+id_2*id_3$ \$	shift
\$ id_1	$+id_2*id_3$ \$	reduce by $E \rightarrow id$
\$ E	$+id_2*id_3$ \$	shift
\$ E+	id_2*id_3 \$	shift
\$ $E+id_2$	$*id_3$ \$	reduce by $E \rightarrow id$
\$ E+E	$*id_3$ \$	shift
\$ E+E*	id3 \$	shift
\$ E+E*id3	\$	reduce by $E \rightarrow id$
\$ E+E*E	\$	reduce by $E \rightarrow E *E$
\$ E+E	\$	reduce by $E \rightarrow E+E$
\$ E	\$	accept

<u>Actions in shift-reduce parser:</u>

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

<u>Conflicts in shift-reduce parsing:</u>

There are two conflicts that occur in shift shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.

2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

1. <u>Shift-reduce conflict:</u>

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	**Input**	**Action**	**Stack**	**Input**	**Action**
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

2. <u>Reduce-reduce conflict:</u>

Consider the grammar:

M → R+R | R+c |
R R → c
and input c+c

Stack	**Input**	**Action**	**Stack**	**Input**	**Action**
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

<u>Viable prefixes:</u>

- α is a viable prefix of the grammar if there is *w* such that α*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ε or has two adjacent non-terminals.

Example:

Consider the grammar:

$E \rightarrow EAE \mid (E) \mid -E \mid id$
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid -E \mid id$

Operator precedence relations:

There are three disjoint precedence relations namely

$<\cdot$ - less than
$=$ - equal to
$\cdot>$ - greater than

The relations give the following meaning:

$a <\cdot b$ – a yields precedence to b
$a = b$ – a has the same precedence as b
$a \cdot> b$ – a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator θ_2, then make
   $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

2. If operators θ_1 and θ_2, are of equal precedence, then make
   $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if operators are left associative
   $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if right associative

3. Make the following for all operators θ:
   $\theta <\cdot id$, $id \cdot> \theta$
   $\theta <\cdot ($, $(<\cdot \theta$
   $) \cdot> \theta$, $\theta \cdot>)$
   $\theta \cdot> \$$, $\$ <\cdot \theta$

Also make
(=) , (<· (,) ·>) , (<· id , id ·>) , $ <· id , id ·> $, $ <· (,) ·> $

Example:

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ is of highest precedence and right-associative
2. * and / are of next higher precedence and left-associative, and
3. + and - are of lowest precedence and left-associative

Note that the **blanks** in the table denote error entries.

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*****	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Operator precedence parsing algorithm:

Input : An input string *w* and a table of precedence relations.
Output : If *w* is well formed, a *skeletal* parse tree **,**with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.
Method : Initially the stack contains $ and the input buffer the string *w* $. To parse, we execute the following program :

```
(1) Set ip to point to the first symbol of w$;
(2) repeat forever
(3)    if  $ is on top of the stack and ip points to $ then
(4)       return
       else begin
(5)        let a be the topmost terminal symbol on the stack
              and let b be the symbol pointed to by ip;
(6)        if a <· b or a = b then begin
(7)           push b onto the stack;
(8)           advance ip to the next input symbol;
           end;
```

```
(9)      else if a ·> b then                /*reduce*/
(10)        repeat
(11)            pop the stack
(12)        until the top stack terminal is related by <·
                to the terminal most recently popped
(13)     else error( )
       end
```

Stack implementation of operator precedence parsing:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK	INPUT
$	w $

where w is the input string to be parsed.

Example:

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is **id+id*id** .The implementation is as follows:

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the '*k*' for the number of input symbols. When '*k*' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Drawbacks of LR method:

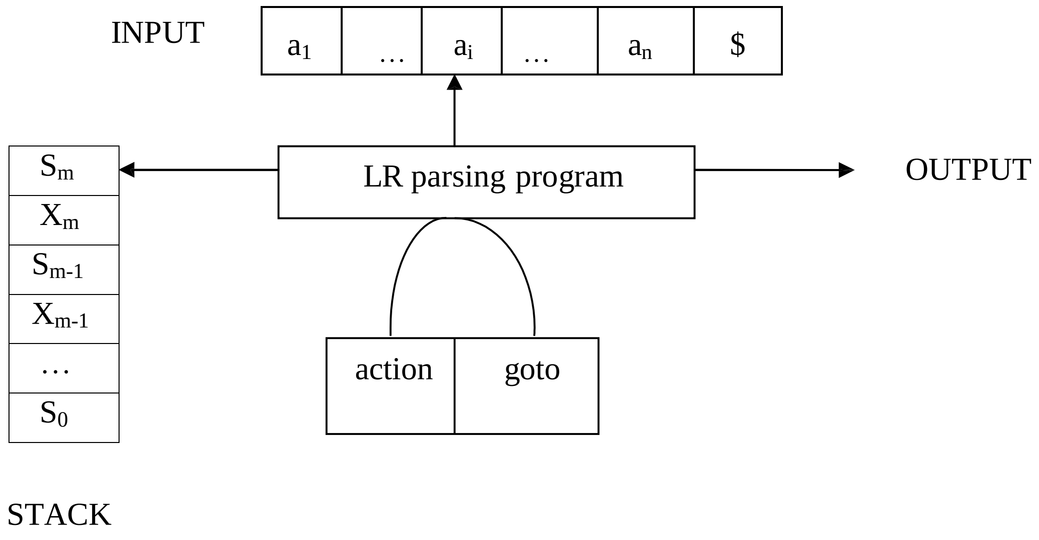
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look-Ahead LR
   - Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults *action*[s_m,a_i] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever begin
          let s be the state on top of the stack and
               a the symbol pointed to by ip;
          if action[s, a] = shift s’ then begin
               push a then s’ on top of the stack;
               advance ip to the next input symbol
          end
          else if action[s, a] = reduce A→β then begin
                 pop 2* | β | symbols off the stack;
                 let s’ be the state now on top of the stack;
                 push A then goto[s’, A] on top of the stack;
                 output  the production A→ β
          end
          else if action[s, a] = accept then
                 return
          else error( )
end
```

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:
1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → XYZ **.**

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

<u>**Algorithm for construction of SLR parsing table:**</u>

Input : An augmented grammar G'
Output : The SLR parsing table functions *action* and *goto* for G'
Method :
1. Construct C = {I_0, I_1, …. I_n}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If [A→α·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If [A→α·] is in I_i , then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(I_i,A) = I_j, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T
T → T * F | F
F → (E) | id

The given grammar is :

G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E' → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2 : Find LR (0) items.

I_0 : E' → **.** E
E → **.** E + T
E → **.** T
T → **.** T * F
T → **.** F
F → **.** (E)
F → **.** id

GOTO (I_0 , E)
I_1 : E' → E **.**
E → E **.** + T

GOTO (I_4 , id)
I_5 : F → id **.**

GOTO (I_0 , T)

$I_2 : E \rightarrow T .$
$\quad T \rightarrow T . * F$

GOTO (I_0 , F)

$I_3 : T \rightarrow F .$

GOTO (I_0 , ( )

$I_4 : F \rightarrow (. E)$
$\quad E \rightarrow . E + T$
$\quad E \rightarrow . T$
$\quad T \rightarrow . T * F$
$\quad T \rightarrow . F$
$\quad F \rightarrow . (E)$
$\quad F \rightarrow . id$

GOTO (I_0 , id)

$I_5 : F \rightarrow id .$

GOTO (I_1 , +)

$I_6 : E \rightarrow E + . T$
$\quad T \rightarrow . T * F$
$\quad T \rightarrow . F$
$\quad F \rightarrow . (E)$
$\quad F \rightarrow . id$

GOTO (I_2 , *)

$I_7 : T \rightarrow T * . F$
$\quad F \rightarrow . (E)$
$\quad F \rightarrow . id$

GOTO (I_4 , E)

$I_8 : F \rightarrow (E .)$
$\quad E \rightarrow E . + T$

GOTO (I_4 , T)

$I_2 : E \rightarrow T .$
$\quad T \rightarrow T . * F$

GOTO (I_4 , F)

$I_3 : T \rightarrow F .$

GOTO (I_6 , T)

$I_9 : E \rightarrow E + T .$
$\quad T \rightarrow T . * F$

GOTO (I_6 , F)

$I_3 : T \rightarrow F .$

GOTO (I_6 , ( )

$I_4 : F \rightarrow (. E)$

GOTO (I_6 , id)

$I_5 : F \rightarrow id .$

GOTO (I_7 , F)

$I_{10} : T \rightarrow T * F .$

GOTO (I_7 , ( )

$I_4 : F \rightarrow (. E)$
$\quad E \rightarrow . E + T$
$\quad E \rightarrow . T$
$\quad T \rightarrow . T * F$
$\quad T \rightarrow . F$
$\quad F \rightarrow . (E)$
$\quad F \rightarrow . id$

GOTO (I_7 , id)

$I_5 : F \rightarrow id .$

GOTO (I_8 ,))

$I_{11} : F \rightarrow (E) .$

GOTO (I_8 , +)

$I_6 : E \rightarrow E + . T$
$\quad T \rightarrow . T * F$
$\quad T \rightarrow . F$
$\quad F \rightarrow . (E)$
$\quad F \rightarrow . id$

GOTO (I_9 , *)

$I_7 : T \rightarrow T * . F$
$\quad F \rightarrow . (E)$
$\quad F \rightarrow . id$

GOTO (I_4 , ( )

I_4 : F $\rightarrow$ (**.** E)

E $\rightarrow$ **.** E + T

E $\rightarrow$ **.** T

T $\rightarrow$ **.** T * F

T $\rightarrow$ **.** F

F $\rightarrow$ **.** (E)

F $\rightarrow$ id

FOLLOW (E) = { $,) , +)

FOLLOW (T) = { $, + ,) , * }

FOOLOW (F) = { * , + ,) , $ }

SLR parsing table:

	ACTION						**GOTO**		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

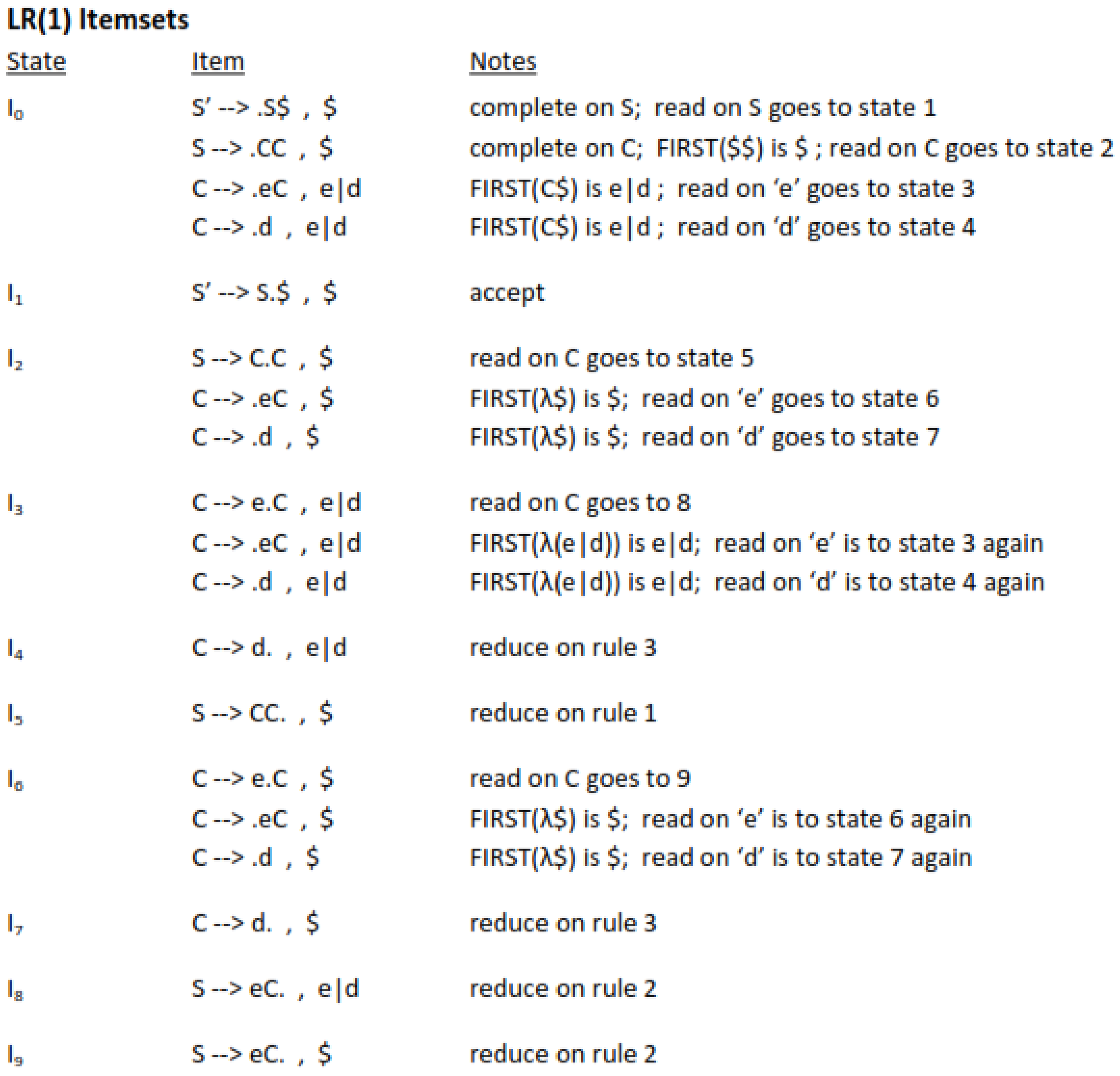
Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**

Building LR(1) itemsets, LR(1) and LALR parse tables

A, S, X: non-terminals
x,y, a, ß: string of terminals and/or non-terminals
C: one terminal or one non-terminal
Start: [S --> . w , $] is the item associated with the start state.
Read: Starting a new state (reading on one terminal or non-terminal, C) comes from
[A --> x.Cy , w] then new state includes [A --> xC.y , w] .
Complete: if [A --> x . X a , u] is an item, then completing on X gives the item(s) [X --> .ß , z] where z ϵ FIRST(au)
Consider the augmented grammar G':

0. S' --> S$
1. S --> CC
2. C --> eC
3. C --> d



LR(1) Itemsets

State	Item	Notes
I_0	S' --> .S$, $	complete on S; read on S goes to state 1
	S --> .CC , $	complete on C; FIRST($$) is $; read on C goes to state 2
	C --> .eC , e|d	FIRST(C$) is e|d ; read on 'e' goes to state 3
	C --> .d , e|d	FIRST(C$) is e|d ; read on 'd' goes to state 4
I_1	S' --> S.$, $	accept
I_2	S --> C.C , $	read on C goes to state 5
	C --> .eC , $	FIRST(λ$) is $; read on 'e' goes to state 6
	C --> .d , $	FIRST(λ$) is $; read on 'd' goes to state 7
I_3	C --> e.C , e|d	read on C goes to 8
	C --> .eC , e|d	FIRST(λ(e|d)) is e|d; read on 'e' is to state 3 again
	C --> .d , e|d	FIRST(λ(e|d)) is e|d; read on 'd' is to state 4 again
I_4	C --> d. , e|d	reduce on rule 3
I_5	S --> CC. , $	reduce on rule 1
I_6	C --> e.C , $	read on C goes to 9
	C --> .eC , $	FIRST(λ$) is $; read on 'e' is to state 6 again
	C --> .d , $	FIRST(λ$) is $; read on 'd' is to state 7 again
I_7	C --> d. , $	reduce on rule 3
I_8	S --> eC. , e|d	reduce on rule 2
I_9	S --> eC. , $	reduce on rule 2

The SLR parse table:

	e	d	$		S	C
0	s3	s4			1	2
1			accept			
2	s3	s4				5
3	s3	s4				6
4	r3	r3	r3			
5			r1			
6	r2	r2	r2			

The LR(1) parse table (same as before, except when you do a reduce – items with dot at end – instead of using the whole FOLLOW set, only use symbols after the comma):

	e	d	$		S	C

To create LALR table, merge states by their core sets (for state use either just first number, or use all the original state numbers to create a unique longer number):

	e	d	$		S	C
0	s36	s47			1	2
1			accept			
2	s36	s47				5
36	s36	s47				89
47	r3	r3	r3			
5			r1			
89	r2	r2	r2			

To create LALR table, merge states by their core sets (for state use either just first number, or use all the original state numbers to create a unique longer number):

	e	d	$		S	C
0	s36	s47			1	2
1			accept			
2	s36	s47				5
36	s36	s47				89
47	r3	r3	r3			
5			r1			
89	r2	r2	r2			

Syntax error handling :

Programs can contain errors at many different levels. For example :
1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler :

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

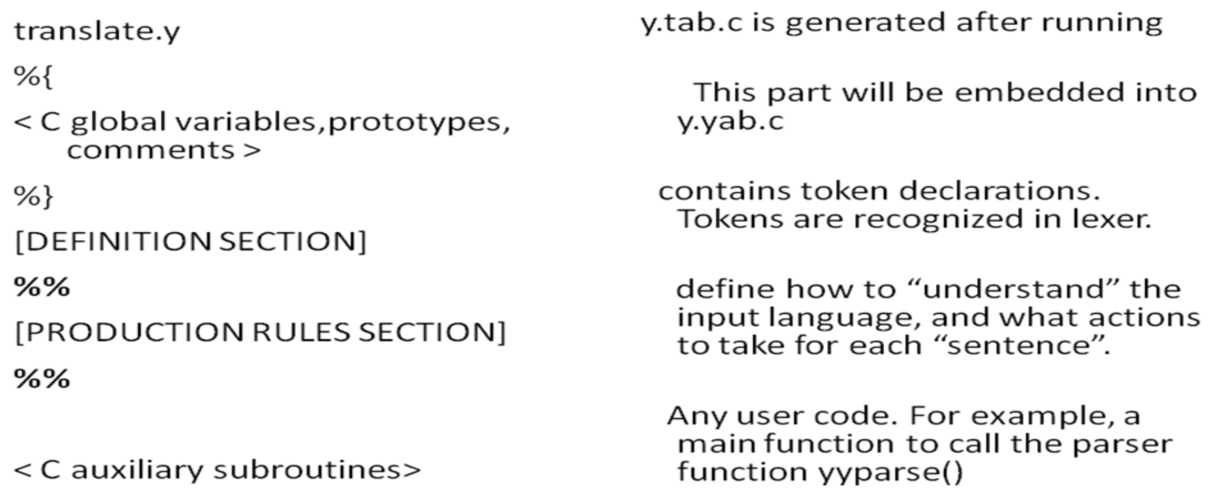
Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

<u>YACC-Design of a syntax Analyzer for a Sample Language</u>

- Yacc is a tool for constructing parsers.
- It reads a specification file that codifies the grammar of a language and generates a parsing routine.
- Yacc specification describes a CFG, that can be used to generate a parser.
- Elements of a CFG:
    1. Terminals: tokens and literal characters,
    2. Variables (nonterminals): syntactical elements,
    3. Production rules, and
    4. Start rule.

Skeleton of a yacc specification (.y file)



```
translate.y                                  y.tab.c is generated after running

%{                                             This part will be embedded into
< C global variables,prototypes,              y.yab.c
     comments >
%}                                           contains token declarations.
[DEFINITION SECTION]                          Tokens are recognized in lexer.

%%                                            define how to "understand" the
[PRODUCTION RULES SECTION]                    input language, and what actions
                                              to take for each "sentence".
%%
                                             Any user code. For example, a
                                              main function to call the parser
< C auxiliary subroutines>                    function yyparse()
```

Example:

A -> Bc   is written in yacc as   a: b 'c';

Format of a yacc specification file:

declarations %% *grammar rules and associatedactions*

Declarations:

To define tokens and their characteristics

%token:	declare names of tokens
%left:	define left-associative operators
%right:	define right-associative operators
%nonassoc:	define operators that may not associate with themselves
%type:	declare the type of variables
%union:	declare multiple data types for semantic values
%start:	declare the start symbol (default is the first variable in rules)
%prec:	assign precedence to a rule

%{
    C declarations   directly copied to the resulting C program
%}   (E.g., variables, types, macros…)

Eg:Yacc program to recognize L = {a^{n}b^n | *n* >=0}.

```
%{
#include<stdio.h>
int valid=1;
%}
%token A B
%%
str:S'\n' {return 0;}
S:A S B
 |
 ;
%%
main()
{
   printf("Enter the string:\n");
   yyparse();
   if(valid==1)
   printf("\nvalid string");
}
```

UNIT IV- SYNTAX DIRECTEDTRANSLATION & RUN TIME ENVIRONMENT

Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-Attribute Definitions- Design of predictive translator - Type Systems-Specification of a simple type checker-Equivalence of Type Expressions-Type Conversions. RUN-TIME ENVIRONMENT: Source Language Issues-Storage Organization-Storage Allocation-Parameter Passing-Symbol Tables-Dynamic Storage Allocation-Storage Allocation in FORTAN.

SEMANTIC ANALYSIS

- ➢ Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- ➢ In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- ➢ The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- ➢ As for Lexical and Syntax analysis, also for Semantic Analysis we need both a Representation Formalism and an Implementation Mechanism.
- ➢ As representation formalism this lecture illustrates what are called Syntax Directed Translations.

SYNTAX DIRECTED TRANSLATION

- ➢ The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- ➢ By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - o We associate Attributes to the grammar symbols representing the language constructs.
  - o Values for attributes are computed by Semantic Rules associated with grammar productions.
- ➢ Evaluation of Semantic Rules may:
  - o Generate Code;
  - o Insert information into the Symbol Table;
  - o Perform Semantic Check;
  - o Issue error messages;
  - o etc.

There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).
2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

• **Syntax Directed Definitions** are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of **Attributes**;
2. Productions are associated with **Semantic Rules** for computing the values of attributes.
   - Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,X.a indicates the attribute a of the grammar symbol X).
   - The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.
2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes

Syntax Directed Definitions: An Example

• **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

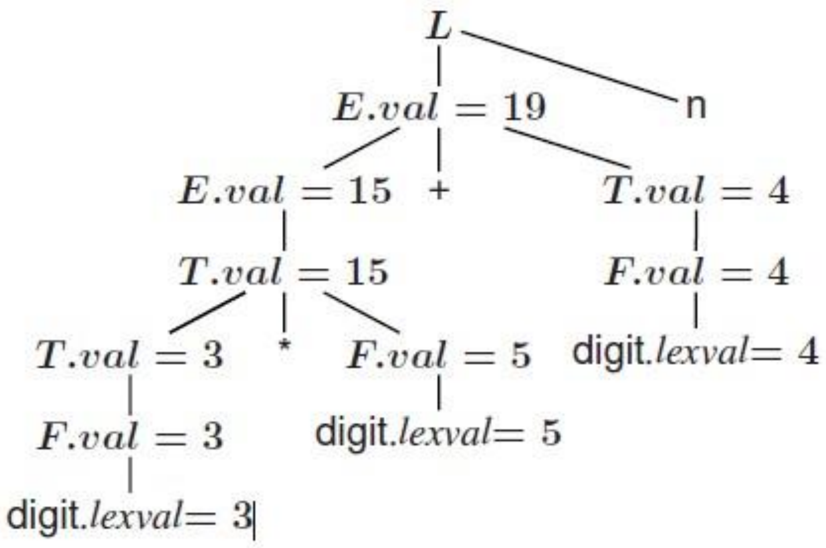
PRODUCTION	SEMANTIC RULE
$L \rightarrow E\text{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \text{digit}$	$F.val := \text{digit}.lexval$

S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

• **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

• **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



L-attributed definition

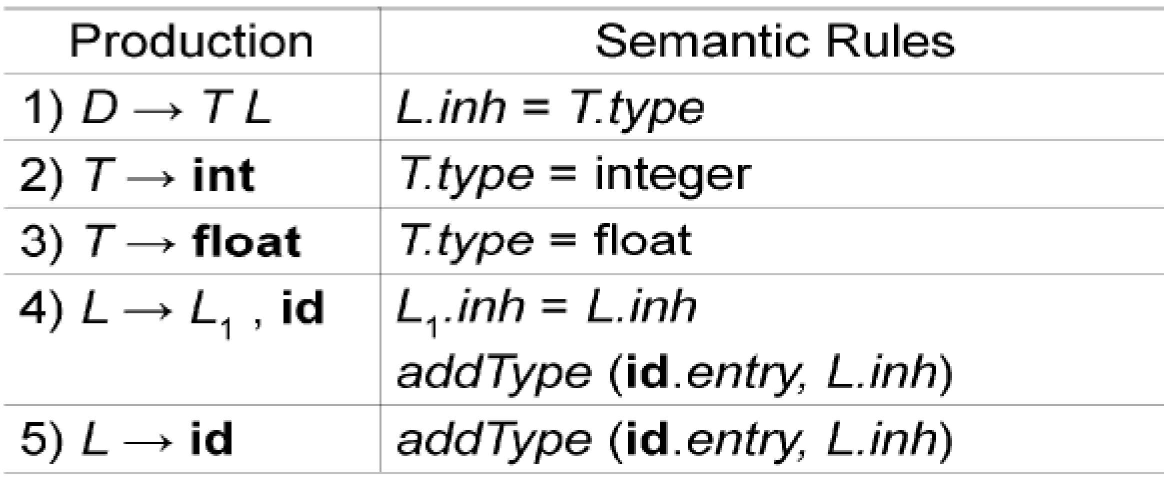
Definition: A SDD its *L-attributed* if each inherited attribute of Xi in the RHS of A ! X1 : :Xn depends only on

1. attributes of X1;X2; : : : ;Xi1 (symbols to the left of Xi in the RHS)
2. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.
2. An action must not refer to synthesized attribute of any symbol to the right of that action.
3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

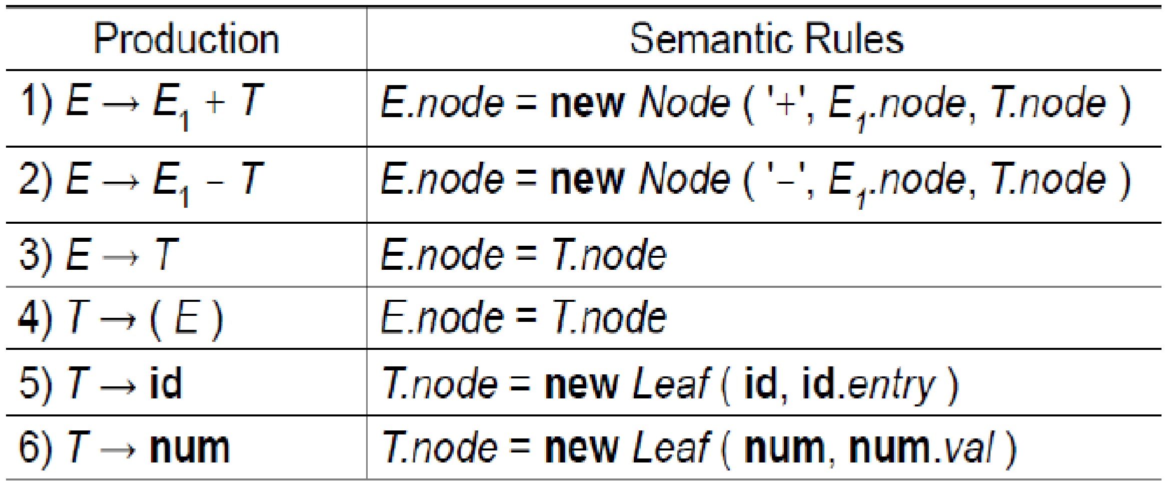
SDD For Simple Type Declarations



Production	Semantic Rules
1) $D \rightarrow T\ L$	$L.inh = T.type$
2) $T \rightarrow$ **int**	$T.type$ = integer
3) $T \rightarrow$ **float**	$T.type$ = float
4) $L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ *addType* (**id**.*entry*, *L.inh*)
5) $L \rightarrow$ **id**	*addType* (**id**.*entry*, *L.inh*)

CONSTRUCTION OF SYNTAX TREE

- SDDs are useful for is construction of syntax trees. A syntax tree is a condensed form of parse tree.
- Syntax trees are useful for representing programming language constructs like expressions and statements.
- They help compiler design by decoupling parsing from translation.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
  e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2
- Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:
  i)If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function Leaf(op, val)
  ii)If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function Node(op, c1, c2,...,ck) .
- Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute node, which represents a node of the syntax
  tree.



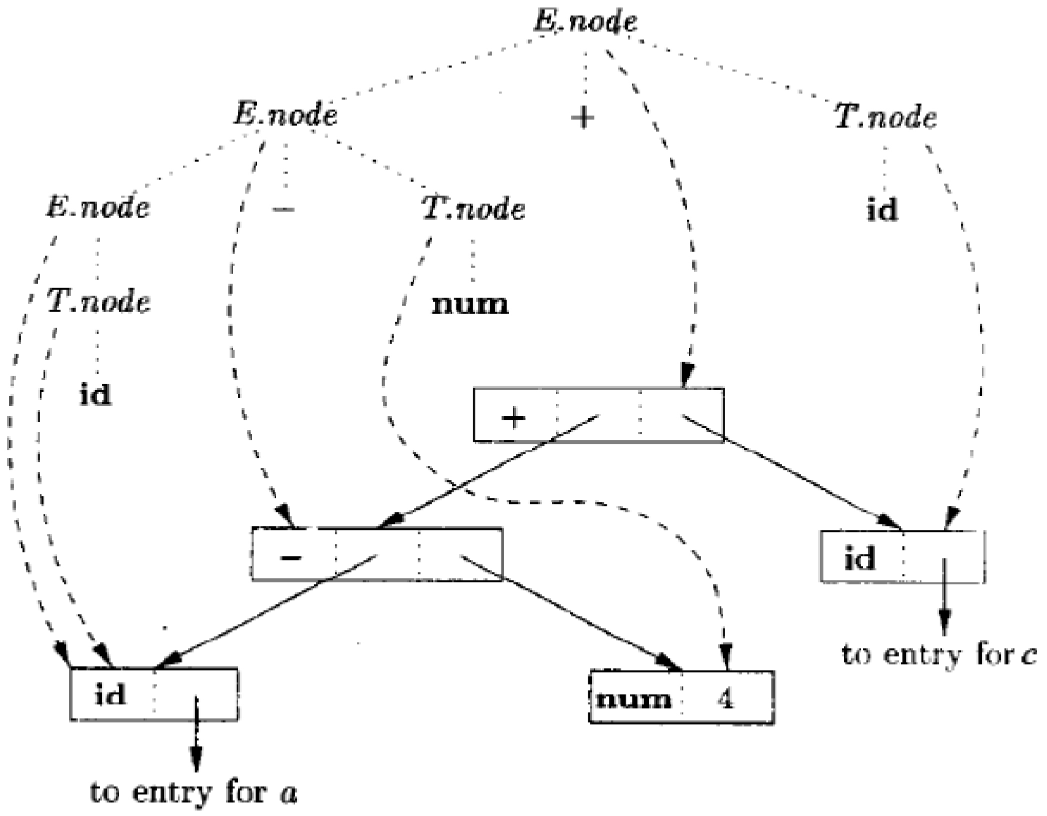
Production	Semantic Rules
1) $E \rightarrow E_1 + T$	$E.node$ = **new** $Node$ ('+', $E_1.node$, $T.node$)
2) $E \rightarrow E_1 - T$	$E.node$ = **new** $Node$ ('−', $E_1.node$, $T.node$)
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$E.node = T.node$
5) $T \rightarrow$ **id**	$T.node$ = **new** $Leaf$ (**id**, **id**.$entry$)
6) $T \rightarrow$ **num**	$T.node$ = **new** $Leaf$ (**num**, **num**.val)

Steps in the construction of the syntax tree for a-4+c

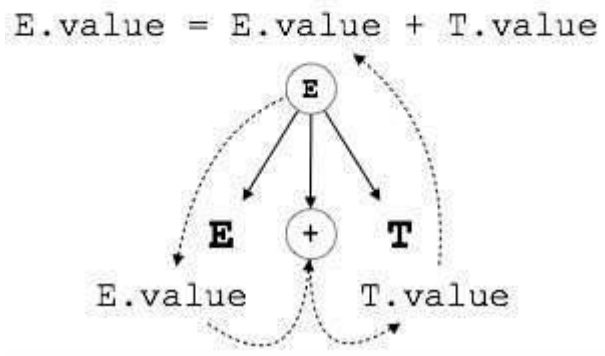
If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

1) p_1 = **new** *Leaf* (**id**, *entry-a*);
2) p_2 = **new** *Leaf* (**num**, 4);
3) p_3 = **new** *Node* ('–', p_1, p_2);
4) p_4 = **new** *Leaf* (**id**, *entry-c*);
5) p_5 = **new** *Node* ('+', p_3, p_4);

Syntax tree for a-4+c using the above SDD is shown below.



Bottom-up Evaluation of S-Attribute Definitions

- Syntax-directed definition with only **synthesized** attributes is called S-attributed
- Use LR Parser
- Implementation:
- Stack to hold info about subtrees that have been parsed
- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- Synthesized attributes
    - A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N it
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).
  

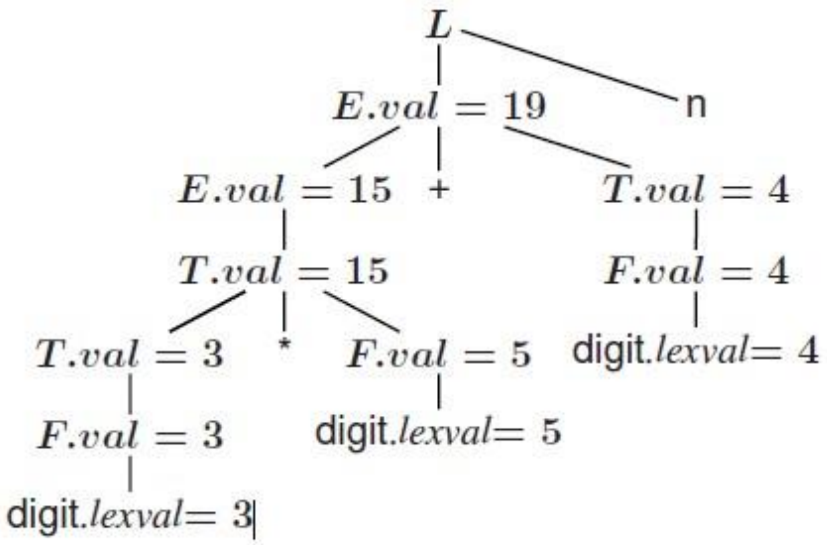
-
- As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

Syntax Directed Definitions: An Example

• **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
$L \to E\text{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \text{digit}$	$F.val := \text{digit}.lexval$

• The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



Input	Stack	Attribute	Production Used
3 * 5 + 4 $	-	-	
* 5 + 4 $	3	3	
* 5 + 4 $	F	3	F ---> digit
* 5 + 4 $	T	3	T ---> F
5 + 4 $	T *	3	
+ 4 $	T * 5	3 * 5	
+ 4 $	T * F	3 * 5	F -→ digit
+ 4 $	T	15	T ---> T * F
+ 4 $	E	15	E ---> T
4 $	E +	15	
$	E + 4	15 + 4	
$	E + F	15 + 4	F ---> digit
$	E + T	15 4	T ---> F
$	E	19	E ---> E + T
	E	19	
	L	19	L ---> E $

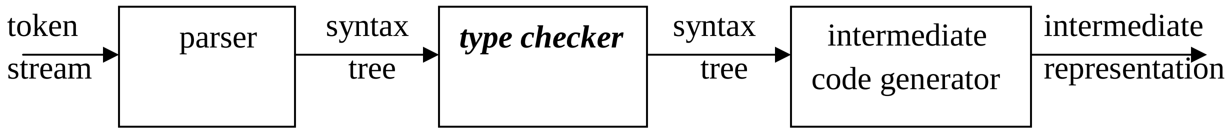
TYPE CHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language.
This checking, called *static checking*, detects and reports

programming errors. Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.
2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



- A ***type checker*** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

- The type of a language construct will be denoted by a "type expression."
- A type expression is either a basic type or is formed by applying an operator called a ***type constructor*** to other type expressions.
- The sets of basic types and constructors depend on the language to be

checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

   A special basic type, *type_error* , will signal an error during type checking; *void* denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression. Constructors include:
   Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products : If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression.

Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

```
type row = record
                    address: integer;
                    lexeme: array[1..15] of char
                  end;
      var table: array[1...101] of row;
```

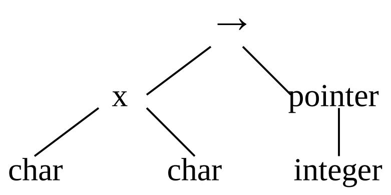
declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".
For example, ***var p:*** ↑ ***row*** declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R.* The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → pointer (integer)



Type systems

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic.
- Any check can be done dynamically, if the target code carries the type of an element

along with the value of that element.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

<u>A Simple Language</u>

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E
↑

Translation scheme:

P → D ; E	
D → D ; D	
D → id : T	{ *addtype* (id.*entry* , T.*type*) }
T → char	{ T.*type* : = char }
T → integer	{ T.*type* : = integer }
T → ↑ T1	{ T.*type* : = pointer(T1.*type*) }
T → array [num] of T1	{ T.*type* : = array (1… num.val , T1.*type*) }

In the above language,
→ There are two basic types : char and integer ;

→ *type_error* is used to signal errors;
→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression ***pointer (integer)***.

Type checking of expressions

In the following rules, the attribute *type* for E gives the type expression assigned to the expression generated by E.

1. E → **literal** { E.*type* : = *char* }
   E → **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. E → **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E1 **mod** E2 { *E.type* : = **if** *E1. type* = *integer* **and**
 E2. type = *integer* **then** *integer*
 else *type_error* }
   The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. E → E1 [E2] { *E.type* : = if *E2.type* = *integer* **and**
 E1.type = *array(s,t)* **then** *t*
 else *type_error* }
   In an array reference E1 [E2] , the index expression E2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E1.

5. E → E1 ↑ { *E.type* : = **if** *E1.type* = *pointer (t)* **then** *t*
 else *type_error* }

   The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type *t* of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:
 S → **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void*
 else *type_error* }

2. Conditional statement:
 S → **if** E **then** S1 { S.*type* : = **if** E.*type* = *boolean* **then** S1.*type*
 else *type_error* }

3. While statement:

S → while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

4. Sequence of statements:

S → S_1 ; S_2 { S.*type* : = **if** S_1.*type* = *void* and
S_1.*type* = *void* **then** *void*
else *type_error* }

<u>Type checking of functions</u>

The rule for checking the type of a function application is :

E → E_1 (E_2) { E.*type* : = **if** E_2.*type* **=** *s* **and**
E_1.*type* **=** *s* → *t* **then** *t*
else *type_error* }

RUNTIME ENVIRONMENT

- ➢ Runtime organization of different storage locations
- ➢ Representation of scopes and extents during program execution.
- ➢ Components of executing program reside in blocks of memory (supplied by OS).
- ➢ Three kinds of entities that need to be managed at runtime:
  - o Generated code for various procedures and programs.
- • forms text or code segment of your program: size known at compile time.
  - o Data objects:
- • Global variables/constants: size known at compile time
- • Variables declared within procedures/blocks: size

known • Variables created dynamically: size unknown.

- o Stack to keep track of procedure
- • activations. Subdivide memory conceptually
  into code and data areas:

▪ Cod

e: Program •
instructions

- ▪ Stack: Manage activation of procedures at runtime.
- ▪ Heap: holds variables created dynamically

SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name,* and the statement is the *procedure body*.

For example, the following is the definition of procedure named *readarray* :

```
procedure readarray;
var i : integer;
begin
    for i : = 1 to 9 do read(a[i])
end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.

- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name.
Declarations may be explicit, such as:

    var i : integer ;

or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

The portion of the program to which a declaration applies is called the ***scope*** of that declaration.

Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. “Data object” corresponds to a storage location that holds values.

The term *environment* refers to a function that maps a name to a storage location.
The term *state* refers to a function that maps a storage location to the value held there.

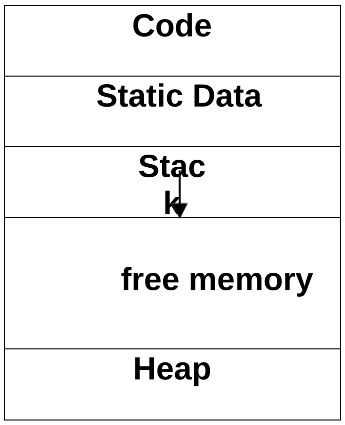
$$\begin{tikzcd} \text{name} \arrow[r, bend left, "\textit{environment}"] & \text{storage} \arrow[r, bend left, "\textit{state}"] & \text{value} \end{tikzcd}$$

When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

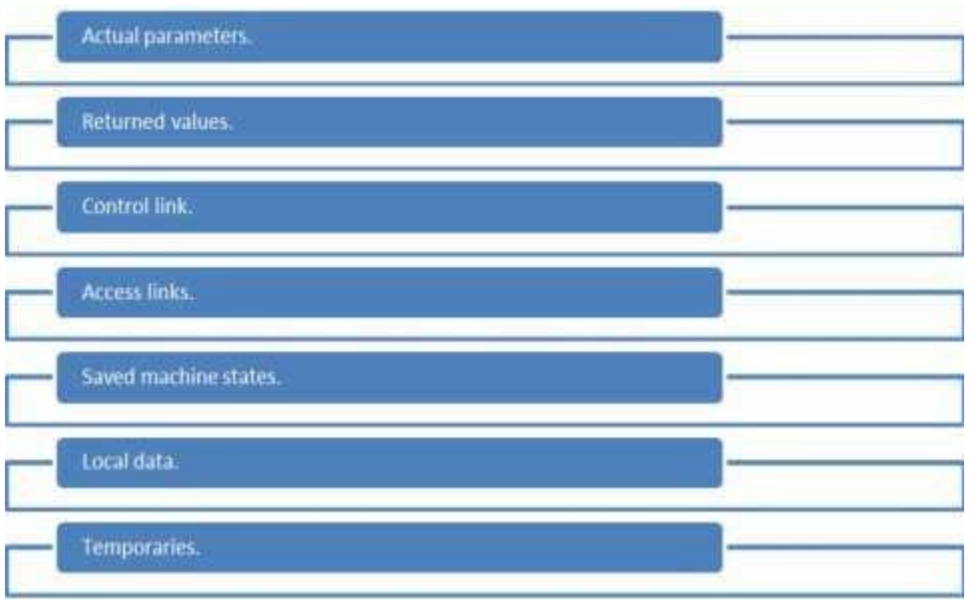
Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

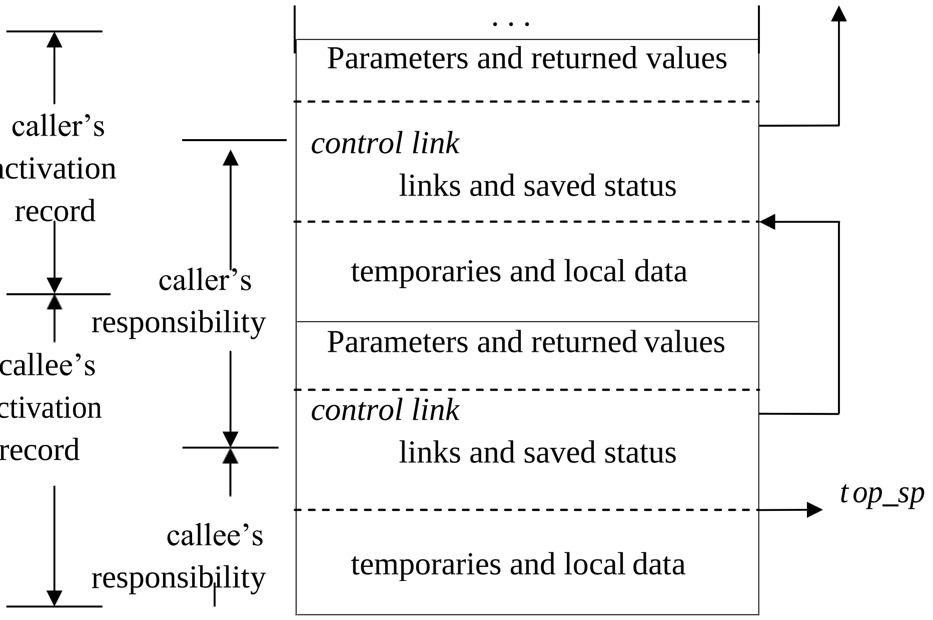
STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - ➢ Values communicated between caller and callee are generally placed at the beginning of the callee’s activation record, so they are as close as possible to the caller’s activation record.

- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.



. . .
Parameters and returned values
control link
links and saved status
temporaries and local data
Parameters and returned values
control link
links and saved status
temporaries and local data
caller's activation record
callee's activation record
caller's responsibility
callee's responsibility
top_sp

Division of tasks between caller and callee

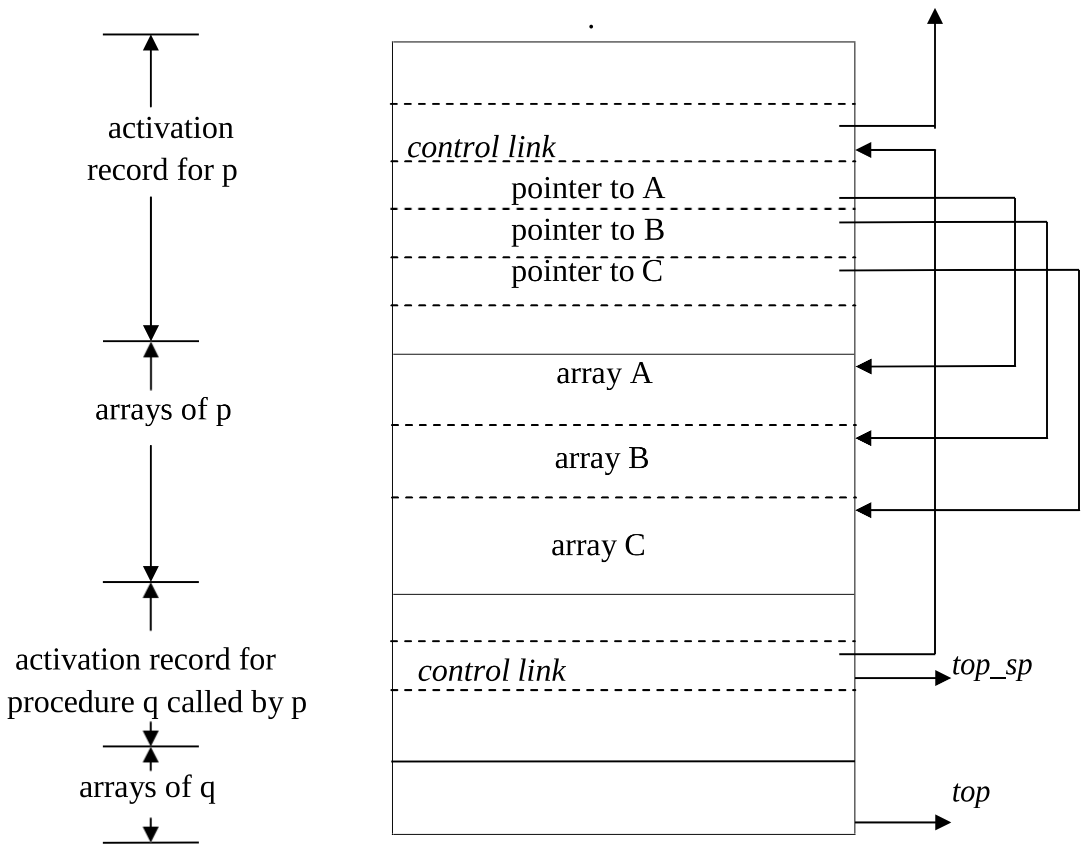
- The calling sequence and its division between caller and callee are as follows.

  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:

  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  - Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



Access to dynamically allocated arrays

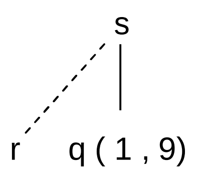
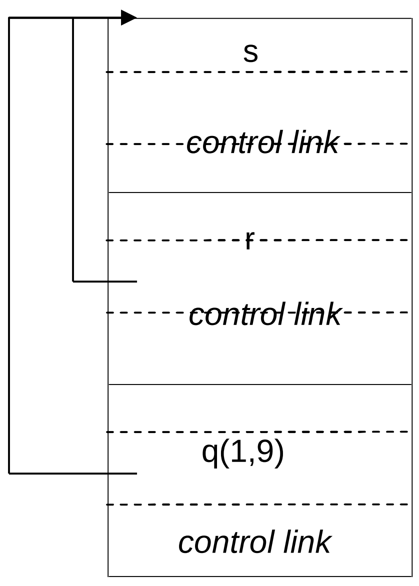
- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the	Activation records in the heap	Remarks
		Retained activation record for r

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

PARAMETERS PASSING

A language has first-class functionsif functions can bedeclared within any scope passed as arguments to other functions returned as results of functions.In a language with first-class functions and static scope, a function value is generally represented by a closure. a pair consisting of a pointer to function code a pointer to an activation record.Passing functions as arguments is very useful in structuring of systems using upcalls

Call-by-Value

The actual parameters are evaluated and their r-values are passed to the called procedure

A procedure called by value can affect its caller either through nonlocal names or through pointers.

Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.

Pascal uses pass by value by default, but var parameters are passed by reference.

Call-by-Reference

Also known as call-by-address or call-by-location. The caller passes to the called procedure the l-valueof the parameter.

If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.

Parameters in Fortran are passed by reference an old implementation bug in Fortran

```
func(a,b) { a = b};
call func(3,4); print(3);
```

Copy-Restore

A hybrid between call-by-value and call-by reference.

The actual parameters are evaluated and their r-values are passed as in call- by-value. In addition, l values are determined before the call.

When control returns, the current r-values of the formal parameters are copied back into the l-values of the actual parameters.

Call-by-Name

The actual parameters literally substituted for the formals. This is like a macro-expansion or in-line expansion Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used. In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.

SYMBOL TABLE

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

- A symbol table may serve the following purposes depending upon the language in hand:
- To store the names of all entities in a structured form at one place.•
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the• source code are semantically correct.
- To determine the scope of a name (scope resolution).

Implementation

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways: \

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Operations

A symbol table, either linear or hash, should provide the following operations.
insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example: int a; should be processed by the compiler as:

insert(a, int);

Lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Scope Management

A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . . int value=10;
 void pro_one()
{
 int one_1;
int one_2;

{

\ int one_3; |_ inner scope 1 int one_4; |

} / int one_5;

{

\ int one_6; |_ inner scope 2 int one_7; |

} / }

void pro_two()

{

int two_1; int two_2;

{ \ int two_3; |_ inner scope 3 int two_4; | }

/ int two_5; } . . .
```

The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e., current symbol table,
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or the global symbol table has been searched for the name.

UNIT V - CODE OPTIMIZATION AND CODE GENERATION

Topics to be Covered

Principal Sources of Optimization-DAG- Optimization of Basic Blocks-Global Data Flow Analysis-

Efficient Data Flow Algorithms-Issues in Design of a Code Generator - A Simple Code Generator

Algorithm.

INTRODUCTION

➢ The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

➢ Optimizations are classified into two categories. They are
  Machine independent optimizations:
  Machine dependant optimizations:

Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

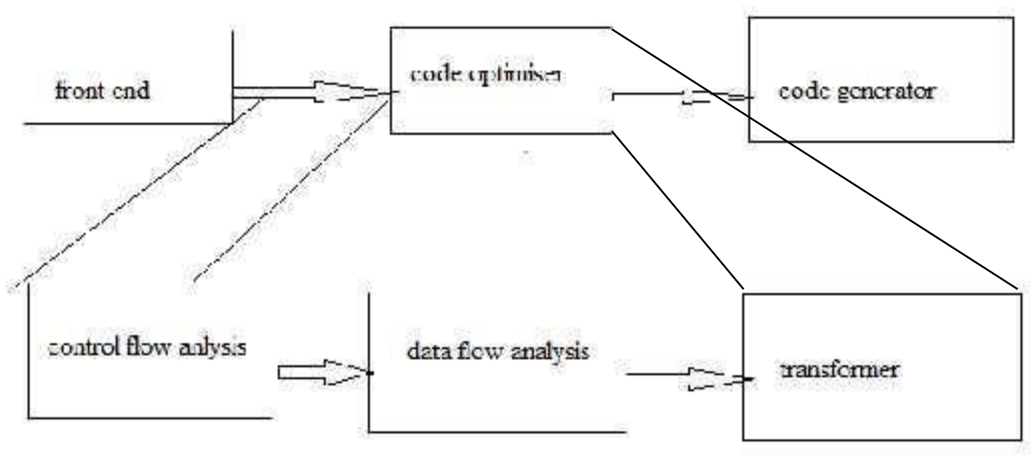
✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.

✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.

✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:



- ➢ Flow analysis is a fundamental prerequisite for many important types of code improvement.

  Generally control flow analysis precedes data flow analysis.

  Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as

  control flow graph

  Call graph

  Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPAL SOURCES OF OPTIMISATION

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.

Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

<u>Function-Preserving Transformations</u>

There are a number of ways in which a compiler can improve a program without changing the function it computes.

- The transformations

✓ Common sub expression elimination,
✓ Copy propagation,
✓ Dead-code elimination, and
✓ Constant folding

are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

➢ **Common Sub expressions elimination:**

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

t_1: = 4*i
t_2: = a [t1]
t_3: = 4*j
t_4: = 4*i
t_5: = n
t_6: = b [t_4] +t_5

The above code can be optimized using the common sub-expression elimination as

t_1: = 4*i
t_2: = a [t_1]
t_3: = 4*j
t_5: = n
t_6: = b [t_1] +t_5

The common sub expression t_4: =4*i is eliminated as its computation is already in t_1. And value of i is not been changed from definition to use.

➢ **Copy Propagation:**

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

```
x=Pi;
……
A=x*r*r;
```

The optimization using copy propagation can be done as follows:

```
A=Pi*r*r;
```

Here the variable x is eliminated

➢ **Dead-Code Eliminations:**

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute

values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.
Example:

```
i=0;
if(i=1)
{
 a=b+5;
}
```

Here, „if" statement is dead code because this condition will never get satisfied.

➢ **Constant folding**:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.

✓ For example,
 a=3.14157/2 can be replaced by
 a=1.570 there by eliminating a division operation.

➢ **Loop Optimizations:**

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:

✓ code motion, which moves code outside a loop;
✓ Induction-variable elimination, which we apply to replace variables from inner loop.
✓ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

➢ **Code Motion:**

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

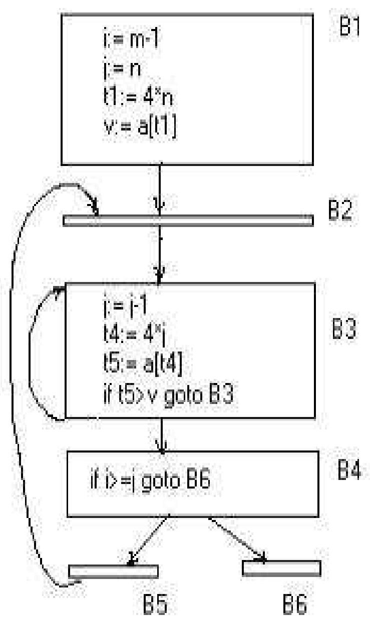
```
while (i <= limit-2)     /* statement does not change limit*/
```

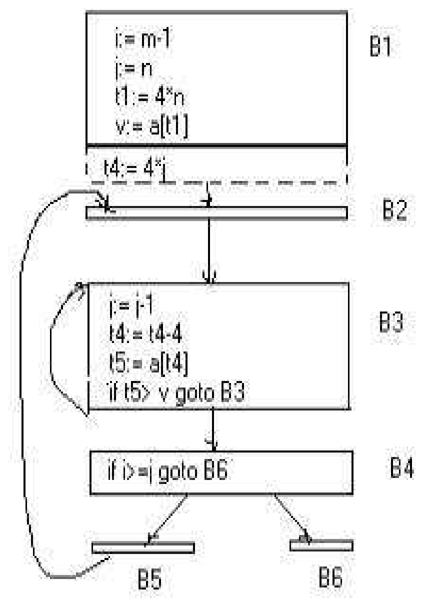
Code motion will result in the equivalent of

t= limit-2;
while (i<=t)   /* statement does not change limit or t */

➢ **Induction Variables :**

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.
  Example:
  As the relationship t_4:=4*j surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t_4:= 4*j-4 must hold. We may therefore replace the assignment t_4:= 4*j by t_4:= t_4-4. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t_4=4*j on entry to the block B3, we place an initializations of t_4 at the end of the block where j itself is





before   **after**

initialized, shown by the dashed addition to block B1 in second Fig.

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

➢ **Reduction In Strength:**

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

  For example, x is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
  1. Leaves are labeled by unique identifiers, either variable names or constants.
  2. Interior nodes are labeled by an operator symbol.
  3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

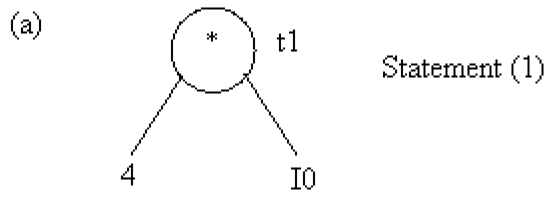
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

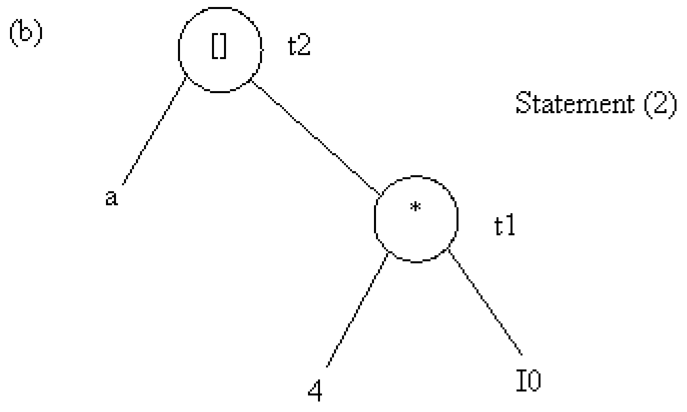
identifiers for the node n found in step 2 and set node(x) to n.

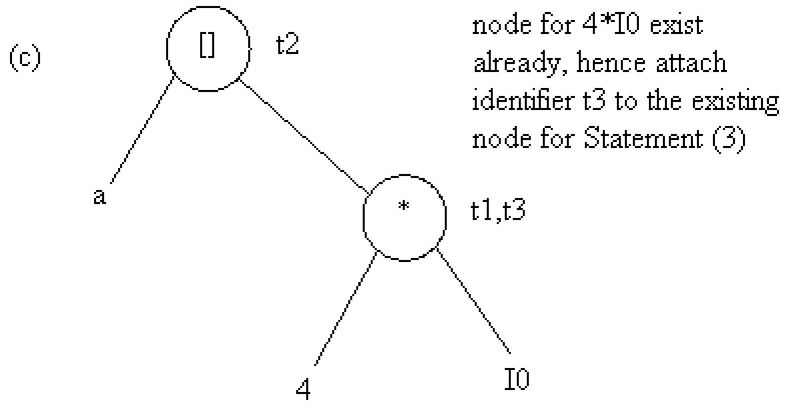
Example: Consider the block of three- address statements:

```
1.  t₁ := 4* i
2.  t₂ := a[t₁]
3.  t₃ := 4* i
4.  t₄ := b[t₃]
5.  t₅ := t₂*t₄
6.  t₆ := prod+t₅
7.  prod := t₆
8.  t₇ := i+1
9.  i := t₇
10. if i<=20 goto (1)
```

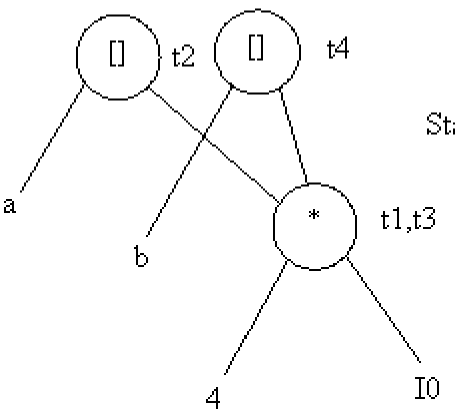
Stages in DAG Construction





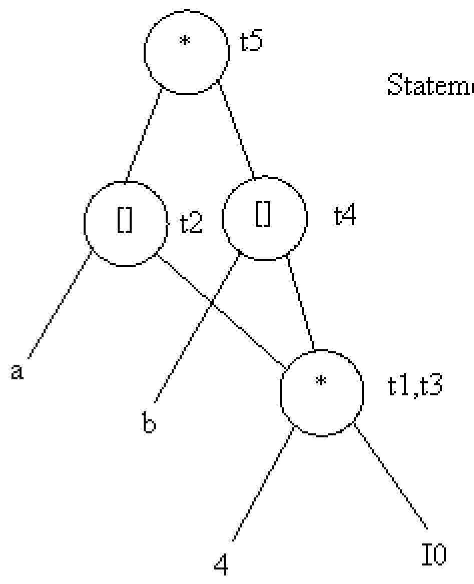

(c)
[]
t2
a
*
t1,t3
4
I0
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)



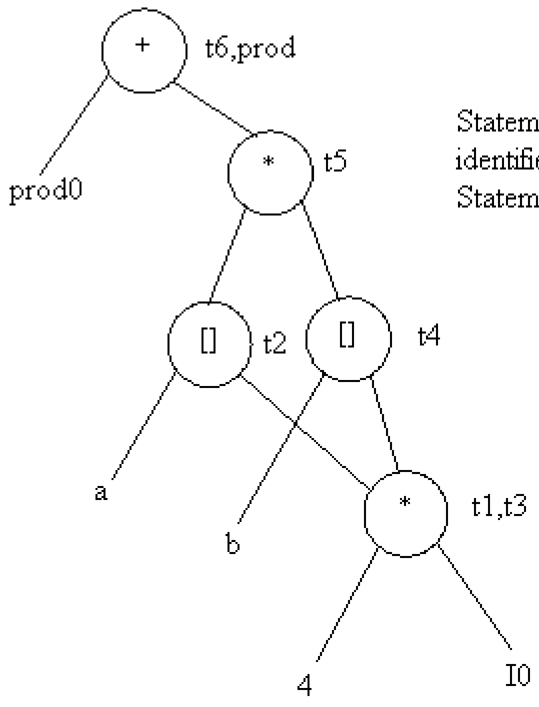
Statement (4)

(e)



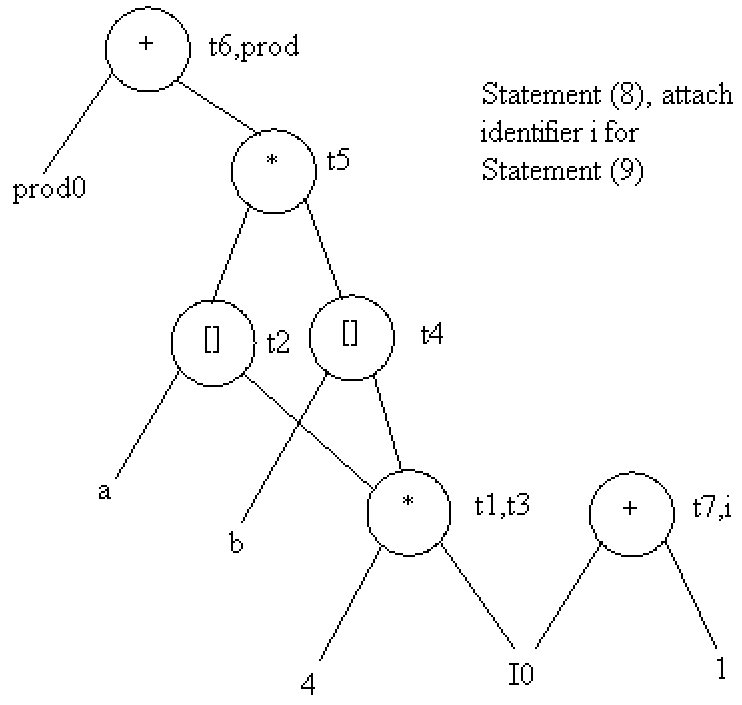
Statement (5)

(f)



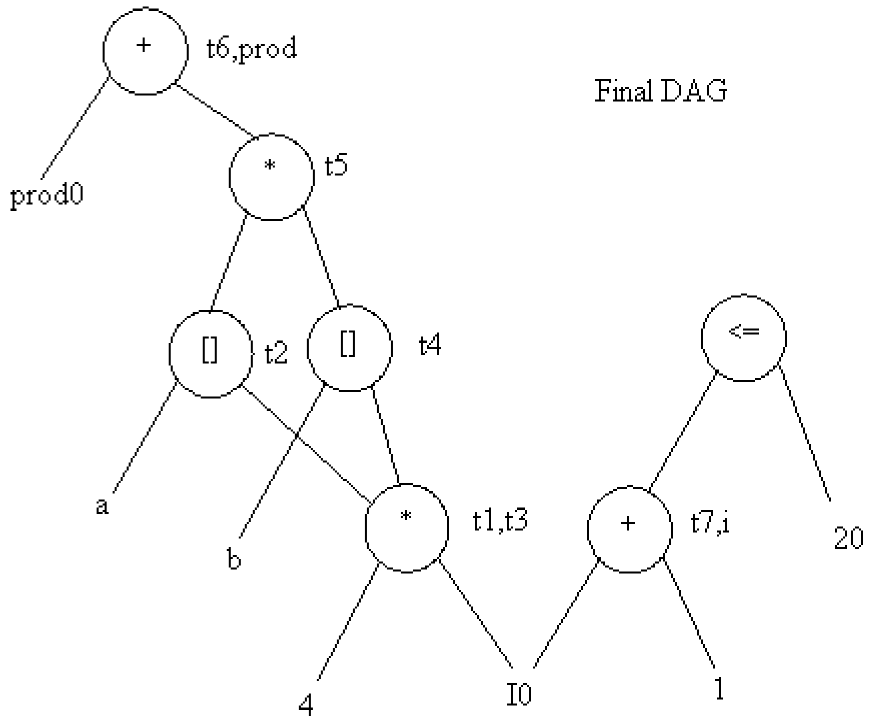
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

t_1 : = a + b
t_2 : = c + d
t_3 : = e – t_2
t_4 : = t_1 – t_3

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

t_2 : = c + d
t_3 : = e – t_2
t_1 : = a + b
t_4 : = t_1 – t_3

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions **MOV R_0 , t_1** and **MOV t_1 , R_1** have been saved.

A Heuristic ordering for Dags

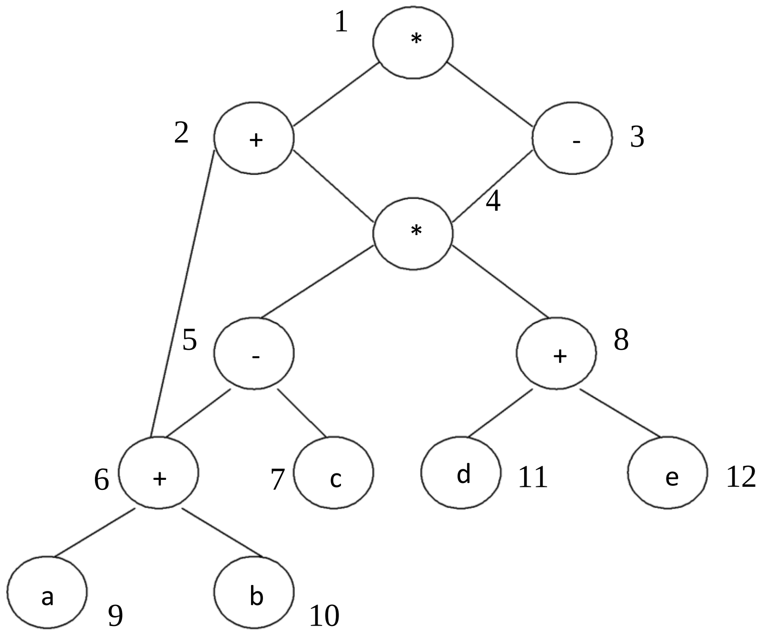
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

Algorithm:

```
1)  while unlisted interior nodes remain do begin
2)      select an unlisted node n, all of whose parents have been listed;
3)      list n;
4)      while the leftmost child m of n has no unlisted parents and is not a leaf do
            begin
5)              list m;
6)              n : = m
            end
        end
```

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

t_8 : = d + e t_6 : = a
+ b t_5 : = t_6 – c t_4
: = t_5 * t_8 t_3 : = t4
– e t_2 : = t_6 + t_4 t_1
: = t_2 * t_3

This will yield an optimal code for the DAG on machine whatever be the number of registers.

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- ✓ Structure-Preserving Transformations
- ✓ Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- ✓ Common sub-expression elimination
- ✓ Dead code elimination
- ✓ Renaming of temporary variables
- ✓ Interchange of two independent adjacent statements.

➢ **Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it‟s referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

```
a:  =b+c
b:  =a-d
c:  =b+c
d: =a-d
```

The 2nd and 4th statements compute the same expression: b+c and a-d
Basic block can be transformed to

```
a: = b+c
b: = a-d
c: = a
d: = b
```

➢ **Dead code elimination:**

It"s possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of constructio n or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

➢ **Renaming of temporary variables:**

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
- In this we can transform a basic block to its equivalent block called normal-form block.

➢ **Interchange of two independent adjacent statements:**

- Two statements

  t_1:=b+c
  t_2:=x+y

  can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

  a :=b+c
  e :=c+d+b

  the following intermediate code may be generated:

  a :=b+c
  t :=c+d
  e :=t+b

- Example:

  x:=x+0 can be removed

  x:=y**2 can be replaced by a cheaper statement x:=y*y

- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

LOOPS IN FLOW GRAPH

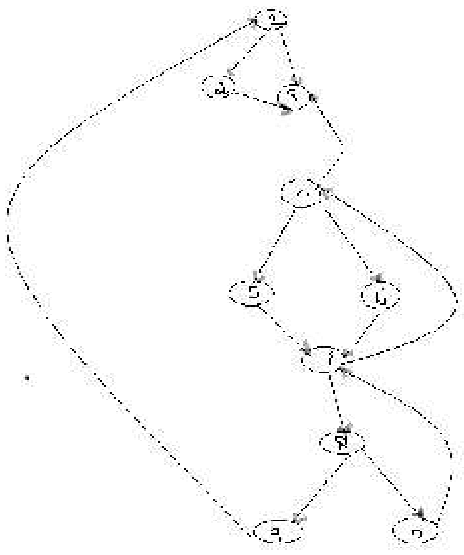
A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

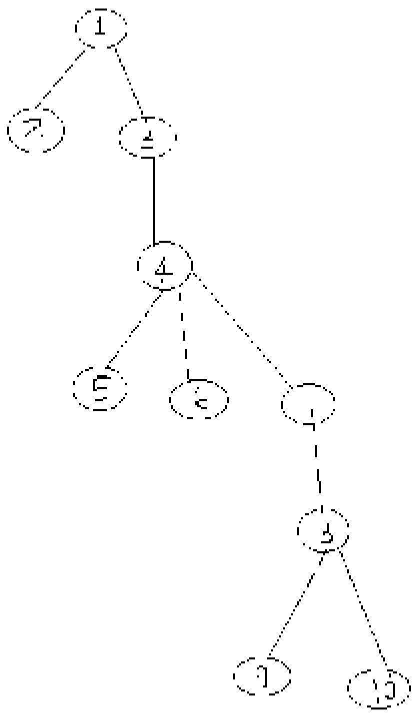
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

*In the flow graph below,
*Initial node,node1 dominates every node.
*node 2 dominates itself
*node 3 dominates all but 1 and 2.
*node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10.
*node 8 dominates 8,9 and 10.
*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is theroot.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n.
- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.



D(1)={1} D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

D(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.

- The properties of loops are

  ✓ A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
  ✓ There must be at least one way to iterate the loop(i.e.)at least one path back to the header.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

  ✓ Example:

  In the above graph,

7 → 4	4 DOM 7
10 →7	7 DOM 10
4 → 3	
8 → 3	
9 →1	

- The above edges will form loop in flow graph.
- Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.

Input: A flow graph G and a back edge n→d.

Output: The set loop consisting of all nodes in the natural loop n→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m‟s predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

```
Procedure insert(m);
if m is not in loop then begin
        loop := loop U {m};
        push m onto stack
end;

stack : = empty;
```

```
loop : = {d};
insert(n);
while stack is not empty do begin
        pop m, the first element of stack, off stack;
        for each predecessor p of m do insert(p)
end
```

Inner loop:

- If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

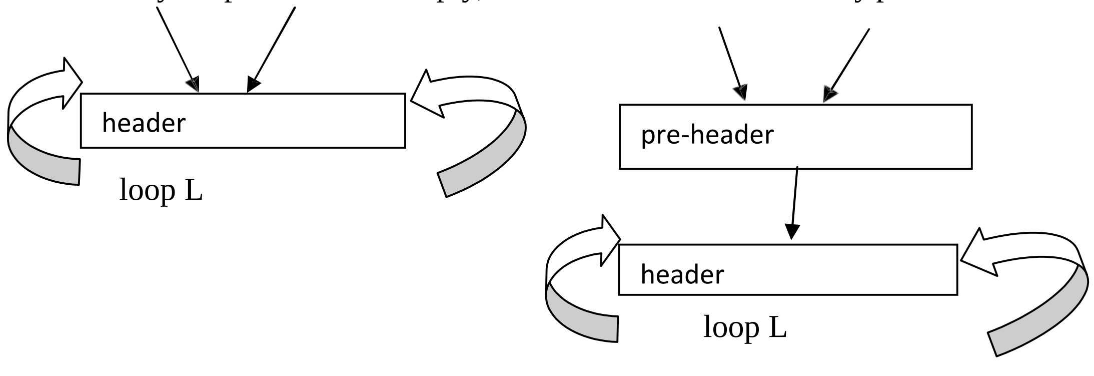
Pre-Headers:

- Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater.

- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.

- Edges from inside loop L to the header are not changed.

- Initially the pre-header is empty, but transformations on L may place statements in it.



(a) Before **(b) After**

Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.

- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

- ***Definition***:
  A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.

✓ The forward edges from an acyclic graph in which every node can be reached from initial node of G.

✓ The back edges consist only of edges where heads dominate theirs tails.

✓ Example: The above flow graph is reducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.

- The remaining edges are forward edges.

- If the forward edges form an acyclic graph, then we can say the flow graph reducible.

- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.

- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:

  ✓ Redundant-instructions elimination
  ✓ Flow-of-control optimizations
  ✓ Algebraic simplifications
  ✓ Use of machine idioms
  ✓ Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R_0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
 #define debug 0
 ….
If ( debug  ) {
Print debugging information
}
```

- In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2
goto L2
L1: print debugging information
L2: …………………………(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2
Print debugging information
L2: ……………………………(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

```
If debug ≠0 goto L2
Print debugging information
L2: ……………………………(c)
```

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or th e target code by the following types of peephole optimizations. We can replace the jump sequence

```
goto L1
….
L1: gotoL2 by the sequence
goto L2
….
L1: goto L2
```

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

```
if a < b goto L1
….
L1: goto L2 can be replaced by
If a < b  goto  L2
….
L1: goto L2
```

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto.

```
Then the sequence
goto L1
……..
L1: if a < b goto L2
L3:     …………………………………..(1)
```

- May be replaced by

```
If a < b goto L2 goto L3
…….
L3:      ………………………………….(2)
```

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

```
x := x+0 Or
x := x * 1
```

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

- For example, x is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like
  i :=i+1.
  i:=i+1 → i++ i:=i-1 →
  i- -

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.

- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

- Data-flow information can be collected by setting up and solving systems of equations of the form :

$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."

- The details of how data-flow equations are set and solved depend on three factors.

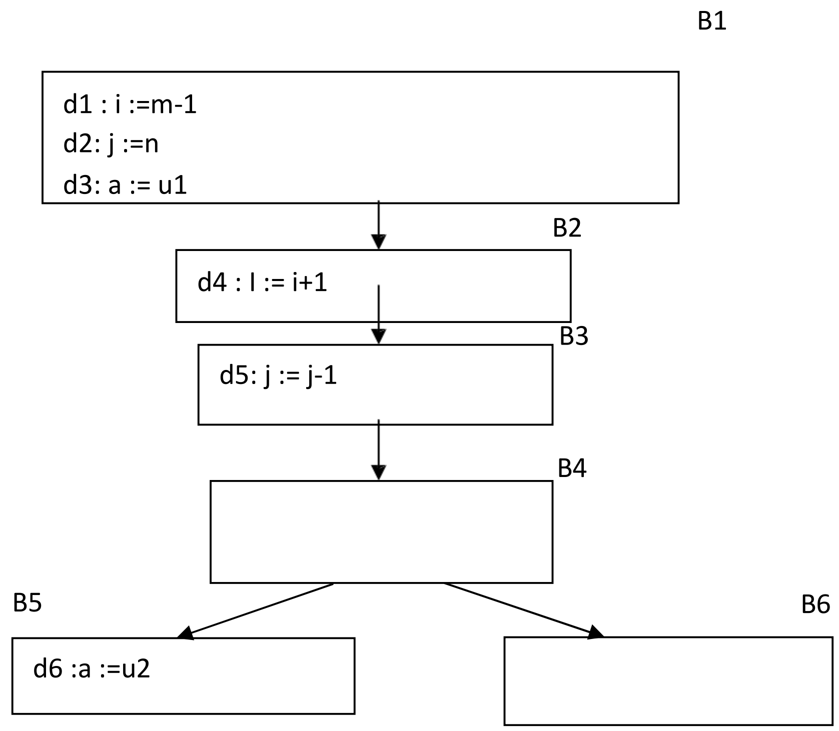
✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path from p_1 to p_n is a sequence of points p_1, p_2,….,p_n such that for each i between 1 and n-1, either

✓ P_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or

✓ P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

- A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.

- These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:

✓ A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.

✓ An assignment through a pointer that could refer to x. For example, the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not “killed” along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

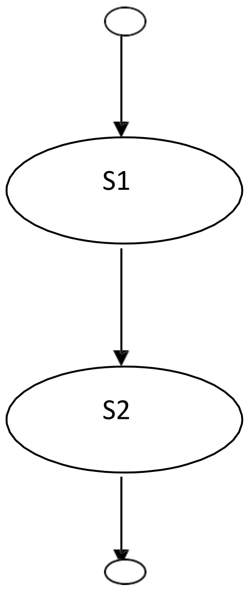
Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

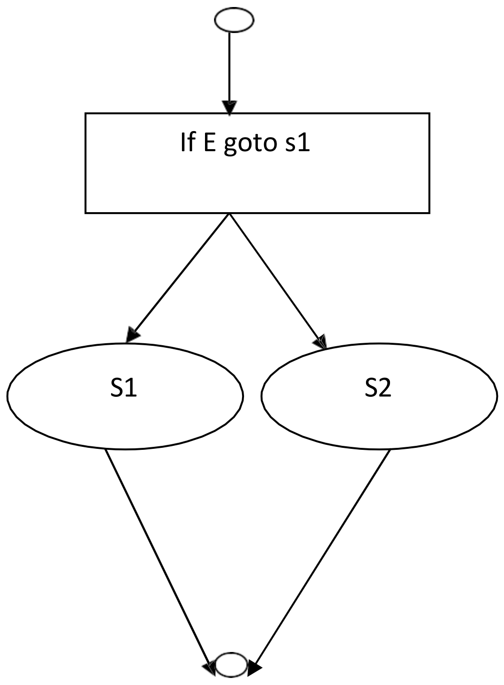
  S $\longrightarrow$ id: = E| S; S | if E then S else S | do S while E

  E $\longrightarrow$ id + id| id

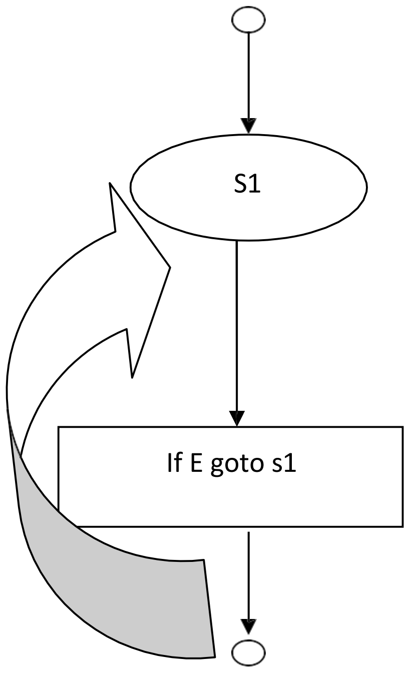
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



S1 ; S2



IF E then S1 else S2

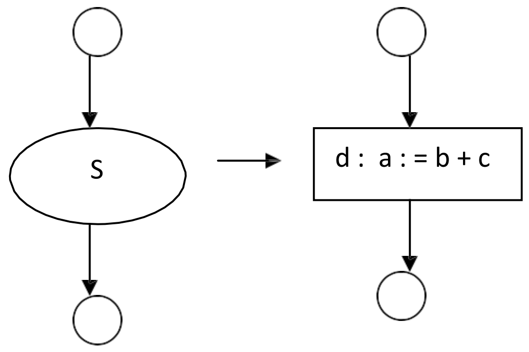


do S1 while E

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

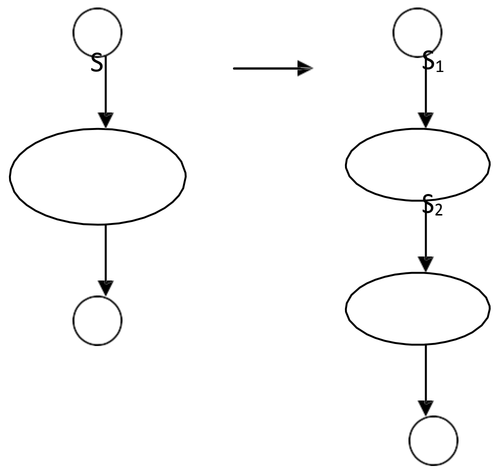
- We say that the beginning points of the dummy blocks at the entry and exit of a statement‟s region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i)



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus
  Gen[S]={d}
- On the other hand, d "kills" all other definitions of a, so we write
  Kill[S] = D_a – {d}
  Where, D_a is the set of all definitions in the program for variable a. ii)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])
Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

in [S_1] = in [S]
in [S_2] = out [S_1]
out [S] = out [S_2]

- Under what circumstances is definition d generated by $S=S_1$; S_2? First of all, if it is generated by S_2, then it is surely generated by S. if d is generated by S_1, it will reach the end of S provided it is not killed by S_2. Thus, we write
  gen[S]=gen[S_2] U (gen[S_1]-kill[S_2]
- Similar reasoning applies to the killing of a definition, so we have
  Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way.

- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.

- When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.

- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.

- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.

- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in.

S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.
- Assuming we know in[S] we compute out by equation, that

  is Out[S] = gen[S] U (in[S] - kill[S])
- Considering cascade of two statements S_1; S_2, as in the second case. We start by observing in[S_1]=in[S]. Then, we recursively compute out[S_1], which gives us in[S_2], since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1. Now we can compute out[S_2], and this set is equal to out[S].
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.

  In[S_1] = in[S_2] = in[S]
- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

  Out[S]=out[S_1] U out[S_2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A

.

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.

- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.

- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.

- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.

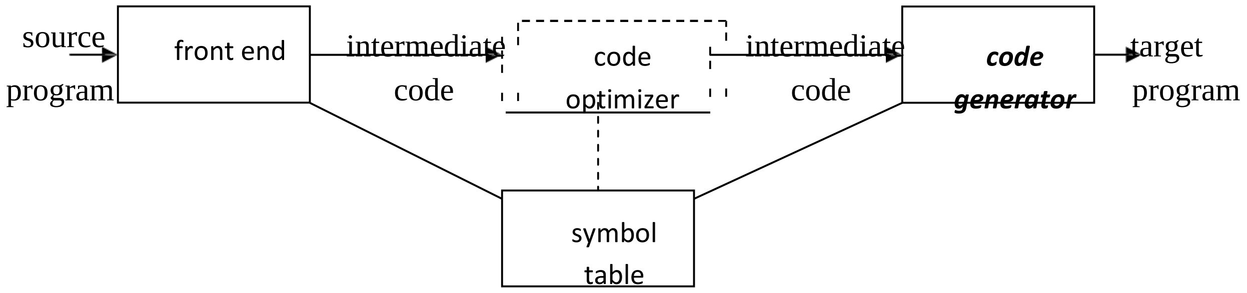
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval -based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representation can be :
  a. Linear representation such as postfix notation
  b. Three address representation such as quadruples
  c. Virtual machine representation such as stack machine code
  d. Graphical representations such as syntax trees and dags.
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :
  a. Absolute machine language
     - It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language
- It allows subprograms to be compiled separately.

c. Assembly language
- Code generation is made easier.

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions. For example,

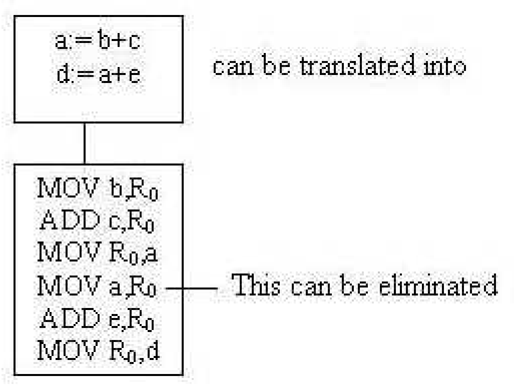
  j : **goto** *i* generates jump instruction as follows :

  ➢ if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.

  ➢ if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



```
a:=b+c
d:=a+e
```

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :

  ➢ ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.

- ➢ ***Register assignment*** – the specific register that a variable will reside in is picked

- Certain machine requires even-odd *register pairs* for some operands and results. For example , consider the division instruction of the form :

  D x, y

  where, x – dividend even register in even/odd register pair
  y – divisor
  even register holds the remainder
  odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n* general-purpose registers, R_0, R_1, . . . , R_{n-1}.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :

  MOV (move *source* to *destination*)
  ADD (add *source* to *destination*)
  SUB (subtract *source* from *destination*)

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	**FORM**	***ADDRESS***	**ADDED COST**
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c*+*contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ; MOV 4(R_0), M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
  For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

  i) MOV b, R_0
  ADD c, R_0 cost = 6
  MOV R_0, a

  ii) MOV b, a
  ADD c, a cost = 6

  iii) Assuming R_0, R_1 and R_2 contain the addresses of a, b, and c :
  MOV *R_1, *R_0
  ADD *R_2, *R_0 cost = 2

- In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectivelyuses registers to store operands of the statements.

- For example: consider the three-address statement **a := b+c**
  It can have the following sequence of codes:

  ADD R_j, R_i Cost = 1 // if R_i contains b and R_j contains c

  (or)

  ADD c, R_i Cost = 2 // if c is in a memory location

  (or)

  MOV c, R_j Cost = 3 // move c from memory to Rj and add

  ADD R_j, R_i

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

2. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

3. Consult the address descriptor for y to determine y‟‟, the current location of y. Prefer the register for y‟‟ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.

4. Generate the instruction **OP z' , L** where z‟‟ is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

5. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

  t : = a – b
  u : = a – c
  v : = t + u
  d : = v + u

with d live at the end.
Code sequence for the example is:

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements
a : = b [i] and **a [i] : = b**

Statements	**Code Generated**	**Cost**
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments
a : = *p and ***p : = a**

Statements	**Code Generated**	**Cost**
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	**Code**
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z if x < 0 goto z	MOV y, R_0 ADD z, R_0 MOV R_0,x CJ< z