PRATHYUSHA

ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION 2021 – IV SEMESTER

CS3492 – DATABASE MANAGEMENT SYSTEMS

CS3492 DATABASE MANAGEMENT SYSTEMS

COURSE OBJECTIVES:

- To learn the fundamentals of data models, relational algebra and SQL
- To represent a database system using ER diagrams and to learn normalization techniques
- To understand the fundamental concepts of transaction, concurrency and recovery processing
- To understand the internal storage structures using different file and indexing techniques which will help in physical DB design
- To have an introductory knowledge about the Distributed databases, NOSQL and database security

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

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 77 Preservation – Boyce/Codd Normal Form – Multi-valued Dependencies and Fourth Normal Form – Join Dependencies and Fifth Normal Form

UNIT III TRANSACTIONS

Transaction Concepts – ACID Properties – Schedules – Serializability – Transaction support in SQL – Need for Concurrency – Concurrency control –Two Phase Locking- Timestamp – Multiversion – Validation and Snapshot isolation– Multiple Granularity locking – Deadlock Handling – Recovery Concepts – Recovery based on deferred and immediate update – Shadow paging – ARIES Algorithm

UNIT IV IMPLEMENTATION TECHNIQUES

RAID – File Organization – Organization of Records in Files – Data dictionary Storage – Column Oriented Storage– Indexing and Hashing –Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for Selection, Sorting and join operations – Query optimization using Heuristics - Cost Estimation.

UNIT V ADVANCED TOPICS

Distributed Databases: Architecture, Data Storage, Transaction Processing, Query processing and optimization – NOSQL Databases: Introduction – CAP Theorem – Document Based systems – Key value Stores – Column Based Systems – Graph Databases. Database Security: Security issues – Access control based on privileges – Role Based access control – SQL Injection – Statistical Database security – Flow control – Encryption and Public Key infrastructures – Challenges

COURSE OUTCOMES:

Upon completion of this course, the students will be able to
CO1:Construct SQL Queries using relational algebra
CO2:Design database using ER model and normalize the database
CO3: Construct queries to handle transaction processing and maintain consistency of the database
CO4: Compare and contrast various indexing strategies and apply the knowledge to tune the performance of the database
CO5: Appraise how advanced databases differ from Relational Databases and find a suitable database for the given requirement.

TEXT BOOKS:

1. Abraham Silberschatz, Henry F. Korth, S. Sudharshan, "Database System Concepts", Seventh Edition, McGraw Hill, 2020.
2. Ramez Elmasri, Shamkant B. Navathe, "Fundamentals of Database Systems", Seventh Edition, Pearson Education, 2017

REFERENCES:

1. C.J.Date, A.Kannan, S.Swamynathan, "An Introduction to Database Systems", Eighth Edition, Pearson Education, 2006.

CS3492 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.

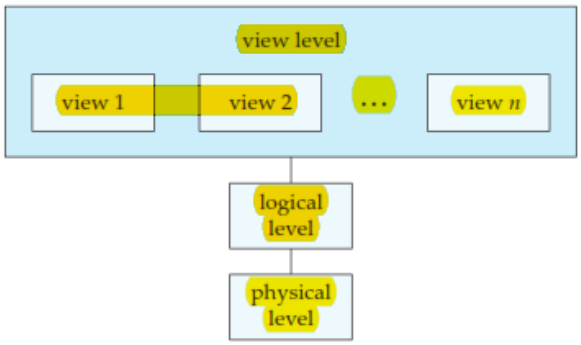


Figure . The three levels of data abstraction.

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.

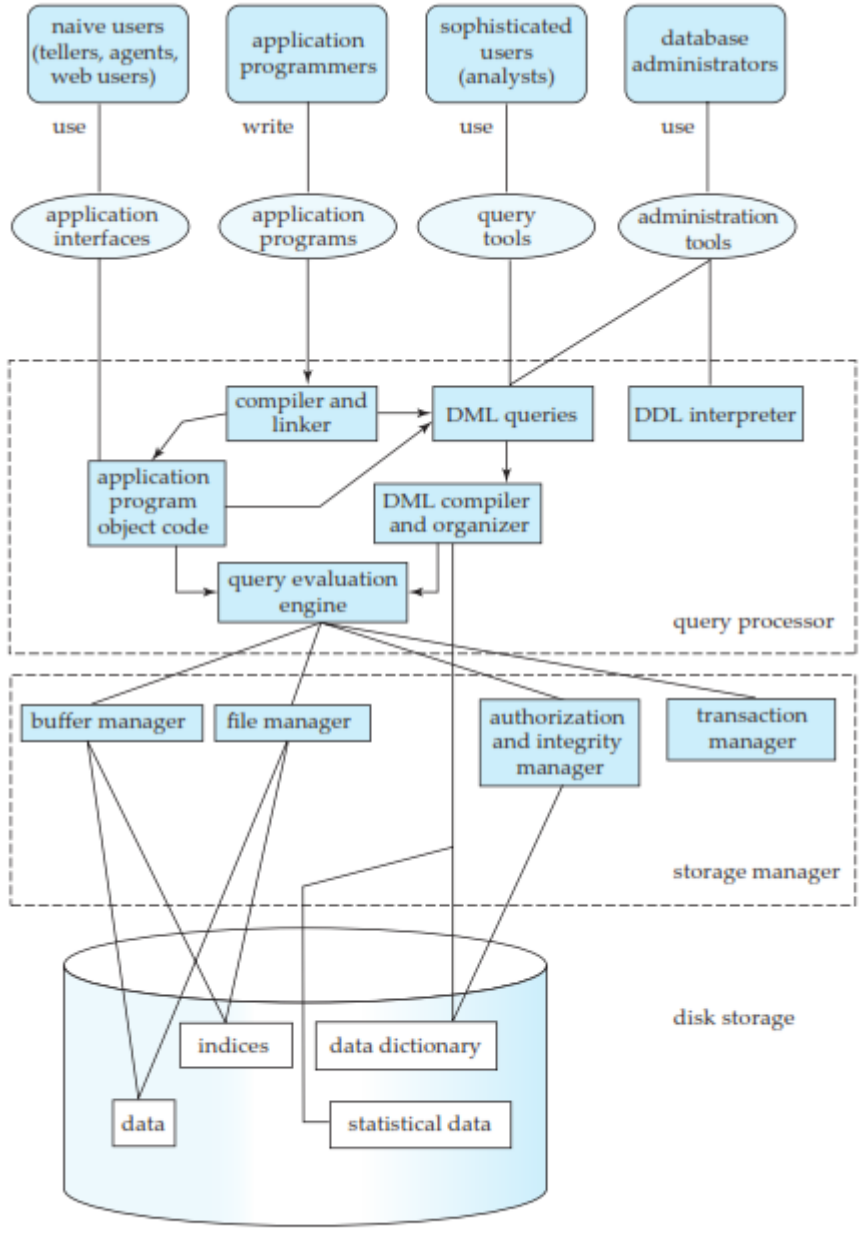


Figure 1.5 System structure.

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:

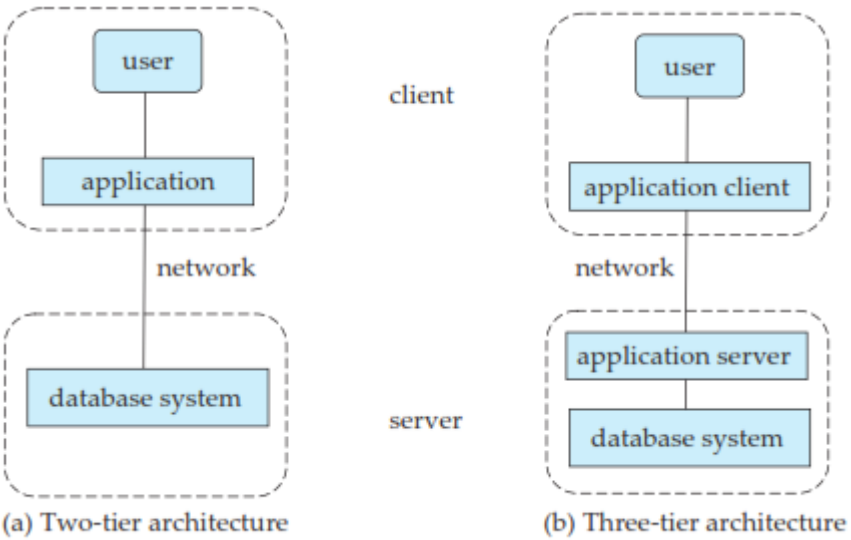


Figure . Two-tier and three-tier architectures.

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 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.

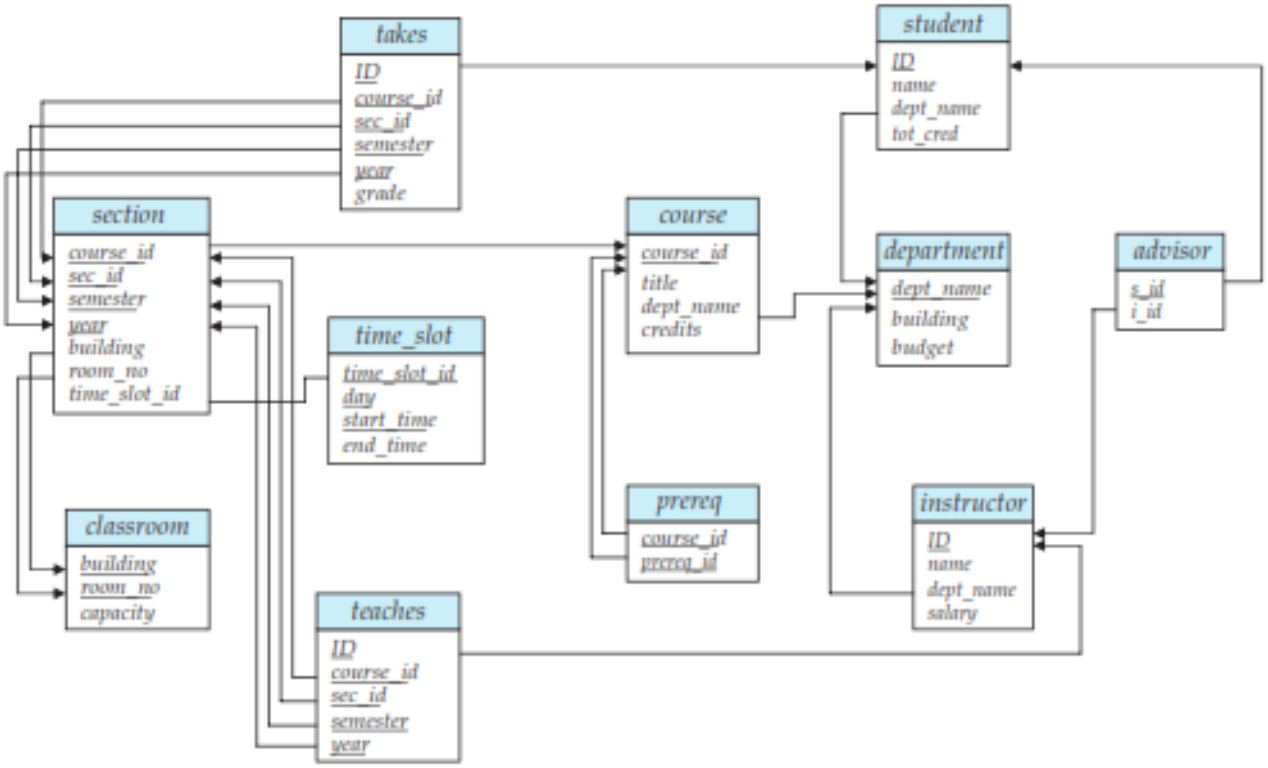


Figure. Schema diagram for the university database.

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. $\sigma_{BRANCH\_NAME="perryride"}$ (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. $\prod_{NAME, CITY}$ (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. $\prod_{\text{CUSTOMER\_NAME}}$ (BORROW) ∩ $\prod_{\text{CUSTOMER\_NAME}}$ (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. $\prod_{\text{CUSTOMER\_NAME}}$ (BORROW) - $\prod_{\text{CUSTOMER\_NAME}}$ (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)

8. 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

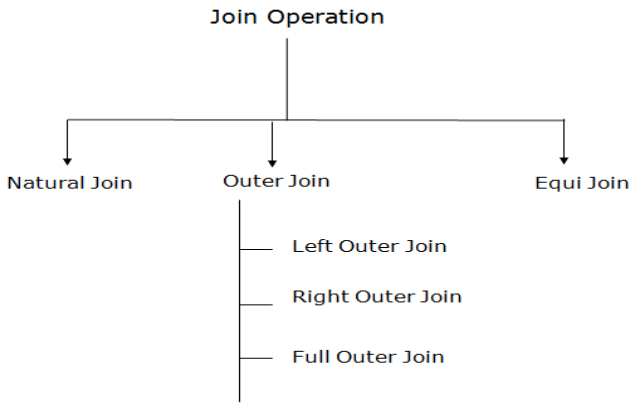
EMP_CODE	SALARY
101	50000
102	30000
103	25000

1. Operation: (EMPLOYEE ⋈ SALARY)

Result:

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

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. $\prod_{EMP\_NAME, SALARY}$ (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

9. 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

10. 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);

Subqueries in the From Clause:

- 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

With Clause:

- 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;

Complex Queries using With Clause:

- 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;

Scalar Subquery:

- 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

Modification of the Database:

- 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.

  - **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**

11. 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 $(A_1, A_1, \ldots, A_n)$

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

12. 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;
```

13. 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, …, 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

BUILT-IN DATA TYPES IN SQL

- **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

INDEX CREATION

- **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.

14. 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.

15. 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
```

Program 3:

```
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

Language Constructs for Procedures & Functions:

- 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

16. 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;
/
```

17. 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*

18. <u>Discuss about dynamic SQL.</u>

- 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 ;.
```

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.

Embedded SQL is essentially identical across different host languages. Programming conventions and syntax change very little. Therefore, to write applications in different languages, you need not learn new syntax.

Embedded SQL is easy to use because it is simply Transact- SQL with some added features that facilitate using it in an application.

```
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 ;
```

CS3492 DATABASE MANAGEMENT SYSTEMS

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.

---

1. Discuss about main phases of database design.

The main phases of database design are

- Requirements collection and analysis
- Conceptual design
- Logical design
- Physical 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.

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.

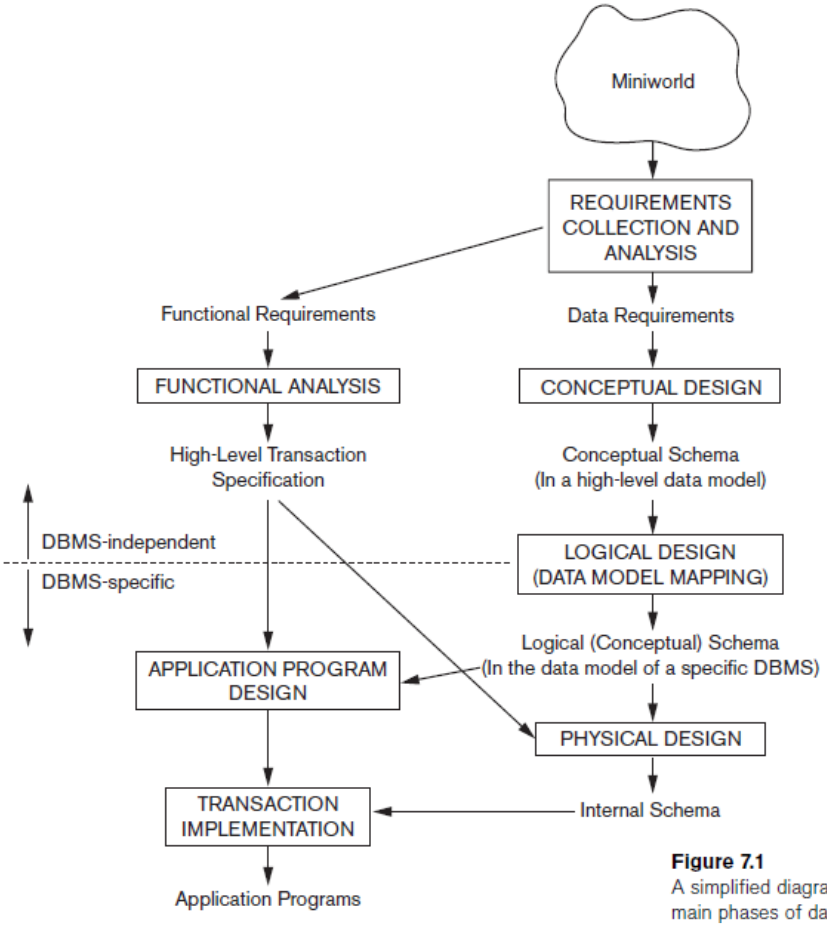


Figure 7.1
A simplified diagram to illustrate the main phases of database design.

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.

2. Explain symbols used in Entity-Relationship (ER) model.

- **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**.

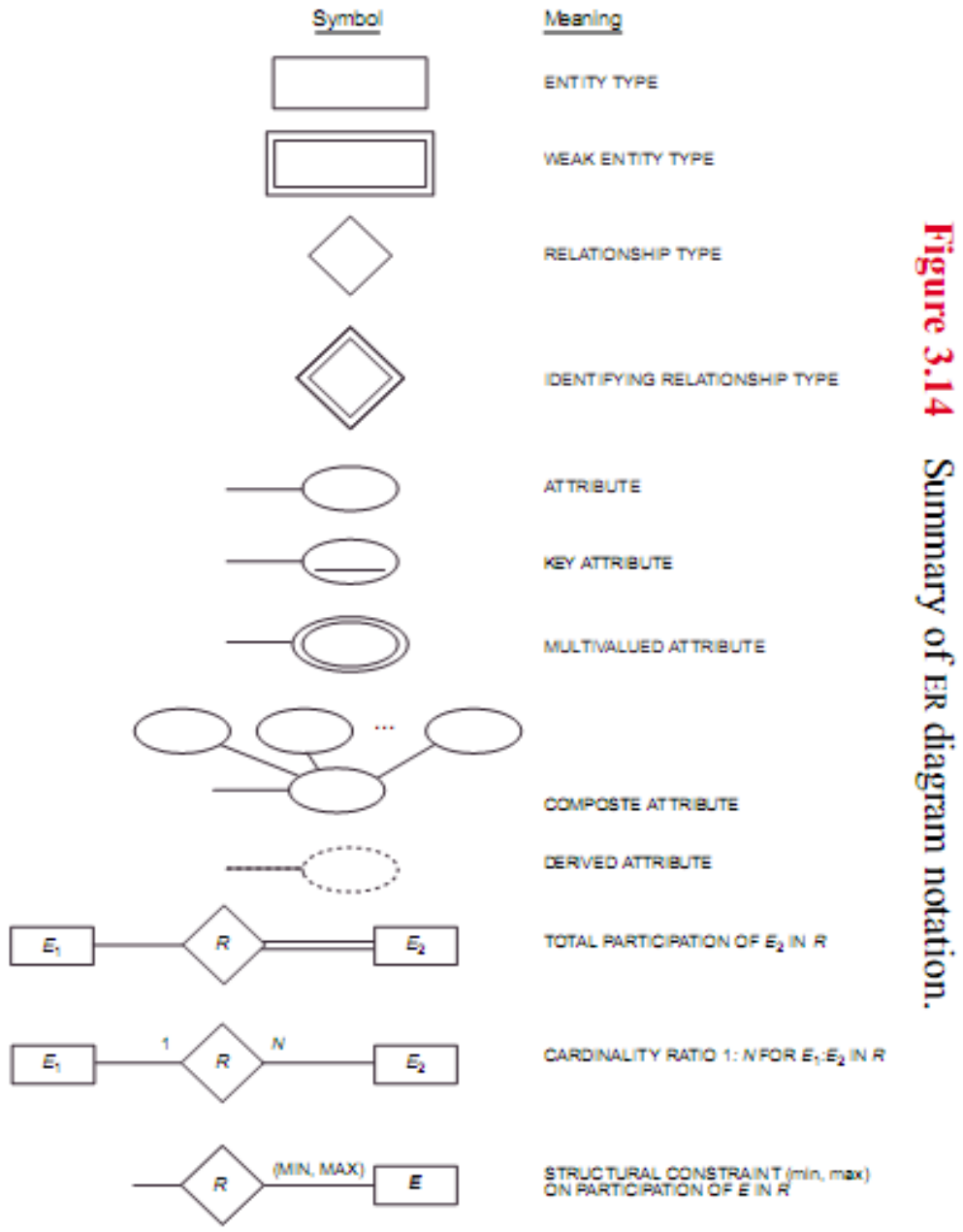


Figure 3.14 Summary of ER diagram notation.

- 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.

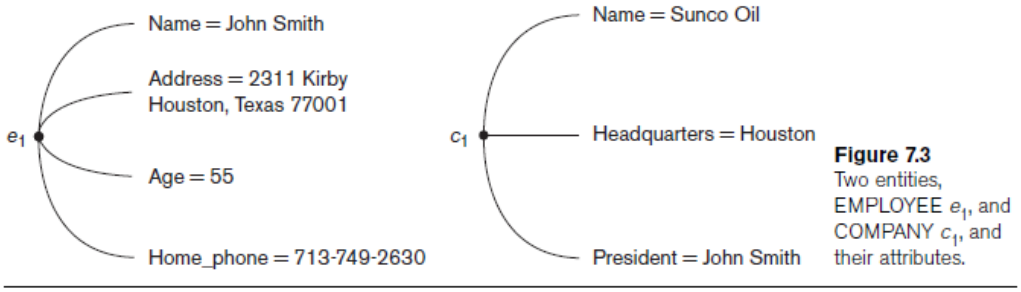


Figure 7.3
Two entities, EMPLOYEE e_1, and COMPANY c_1, and their attributes.

- 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.

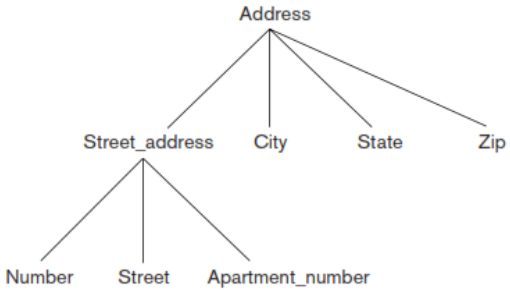


Figure .A hierarchy of composite attributes.

- 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. 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. For such situations, a special value called NULL is created.

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.
- 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.

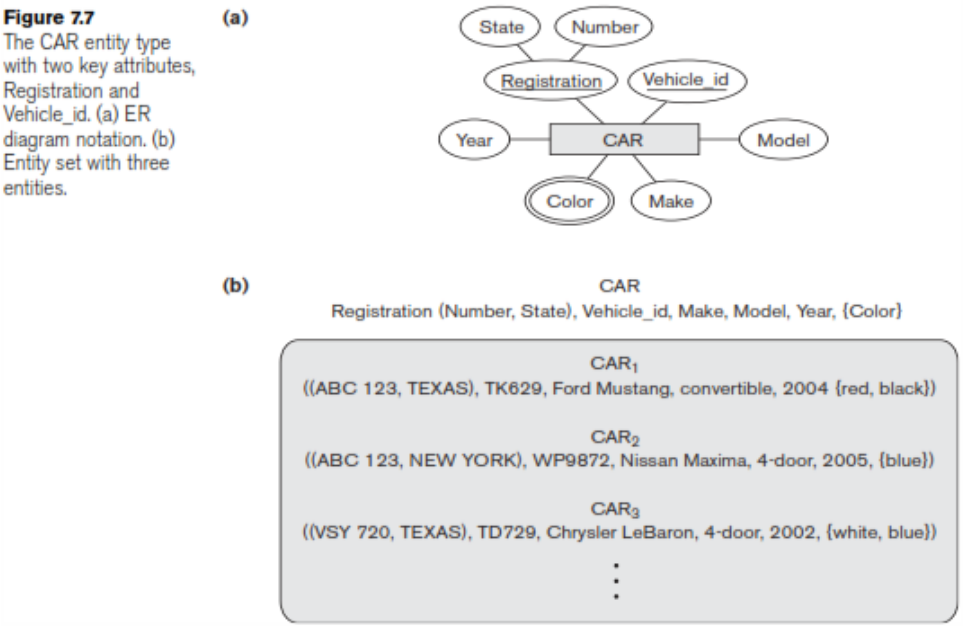


Figure 7.7
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.

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.
- In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval.

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.
- 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.

Relationship types, Roles, Structural Constraints:

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

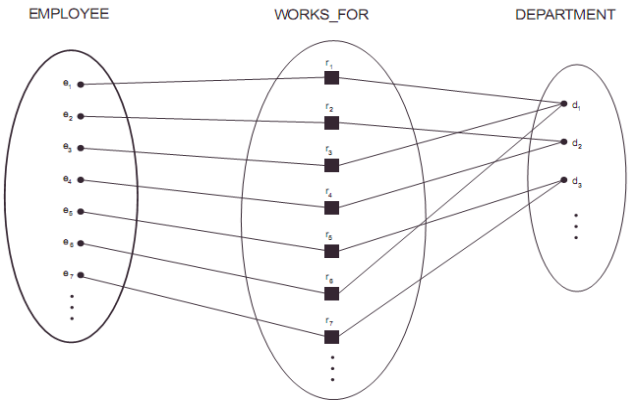


Figure. Some instances of the WORKS_FOR relationship between EMPLOYEE and DEPARTMENT.

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)

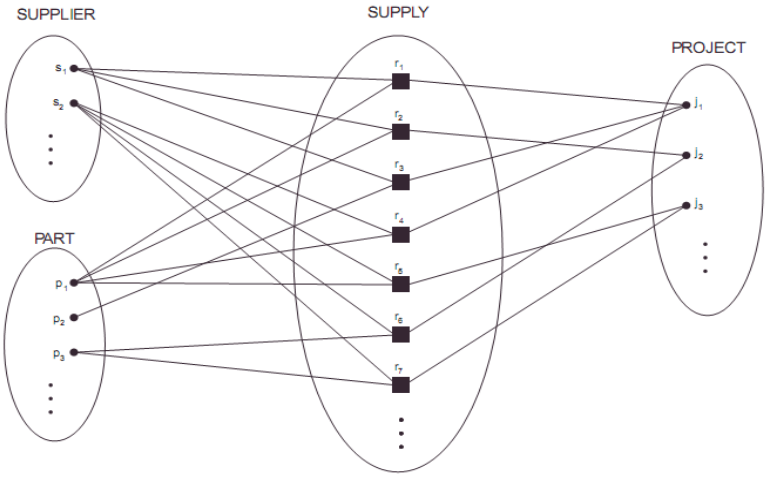


Figure. Some relationship instances of a ternary relationship SUPPLY.

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.

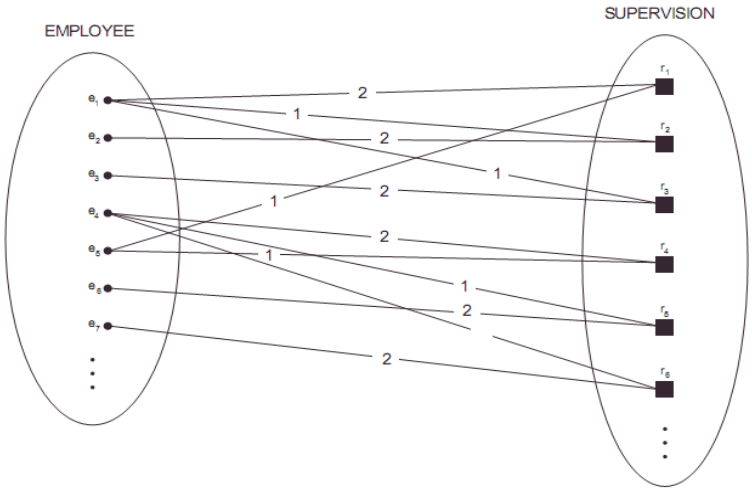


Figure. The recursive relationship SUPERVISION, where the EMPLOYEE entity type plays the two roles of supervisor (1) and supervisee (2).

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

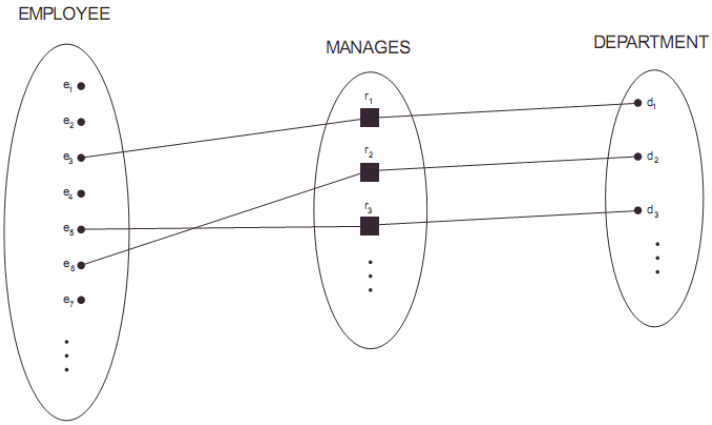


Figure. The 1:1 relationship MANAGES, with partial participation of employee and total participation of DEPARTMENT.

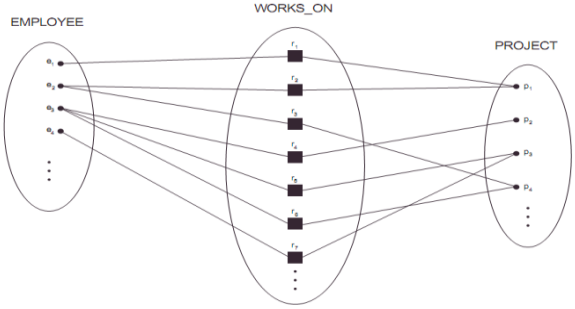


Figure. The M:N relationship WORKS_ON between EMPLOYEE and PROJECT.

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

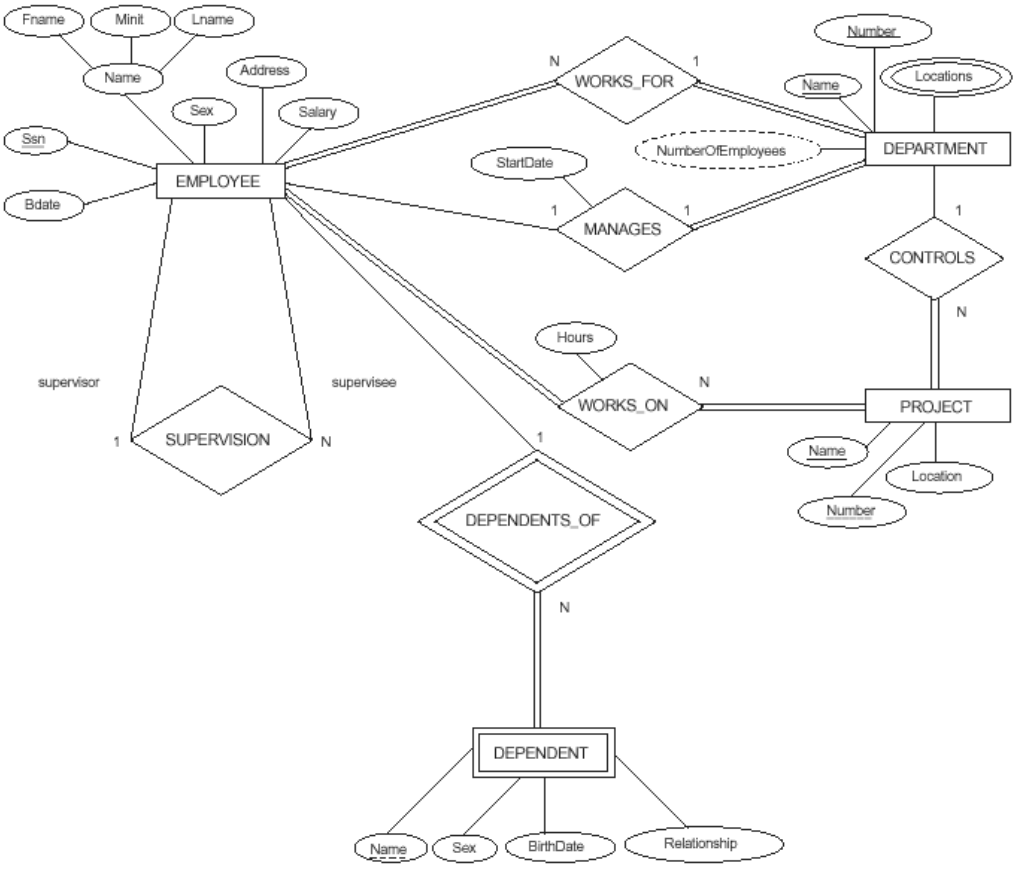
Example:

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

Figure. ER diagram of company.



3.Explain about Enhanced ER model. (or)

Discuss about 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

Specialization and Generalization:

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

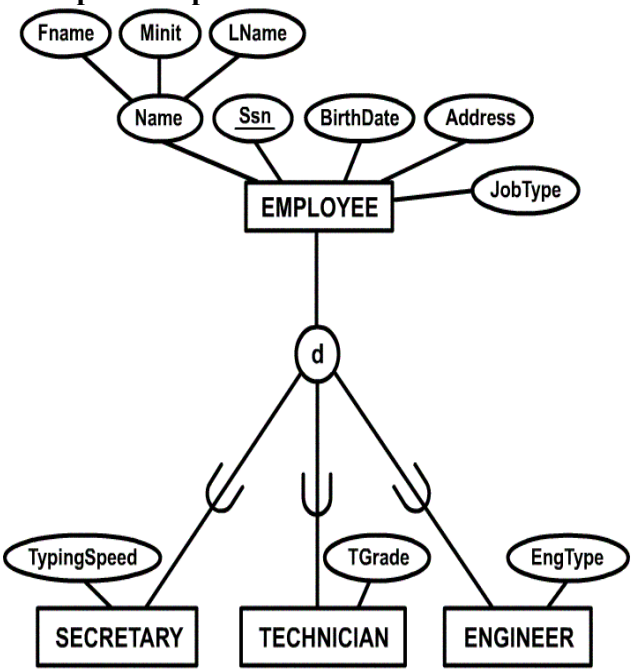
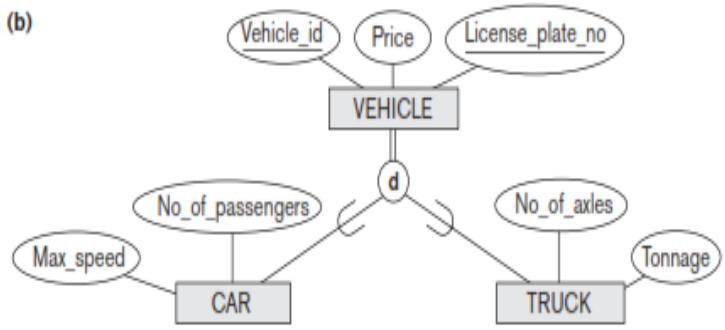


Figure.Specialization

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



<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)

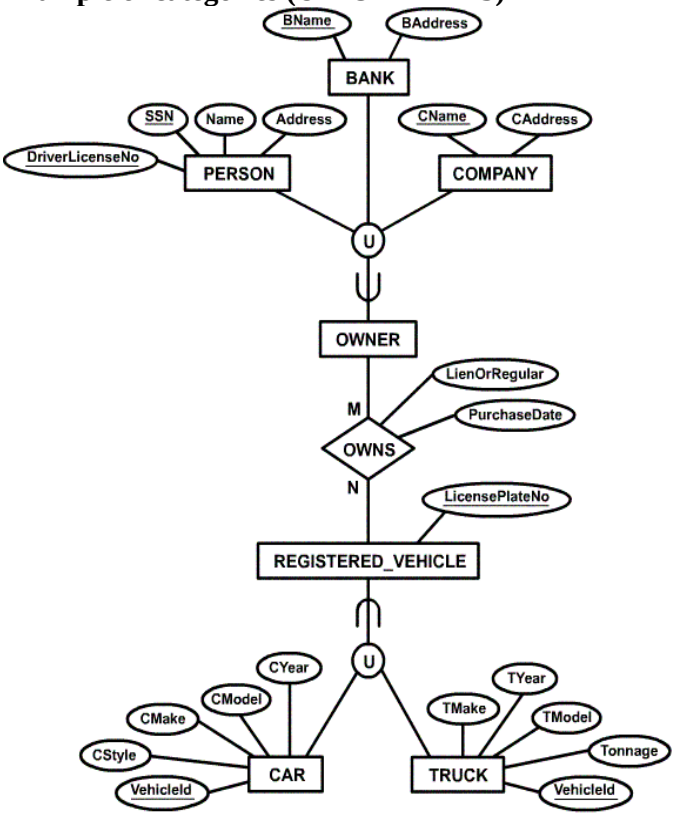
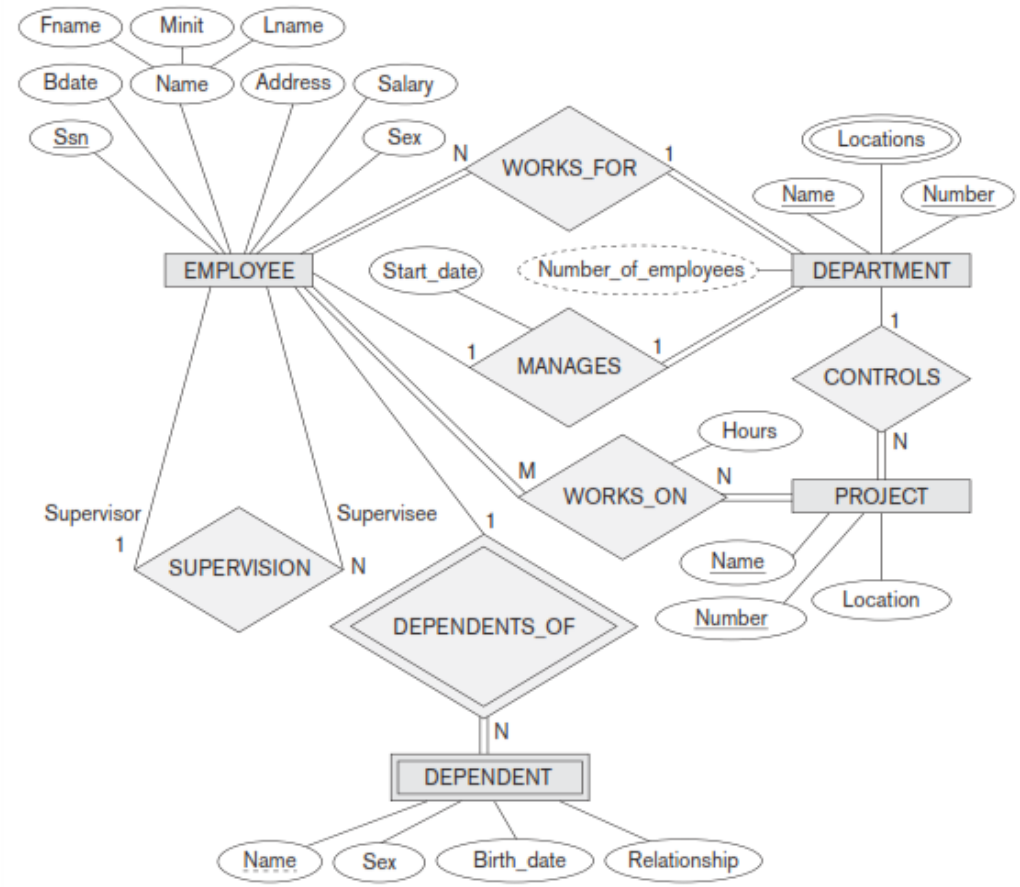


Figure. **categories (UNION TYPES)**

4. 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.



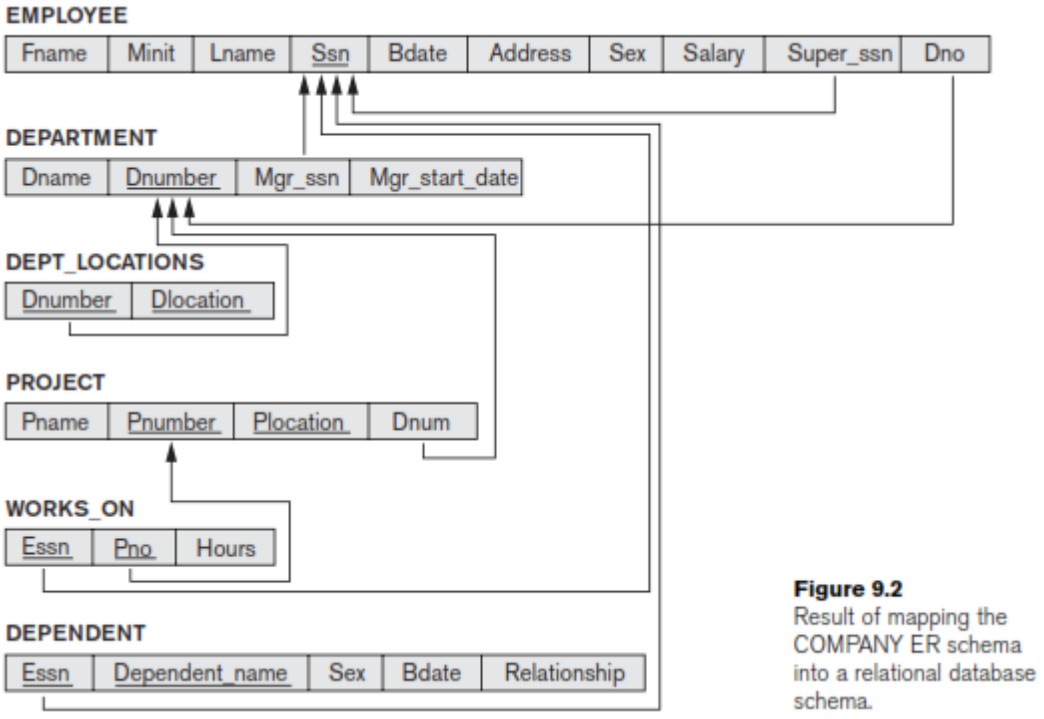


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

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.

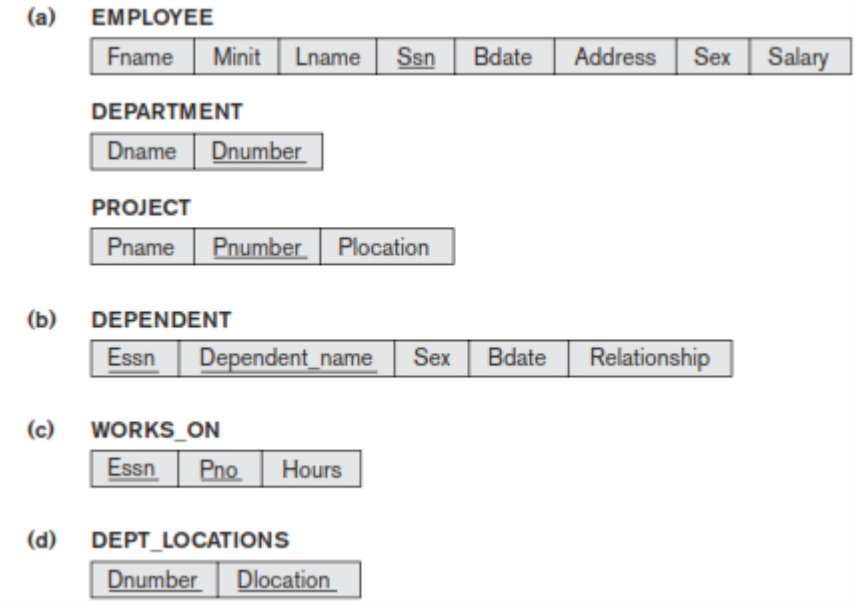
Figure . Illustration of some mapping steps.

(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.



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 values 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 composite, 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

5. Explain about functional dependency and nonloss decomposition.

- A functional dependency is a constraint between two sets of attributes from the database.
- A **functional dependency**, denoted by X → 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.

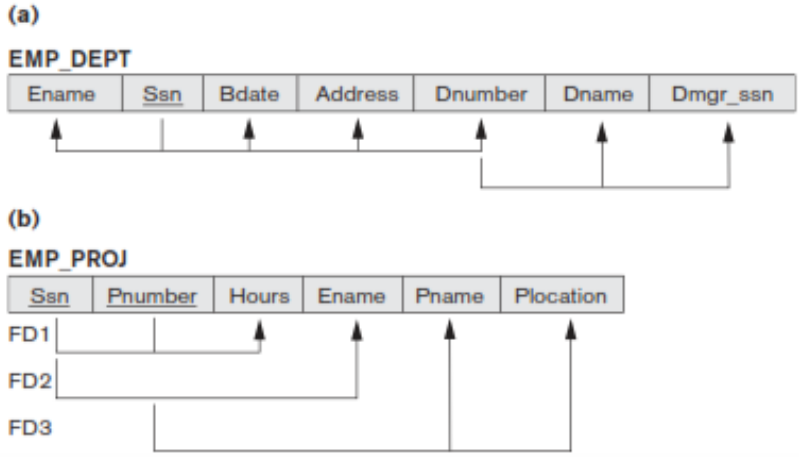
- Consider the relation schema EMP_PROJ. From the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

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.

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.

6. **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.**

(or)

Explain the boyce-code normal form with an example. Also state how it differs from that of 3NF.
Discuss join dependencies and fifth normal form, and explain why 5NF?
What are normal forms? Explain the type of normal forms with suitable example.
Describe about the multi_valued dependencies and fourth normal form with suitable example.

- 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.

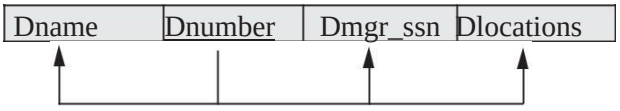
- **Denormalization** is the process of storing the join of higher nor-mal form relations as a base relation, which is in a lower normal form.
- A **superkey** of a relation schema $R = \{A_1, A_2, ..., A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in 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.

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**.
- **A relation is in 1NF if the domain of attribute must be an single atomic value**.
- 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

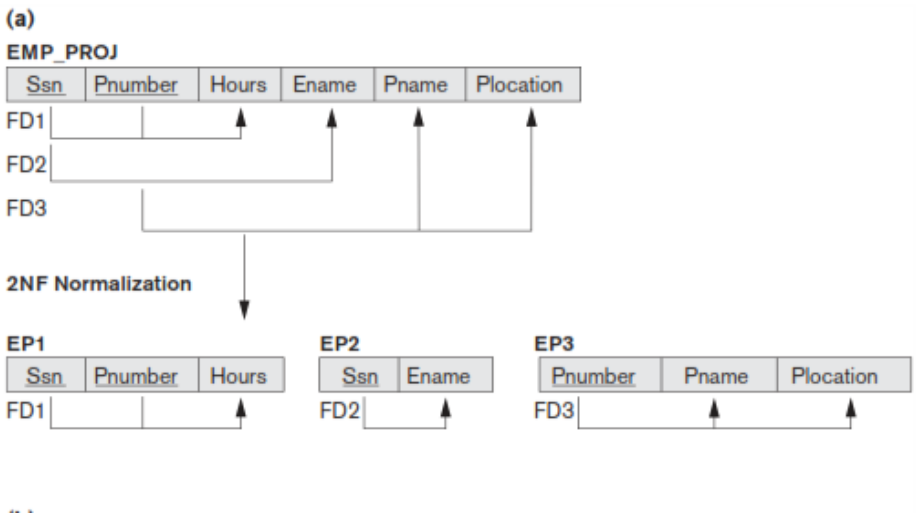
Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bangalore
Research	5	333445555	Chennai
Research	5	333445555	Delhi
Administration	4	987654321	Chennai
Headquarters	1	888665555	Delhi

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.

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.



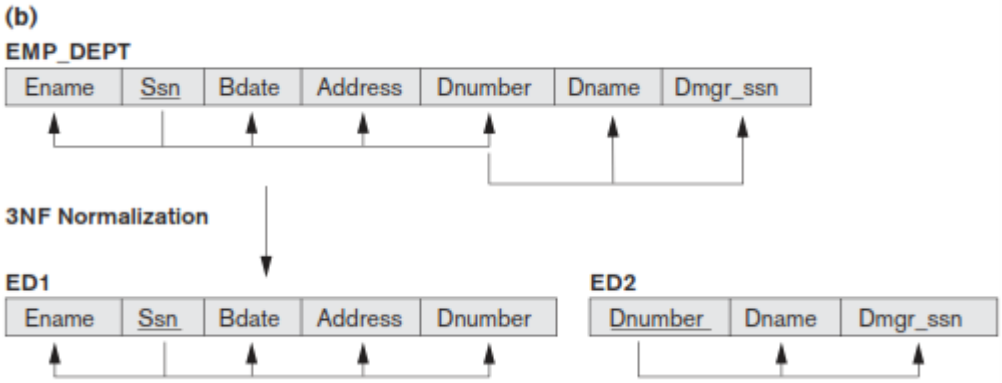
- The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure is in 1NF but is not in 2NF.
- The nonprime attribute Ename 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.

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.

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 relation 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.

MULTIVALUED DEPENDENCY AND FOURTH NORMAL FORM

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 $X \rightarrow\rightarrow Y$ in F, X is a superkey for R.

Definition. A multivalued dependency $X \rightarrow\rightarrow Y$ 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 $X \rightarrow\rightarrow Y$ holds, we say that X **multidetermines** Y. Because of the symmetry in the definition, whenever $X \rightarrow\rightarrow Y$ holds in R, so does $X \rightarrow\rightarrow Z$. Hence, $X \rightarrow\rightarrow Y$ implies $X \rightarrow\rightarrow Z$, and therefore it is sometimes written as $X \rightarrow\rightarrow Y|Z$.

- An MVD $X \rightarrow\rightarrow Y$ in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) $X \cup Y=R$.

- For example, the relation EMP_PROJECTS in Figure 15.15(b) has the trivial MVD Ename $\rightarrow\rightarrow$ 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.

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 relation will be in 4NF if it is in Boyce Codd normal form and has only one trivial multi-valued dependency.**

(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.

JOIN DEPENDENCIES AND FIFTH NORMAL FORM

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)

5NF

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

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-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?

    Normalization 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 → D , D → C , C → 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 (1^{st}, 2^{nd}, and 3rd, BCNF, 4^{th}, 5^{th} 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.

CS3492 DATABASE MANAGEMENT SYSTEMS

UNIT III TRANSACTIONS

Transaction Concepts – ACID Properties – Schedules – Serializability – Transaction support in SQL –Need for Concurrency – Concurrency control –Two Phase Locking- Timestamp – Multiversion – Validation and Snapshot isolation– Multiple Granularity locking – Deadlock Handling – Recovery Concepts – Recovery based on deferred and immediate update – Shadow paging – ARIES Algorithm

---

1. Explain about Transaction concept and schedules.
(or)
Discuss about transaction states , ACID properties and schedules.

- 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 operationsexecuted 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.

ACID Properties:

Properties of the transactions:

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

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

3) Isolation. Even though multiple transactions may execute concurrently, thesystem 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 executingconcurrently in the system.

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

These properties are often called the **ACIDproperties**; 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).

STATES OF TRANSACTION:

- 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.

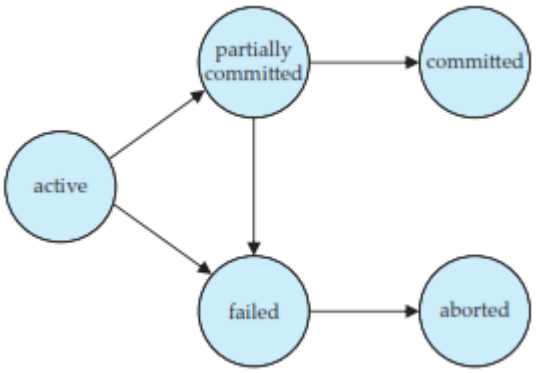


Figure. State diagram of a transaction.

- 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 databaseinto a new consistent state, which must persist even if there is a system failure.

- 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 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.

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

Two good reasons for allowing concurrency:

1) Improved throughput and resource utilization.
2) Reduced waiting time.

- Consider 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:

T_1: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:

T_2: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_2appearing in the right column. The final values of accounts A and B, after theexecution 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.

- The first execution sequence (T_1 followed by T_2) is referred as schedule 1, and the second execution sequence (T_2 followed by T_1) referred 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.
- Schedules include **commit** operation to indicate that the transaction has entered the committed state.

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.

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.

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.

2. Discuss View Serializability and conflict Serializability.
(or)
Explain about serializability and precedence graph.

A serializable schedule always leaves the database in consistent state. A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.

A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

Types of Serializability

1) Conflict serializability.
2) View serializablity

1) CONFLICT SERIALIZABILITY:

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. A schedule *S* is conflict serializable if it is conflict equivalent to a serial schedule.

- 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*). ----------Non conflicting instruction.
  b. *I* =read(*Q*), *J* =write(*Q*) ----------conflicting instruction.
  c. *I* =write(*Q*), *J* =read(*Q*) ----------conflicting instruction.
  d. *I* =write(*Q*), *J* =write(*Q*) ---------- conflicting instruction.

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.

- 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. 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.
- 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 equivalentto 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.

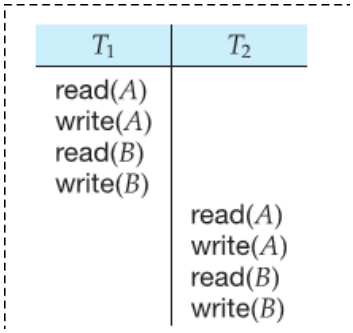


Figure . Schedule 6 — a serial schedule that is equivalent to schedule 3.

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 thetransactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$for which one of three conditions holds:

  ✓ $T_i$executeswrite(Q) before$T_j$executesread(Q).

- ✓ $T_i$executesread(Q) before$T_j$executeswrite(Q).
- ✓ $T_i$executeswrite(Q) before$T_j$executeswrite(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).

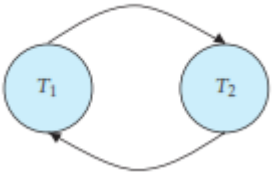


Figure. Precedence graph for schedule 4.

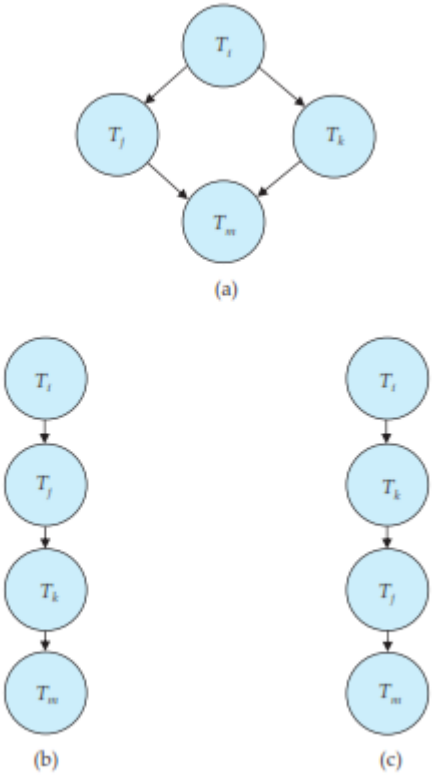


Figure . Illustration of topological sorting.

- 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).

Recoverable Schedules:

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-tionT_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:

- Consider two schedules S and S' , where the same set of transactions participates 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 .

- 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. Discuss about Transaction Support in SQL with an example.

SQL transaction is is a logical unit of work and is guaranteed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged.

With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK. Every transaction has certain characteristics attributed to it. These characteristics are specified by a SET TRANSACTION statement in SQL. The characteristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.

The **access mode** can be specified as READ ONLY or READ WRITE. The default is READ WRITE, unless the isolation level of READ UNCOMMITTED is specified , in which case READ

ONLY is assumed. A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed. A mode of READ ONLY, as the name implies, is simply for data retrieval.

The **diagnostic area size** option, DIAGNOSTIC SIZE *n*, specifies an integer value *n*, which indicates the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or exceptions) to the user or program on the *n* most recently executed SQL statement.

The **isolation level** option is specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be **READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.** The default isolation level is SERIALIZABLE, although some systems use READ COMMITTED as their default. The use of the term SERIALIZABLE here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms, and it is thus not identical to the way serializability. If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three violations may occur:

1. **Dirty read.** A transaction T_1 may read the update of a transaction T_2, which has not yet committed. If T_2 fails and is aborted, then T_1 would have read a value that does not exist and is incorrect.
2. **Nonrepeatable read.** A transaction T_1 may read a given value from a table. If another transaction T_2 later updates that value and T_1 reads that value again, T_1 will see a different value.
3. **Phantoms.** A transaction T_1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T_2 inserts a new row *r* that also satisfies the WHERE-clause condition used in T_1, into the table used by T_1. The record *r* is called a **phantom record** because it was not there when T_1 starts but is there when T_1 ends. T_1 may or may not see the phantom, a row that previously did not exist. If the equivalent serial order is T_1 followed by T_2, then the record *r* should not be seen; but if it is T_2 followed by T_1, then the phantom record should be in the result given to T_1. If the system cannot ensure the correct behavior, then it does not deal with the phantom record problem.

Table. Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Table summarizes the possible violations for the different isolation levels. An entry of *Yes* indicates that a violation is possible and an entry of *No* indicates that it is not possible. READ

UNCOMMITTED is the most forgiving, and SERIALIZABLE is the most restrictive in that it avoids all three of the problems mentioned above.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQL ERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
        READ WRITE
        DIAGNOSTIC SIZE 5
        ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)VALUES
                ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
            SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

The above transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2. If an error occurs on any of the SQL statements, the entire transaction is rolled back. This implies that any updated salary (by this transaction) would be restored to its previous value and that the newly inserted row would be removed.

4. **What is concurrency control? How is it implemented in DBMS? Illustrate with a suitable example. (or)**
   Explain briefly about two phase commit. (or) Explain about locking protocols. (or)
   Explain rigorous two phase locking protocol. (or) Explain strict two phase locking protocol.

Concurrency control is the process of managing simultaneous operations on the database without having them interfere with each other.

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** (denotedby 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.

- 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. Thetransaction 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.

- 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 theunlock(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.

  T_1: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.**TransactionT_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 $A + B$..

  T_2: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	concurrency-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.

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.

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 anylock.
2) Shrinking phase. A transaction may release locks, but may not obtain anynew 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 LOCKINGPROTOCOL:

- Another variant of two-phase locking is the **rigorous two-phase lockingprotocol**, 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 beallowed 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.
- **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, thesystem 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.

IMPLEMENTATION OF LOCKING:

- 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 **locktable**.
- Each record of the linked list for a data item notes which transaction madethe 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.

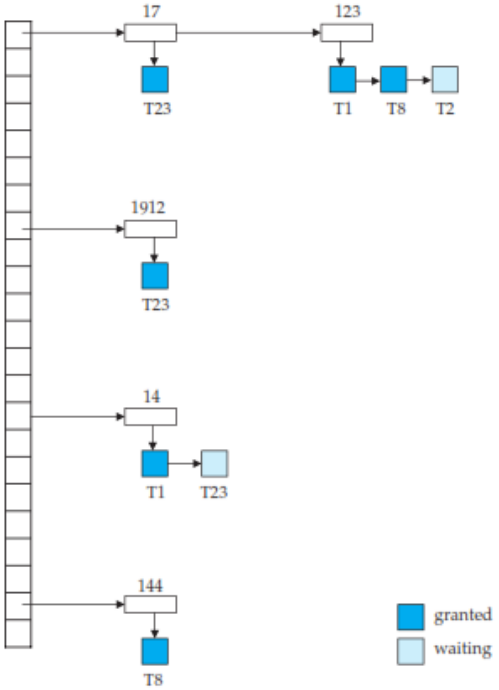


Figure. Lock table.
This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted.

5. Explain about tree protocol. (Or) Explain about graph based 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, . . . ,d_h}of all data items. Ifd_i→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 protocolsare 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*).

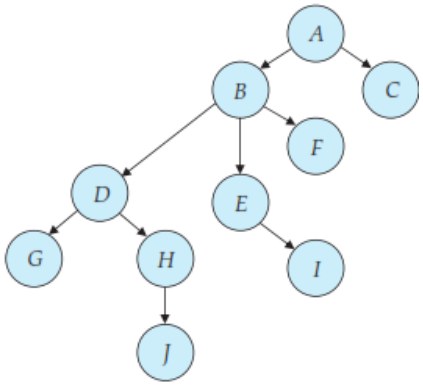


Figure. Tree-structured database graph.

- 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.

Advantage:

- It is deadlock-free, so no rollbacks are required.
- Unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

Disadvantage:

- 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.

6. Explain in detail about 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 dataitem 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.

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:

1) 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.

2) 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.

3) 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.

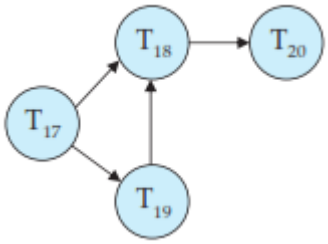


Figure. Wait-for graph with no cycle.

- 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 removedonly 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 inFigure 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.

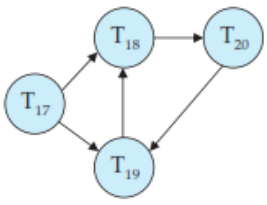


Figure. Wait-for graph with a cycle.

- 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 moretransactions 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 rolledback, we must determine how far this transaction should be rolled back.

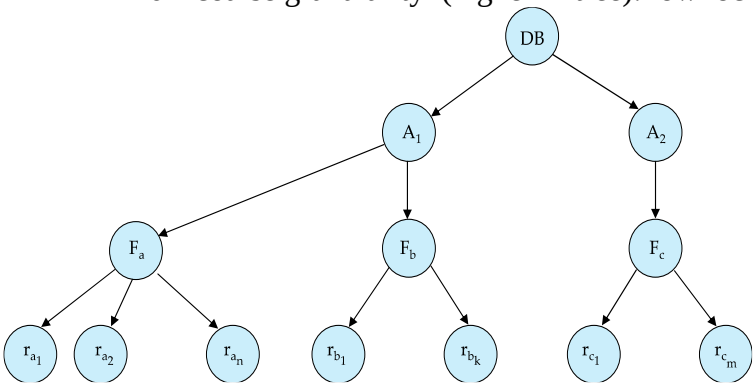
- The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all 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.

- **Starvation**. In a system where the selection of victims is based primarily oncost 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.

7. Explain about Multiple Granularity in details.

- Granularity is the size of data that is allowed to be locked. Multiple granularity is hierarchically breaking up our database into smaller blocks that can be locked. This locking technique allows the locking of various data sizes and sets. This way of breaking the data into blocks that can be locked decreases lock overhead and increases the concurrency in our database.

- **Multiple Granularity a**llow the data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- It Can be represented graphically as a tree.
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- **Granularity of locking** (level in tree where locking is done):
    - fine granularity (lower in tree): high concurrency, high locking overhead
    - coarse granularity (higher in tree): low locking overhead, low concurrency



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes:

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
    - ***Intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
    - ***Intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
    - ***Shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.
- The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  - The lock compatibility matrix must be observed.
  - The root of the tree must be locked first, and may be locked in any mode.
  - A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  - A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  - T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  - T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- Lock granularity escalation: in case there are too many locks at a particular level, switch to higher granularity S or X lock

8. Explain about timestamp-based protocols in details.

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully.
  - R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- **Suppose a transaction T_i issues a read(Q)**
  - If TS(T_i) ≤ W-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
    - Hence, the read operation is rejected, and T_i is rolled back.
  - If TS(T_i) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to max(R-timestamp(Q), TS(T_i)).
- **Suppose that transaction T_i issues write(Q).**
  - If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the write operation is rejected, and T_i is rolled back.
  - If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
    - Hence, this write operation is rejected, and T_i is rolled back.
  - Otherwise, the write operation is executed, and W-timestamp(Q) is set to TS(T_i).

Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Thomas' Write Rule:

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

9. Explain about the validation-based protocol.

(or)

Explain about Optimistic Concurrency Control Technique.

- **Validation Based Protocol** is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding

concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs, therefore, there is no need for checking while the transaction is executed.

- In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database. All updates are applied to local copies of data items kept for the transaction. At the end of transaction execution, while execution of the transaction, a **validation phase** checks whether any of transaction updates violate serializability. If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.

- Execution of transaction T_i is done in three phases.

1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.

- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time; this is done to increase concurrency.
  - Thus, TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Validation Test for Transaction T_j:

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.
- then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - The writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - The writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation:

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B$ 50
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

10. Discuss about multiversion schemes.

In **multiversion concurrency-control** schemes, each write(Q) operation creates a new **version** of Q. When a transaction issues a read(Q) operation, the concurrency-control manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering:

- Each data item Q has a sequence of versions <Q_1, Q_2,...., Q_m>. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  1. If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.

2. If transaction T_i issues a **write**(Q)
    1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
    2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
    3. else a new version of Q is created.

- Observe that
    - Reads always succeed
    - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability

Multiversion Two-Phase Locking:

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
    - Each successful **write** results in the creation of a new version of the data item written.
    - Each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

- When an update transaction wants to read a data item:
    - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
    - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction T_i completes, commit processing occurs:
    - T_i sets timestamp on the versions it has created to **ts-counter** + 1
    - T_i increments **ts-counter** by 1
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues:

- Creation of multiple versions increases storage overhead
    - Extra tuples
    - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
    - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

11. Explain about the snapshot isolation.

- The isolation levels in DBMS are used to maintain concurrent execution of transactions without facing interruption through problems like dirty read, phantom read, and non-repeatable read. Snapshot isolation is one such isolation level that achieves the maximum level of concurrency.

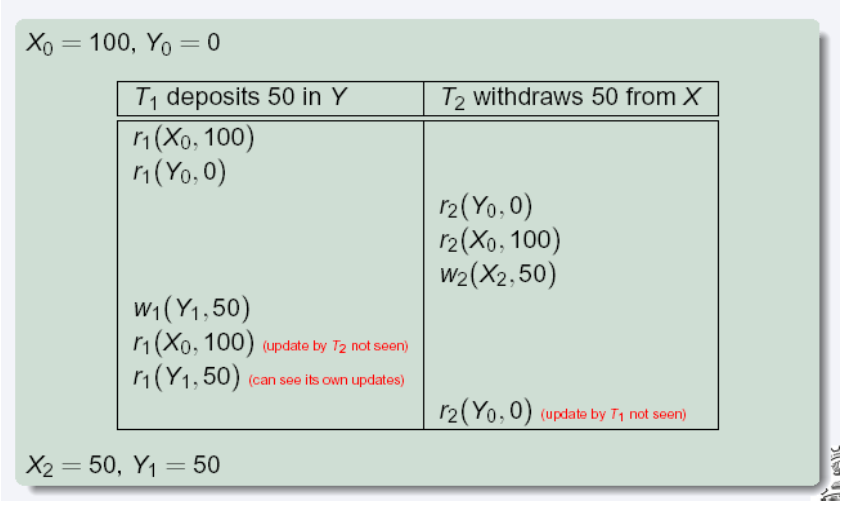
- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
    - Poor performance results
- Solution 1: Give logical "snapshot" of database state to read only transactions, read-write transactions use normal locking
    - Multiversion 2-phase locking
    - Works well, but how does system know a transaction is read only?
- Solution 2: Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
    - Problem: variety of anomalies such as lost update can result
    - Partial solution: snapshot isolation level
        - Proposed by Berenson et al, SIGMOD 1995
        - Variants implemented in many database systems
            - E.g. Oracle, PostgreSQL, SQL Server 2005

- A transaction T1 executing with Snapshot Isolation
    - takes snapshot of committed data at start
    - always reads/modifies data in its own snapshot
    - updates of concurrent transactions are not visible to T1
    - writes of T1 complete when it commits
    - **First-committer-wins rule**:
        - Commits only if no other concurrent transaction has already written data that T1 intends to write.

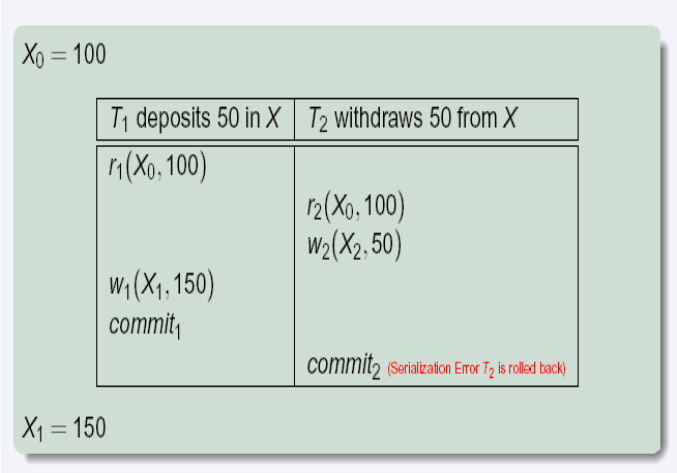
T1	**T2**	**T3**
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Snapshot Read:

- Concurrent updates invisible to snapshot read



Snapshot Write: First Committer Wins



  - Variant: "**First-updater-wins**"
    - Check for concurrent updates when write occurs by locking item
      - But lock should be held till all concurrent transactions have finished
    - (Oracle uses this plus some extra features)
    - Differs only in when abort occurs, otherwise equivalent

Benefits of SI:

- Reading is *never* blocked,
  - and also doesn't block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- Problems with SI
  - SI does not always give serializable executions

        - Serializable: among two concurrent txns, one sees the effects of the other
        - In SI: neither sees the effects of the other
    - Result: Integrity constraints can be violated
- **Snapshot Isolation:**
- E.g. of problem with SI
    - T1: x:=y
    - T2: y:= x
    - Initially x = 3 and y = 17
        - Serial execution: x = ??, y = ??
        - if both transactions start at the same time, with snapshot isolation: x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
    - E.g:
        - Find max order number among all orders
        - Create a new order with order number = previous max + 1

Snapshot Isolation Anomalies:

- SI breaks serializability when txns modify *different* items, each based on a previous state of the item the other modified
    - Not very common in practice
        - E.g., the TPC-C benchmark runs correctly under SI
        - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
    - But does occur
        - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
    - Integrity constraint checking usually done outside of snapshot

12. Explain about Recovery Concepts in detail.

(Or)

Discuss about Recovery Techniques Based on deferred update and Immediate Update.

- Recovery process restores database to most recent consistent state before time of failure
- Information kept in system log
- Typical recovery strategies
    - Restore backed-up copy of database
        - Best in cases of extensive damage
    - Identify any changes that may cause inconsistency
        - Best in cases of noncatastrophic failure
        - Some operations may require redo
- Deferred update techniques
    - Do not physically update the database until after transaction commits
    - Undo is not needed; redo may be needed
- Immediate update techniques

- Database may be updated by some operations of a transaction before it reaches commit point
- Operations also recorded in log
- Recovery still possible

- Undo and redo operations required to be idempotent
  - Executing operations multiple times equivalent to executing just once
  - Entire recovery process should be idempotent
- Caching (buffering) of disk blocks
  - DBMS cache: a collection of in-memory buffers
  - Cache directory keeps track of which database items are in the buffers

- Cache buffers replaced (flushed) to make space for new items
- Dirty bit associated with each buffer in the cache
  - Indicates whether the buffer has been modified
- Contents written back to disk before flush if dirty bit equals one
- Pin-unpin bit
  - Page is pinned if it cannot be written back to disk yet
- Main strategies
  - In-place updating
    - Writes the buffer to the same original disk location
    - Overwrites old values of any changed data items
  - Shadowing
    - Writes an updated buffer at a different disk location, to maintain multiple versions of data items
    - Not typically used in practice
- Before-image: old value of data item
- After-image: new value of data item
- Write-ahead logging
  - Ensure the before-image (BFIM) is recorded
  - Appropriate log entry flushed to disk
  - Necessary for UNDO operation if needed
- UNDO-type log entries
- REDO-type log entries

- Steal/no-steal and force/no-force
  - Specify rules that govern when a page from the database cache can be written to disk
- No-steal approach
  - Cache buffer page updated by a transaction cannot be written to disk before the transaction commits
- Steal approach
  - Recovery protocol allows writing an updated buffer before the transaction commits
- Force approach
  - All pages updated by a transaction are immediately written to disk before the transaction commits
  - Otherwise, no-force
- Typical database systems employ a steal/no- force strategy
  - Avoids need for very large buffer space

- Reduces disk I/O operations for heavily updated pages
- Write-ahead logging protocol for recovery algorithm requiring both UNDO and REDO
  - BFIM of an item cannot be overwritten by its after image until all UNDO-type log entries have been force-written to disk
  - Commit operation of a transaction cannot be completed until all REDO-type and UNDO-type log records for that transaction have been force- written to disk

Checkpoints in the System Log and Fuzzy Checkpointing:

- Taking a checkpoint
  - Suspend execution of all transactions temporarily
  - Force-write all main memory buffers that have been modified to disk
  - Write a checkpoint record to the log, and force- write the log to the disk
  - Resume executing transactions
- DBMS recovery manager decides on checkpoint interval
- Fuzzy checkpointing
  - System can resume transaction processing after a begin_checkpoint record is written to the log
  - Previous checkpoint record maintained until end_checkpoint record is written

Transaction Rollback:

- Transaction failure after update but before commit
  - Necessary to roll back the transaction
  - Old data values restored using undo-type log entries
- Cascading rollback
  - If transaction T is rolled back, any transaction S that has read value of item written by T must also be rolled back
  - Almost all recovery mechanisms designed to avoid this

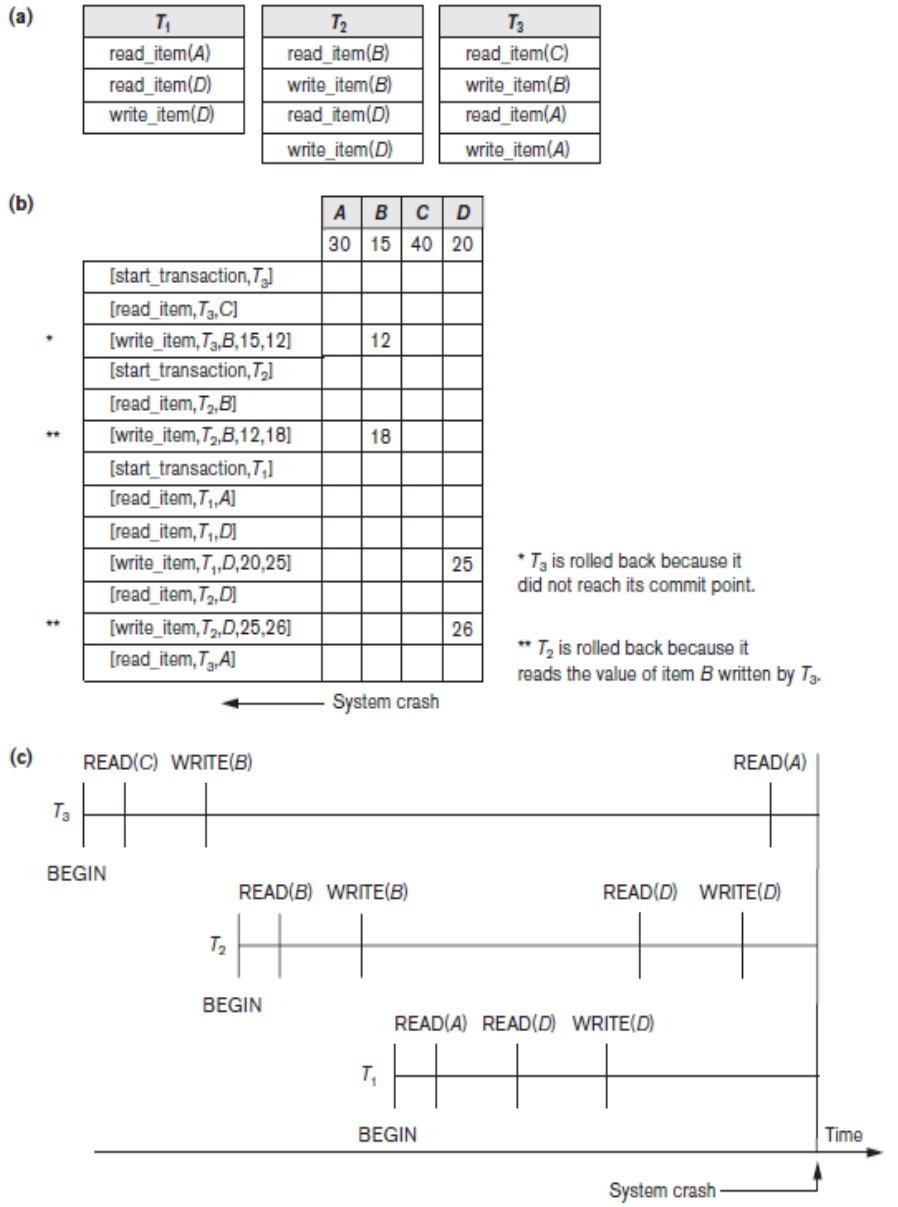


Figure 22.1 Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules) (a) The read and write operations of three transactions (b) System log at point of crash (c) Operations before the crash

Transactions that Do Not Affect the Database:

- Example actions: generating and printing messages and reports
- If transaction fails before completion, may not want user to get these reports
  - Reports should be generated only after transaction reaches commit point
- Commands that generate reports issued as batch jobs executed only after transaction reaches commit point
  - Batch jobs canceled if transaction fails

NO-UNDO/REDO Recovery Based on Deferred Update:

- Deferred update concept
  - Postpone updates to the database on disk until the transaction completes successfully and reaches its commit point
  - Redo-type log entries are needed
  - Undo-type log entries not necessary

- Can only be used for short transactions and transactions that change few items
  - Buffer space an issue with longer transactions
- Deferred update protocol
  - Transaction cannot change the database on disk until it reaches its commit point
    - All buffers changed by the transaction must be pinned until the transaction commits (no-steal policy)
  - Transaction does not reach its commit point until all its REDO-type log entries are recorded in log and log buffer is force-written to disk

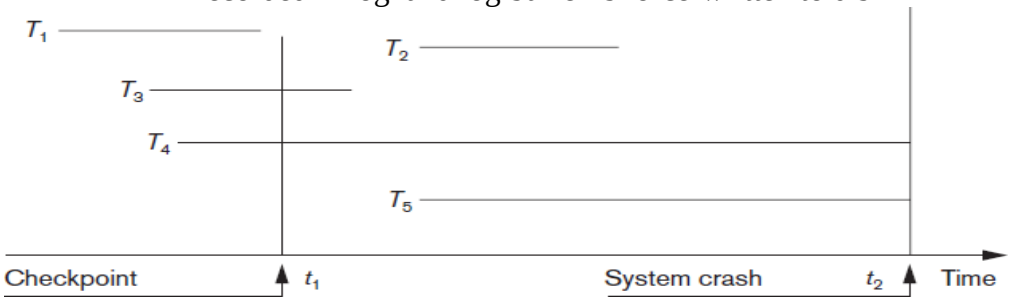


Figure 22.2 An example of a recovery timeline to illustrate the effect of checkpointing

Recovery Techniques Based on Immediate Update

- Database can be updated immediately
  - No need to wait for transaction to reach commit point
  - Not a requirement that every update be immediate
- UNDO-type log entries must be stored
- Recovery algorithms
  - UNDO/NO-REDO (steal/force strategy)

UNDO/REDO (steal/no-force strategy)

T_1
read_item(*A*)
read_item(*D*)
write_item(*D*)

T_2
read_item(*B*)
write_item(*B*)
read_item(*D*)
write_item(*D*)

T_3
read_item(*A*)
write_item(*A*)
read_item(*C*)
write_item(*C*)

T_4
read_item(*B*)
write_item(*B*)
read_item(*A*)
write_item(*A*)

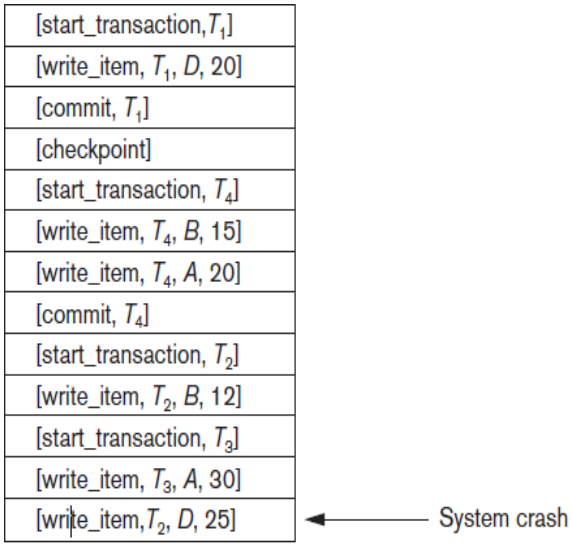


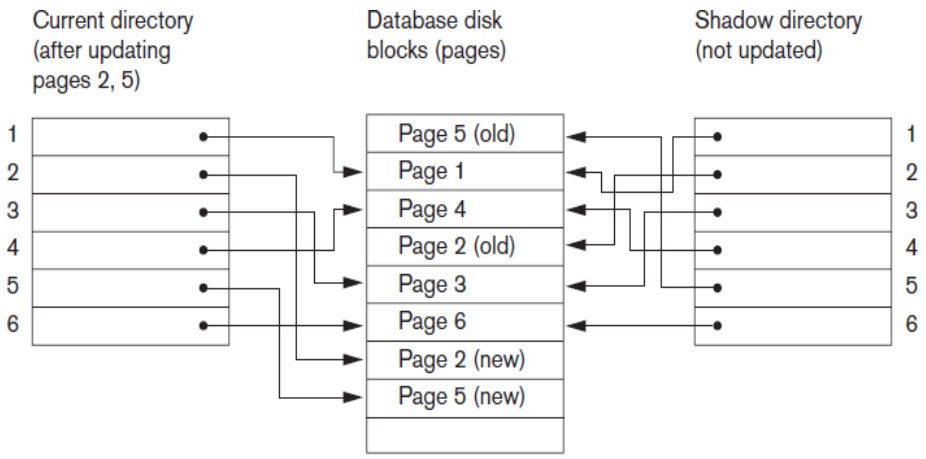
Figure. An example of recovery using deferred update with concurrent transactions
(a) The READ and WRITE operations of four transactions (b) System log at the point of crash

13. Discuss about Shadow Paging.

- No log required in a single-user environment
  - Log may be needed in a multiuser environment for the concurrency control method
- Shadow paging considers disk to be made of *n* fixed-size disk pages
  - Directory with *n* entries is constructed
  - When transaction begins executing, directory copied into shadow directory to save while current directory is being used

Shadow directory is never modified

- New copy of the modified page created and stored elsewhere
  - Current directory modified to point to new disk block
  - Shadow directory still points to old disk block
- Failure recovery
  - Discard current directory
  - Free modified database pages
  - NO-UNDO/NO-REDO technique



[5]The directory is similar to the page table maintained by the operating system for each process.

Figure. An example of shadow paging

14. Elaborate about the ARIES Recovery Algorithm.

- **ARIES Recovery Algorithm** is Used in many IBM relational database products
- It Uses a steal/no-force approach for writing
- Concepts
  - Write-ahead logging
  - Repeating history during redo
    - Retrace all database system actions prior to crash to reconstruct database state when crash occurred
  - Logging changes during undo
    - Prevents ARIES from repeating completed undo operations if failure occurs during recovery
- Analysis step
  - Identifies dirty (updated) pages in the buffer and set of transactions active at the time of crash
  - Determines appropriate start point in the log for the REDO operation
- REDO
  - Reapplies updates from the log to the database
  - Only necessary REDO operations are applied

- UNDO
  - Log is scanned backward
  - Operations of transactions that were active at the time of the crash are undone in reverse order
- Every log record has associated log sequence number (LSN)
  - Indicates address of log record on disk
  - Corresponds to a specific change of some transaction

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	7
B	2
A	6

Figure. An example of recovery in ARIES (a) The log at point of crash (b) The Transaction and Dirty Page Tables at time of checkpoint (c) The Transaction and Dirty Page Tables after the analysis phase

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.

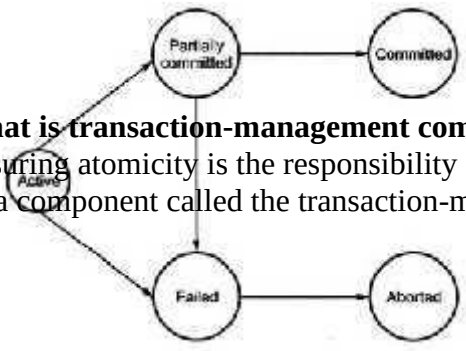
6. **List the four conditions for deadlock. Nov/ Dec 2016**
   i. mutual exclusion
   ii. hold and wait
   iii. no pre-emption
   iv. circular wait
7. **What type of locking is needed for insert and delete operations? April/ May 2017**

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?

The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil. • Starvation is also possible if concurrency control manager is badly designed. For example: – A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. – The same transaction is repeatedly rolled back due to deadlocks. • Concurrency control manager can be designed to prevent starvation.

23. Define recovery in a database system.

Recovery in a database system means, primarily, recovering the database itself; that is, restoring the database to a correct state after some failure has rendered thecurrent state incorrect, or at least suspect.

24. What is a transaction? (MAY 2010)

A transaction is a logical unit of work; it begins with the execution of aBEGIN TRANSACTION operation, and ends with the execution of a COMMIT or ROLLBACK operation.

25. What is a commit point?

A commit point corresponds to the end of a logical unit of work, andhence to a point at which the database is supposed to be in a correct state.

26. What is a system failure?

System failures (e.g., power outage), which affect all transactions currently in progress but do not physically damage the database. A systemfailure is sometimes called a soft crash.

27. What is a media failure?

Media failures (e.g. head crash on the disk), which do cause damage to the database or some portion thereof, and affect at least those transactions currently using that portion. A media failure is sometimes called a hard crash.

28. What is a check point record?

The checkpoint record contains a list of all transactions that were inprogress at the time the checkpoint was taken.

29. Expand ARIES.

The name ARIES stands for –Algorithms for Recovery and Isolation Exploiting Semantics.

30. What are the three broad phases of ARIES?

ARIES operates in three broad phases:

- **Analysis:** Build the REDO and UNDO lists
- **Redo:** Start from a position in the log determined in the analysis phase and restore the database to the state it was in at the time of thecrash.
- **Undo:** Undo the effects of transactions that failed to commit.

31. What is a two-phase commit?

Two-phase commit is important whenever a given transaction can interact with several independent –resource managers, each managing its own set of recoverable resources and maintaining its own recovery log.

32. What is the need for save points?

It might be possible for a transaction to establish intermediate save points while it is executing, and subsequently to roll back to a previously established save point, if required, instead of having to roll back all the way to the beginning.

33. Define concurrency (MAY 2012)

Concurrency refers to the fact that DBMSs typically allow many transactions to access the same database at the same time.

34. What are the three problems that any concurrency control mechanism must address?

The three problems are:

- The lost update problem
- The uncommitted dependency problem
- The inconsistent analysis problem

35. What is the last update problem?

Transaction A retrieves some tuple t at time t1; transaction B retrieves that same tuple t at time t2; transaction A updates the tuple at time t3; and transaction B updates the same tuple at time t4; Transaction A_s update is lostat time t4, because transaction B overwrites it with-out even looking at it.

36. What is the uncommitted dependency problem?

The uncommitted dependency problem arises if one transaction is allowed to retrieve-or, worse, update-a tuple that has been updated by anothertransaction but not yet committed by that other transaction.

37. What is an isolation level?

The isolation level that applies to a given transaction might be defined as the degree of interference the transaction in question is prepared to tolerate on the part of concurrent transactions.

38. What do you mean by phantom problem?

If transactions operate at less than the maximum isolation level is the so-called phantom problem

39. What is an intent locking protocol?

The intent locking protocol, according to which no transaction is allowed to acquire a lock on a tuple before first acquiring a lock-probably an intent lock on the relvar that contains it.

40. What is a shadow copy scheme?

It is simple, but efficient, scheme called the shadow copy schemes. It is based o nmaking copies of the database called shadow copies that one transaction is active at a time. The scheme also assumes that the database is simply a file on disk.

41. What type of locking needed for insert and delete operations (April/May-2017)

When you execute an INSERT, UPDATE, or DELETE statement, the database server uses exclusive locks. An exclusive lock means that no other users can update or delete the item until the database server removes the lock

Part B

1 Discuss View Serializability and conflict Serializability. ***Nov/ Dec 2015***

2 Explain briefly about two phase commit. ***May/ June 2016, (8) May/ June 2015, (6) Nov/ Dec 2016***

3 Explain about locking protocols. ***May/ June 2016***

4 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?

5 Discuss the violations caused by each of the following: Dirty read, non-repeatable read and phantoms with suitable example. ***April/ May 2017***

6 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***

7 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***

8 Why is Recovery needed? Discuss any two Recovery Techniques. (MAY2012)

9 Explain the following protocols for concurrency control **(MAY 2008)**

a) Lock based protocols.
b) Time stamp based protocols.

10 Explain testing for serializability with respect to concurrency controlschemes. How will you determine, whether a schedule is serializble or not.**(MAY 2008)**

11 Explain the deferred and immediate-modification version of the log basedRecovery scheme.(MAY 2007)

12 Write short notes on shadow paging**.(DEC 2007)**

13 What is a transaction? Draw the state diagram corresponding to a transaction and present an outline of the same.

14 Outline the properties that must be satisfied by a transaction.

15 Explain about Timestamp based protocol.
16 Discuss about Multiversion based protocol.
17 Elaborate about Validation protocol for concurrency control.
18 Explain about Snapshot isolation.
19 Discuss about the Multiple Granularity locking.
20 Discuss about Shadow paging
21 Explain in detail about ARIES Algorithm.
22 What is concurrency? Explain it in terms of locking mechanisms and two phase commit protocol. ***Nov/ Dec 2014***
23 Write short notes on Transaction concept and schedules. ***Nov/ Dec 2014***
24 Explain about Deadlock prevention and detection techniques. ***Nov/ Dec 2014, (16) Nov/ Dec 2015, (6) Nov/ Dec 2016***
25 What is concurrency control? How is it implemented in DBMS? Illustrate with a suitable example ***Nov/ Dec 2015***

CS3492 DATABASE MANAGEMENT SYSTEMS

UNIT IV IMPLEMENTATION TECHNIQUES

RAID – File Organization – Organization of Records in Files – Data dictionary Storage – Column Oriented Storage– Indexing and Hashing –Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for Selection, Sorting and join operations – Query optimization using Heuristics - Cost Estimation.

---

1. Explain about RAID system. How does it improve performance and reliability.

- RAID (redundant array of independent disks; originally redundant array of inexpensive disks) is a way of storing the same data in different places on multiple hard disks to protect data in the case of a drive failure.
- **Redundant arrays of independent disks** (**RAID)**, have been proposed to achieve improved performance and reliability.
- Reliability is **achieved** by **redundancy.** 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.
- **Improvement in Performance is achieved by Parallelism.** 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**.

Block-level striping :

- **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* .

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.

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.

RAID level 0:

- **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.

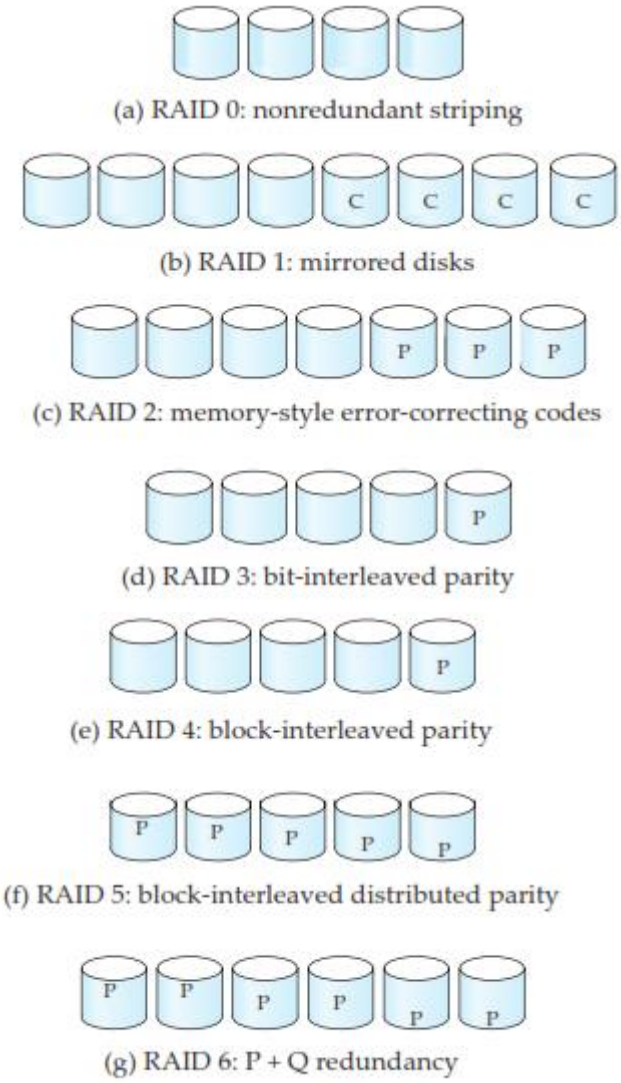


Figure. RAID levels.

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 $4k$, $4k + 1$, $4k + 2$, $4k + 3$ is stored in disk k mod 5; the corresponding blocks of the other four disks store the 4 data blocks $4k$ to $4k + 3$. 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.

Choice of RAID Level:

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).

2. Describe the different types of file organization? Explain using a sketch of each of them with their advantages and disadvantages.

- 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.

- 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.

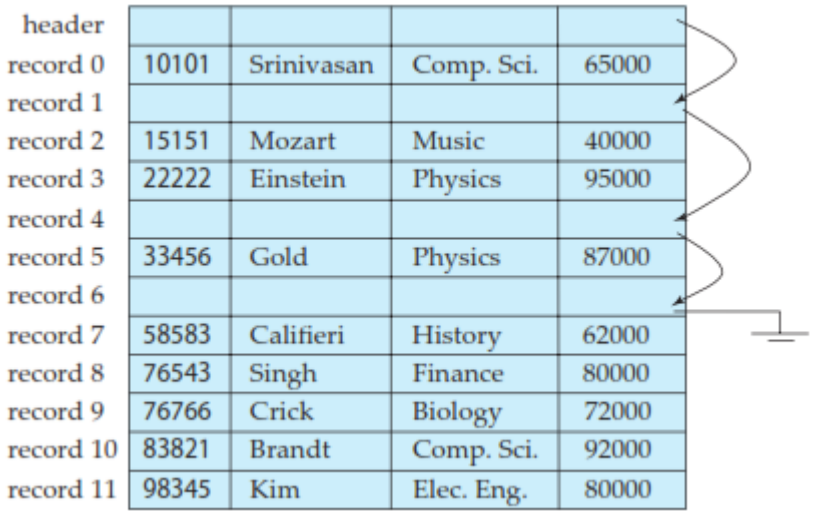
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	CaliÞeri	History	62000
record 8	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.

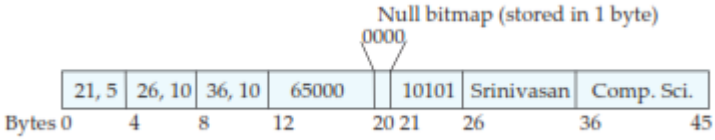


Figure. Representation of variable-length record.

- 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.

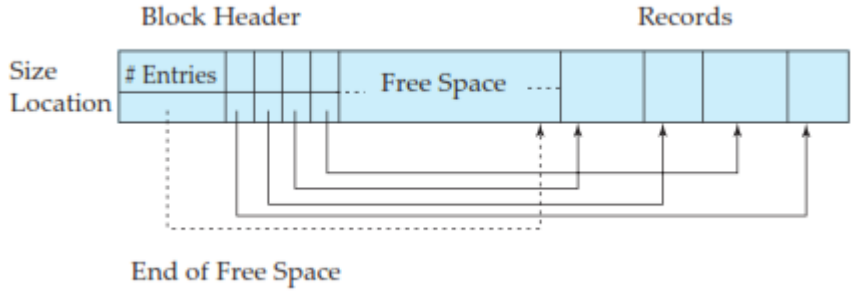


Figure. Slotted-page structure.

- 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.

3. Discuss about organization of records in file.

- 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.

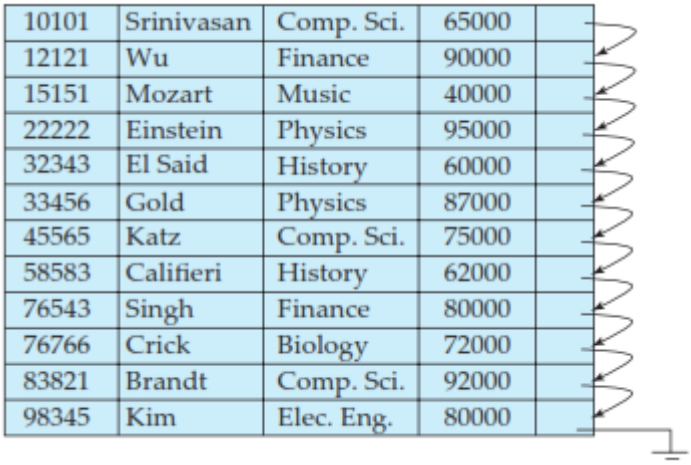
2) Sequential file organization. Records are stored in sequential order, according to the value of a "search key" of each record.

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) <u>Sequential File Organization:</u>

- 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.



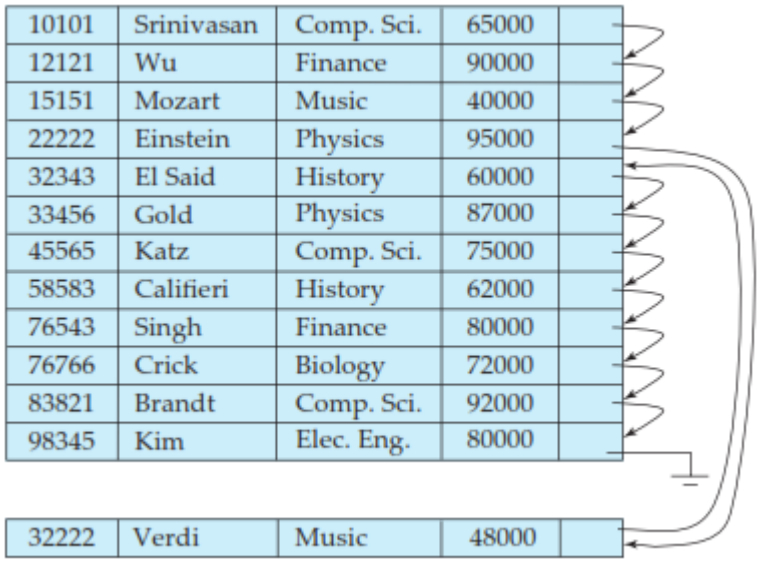
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. 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) Multitable Clustering File Organization:

- 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	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000

Figure. Multitable clustering file structure.

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.

Partitioning

Many databases allow the records in a relation to be partitioned into smaller relations that are stored separately. Such **table partitioning** is typically done on the basis of an attribute value; for example, records in a *transaction* relation in an accounting database may be partitioned by year into smaller relations corresponding to each year, such as *transaction 2018*, *transaction 2019*, and so on. Queries can be written based on the *trans- action* relation but are translated into queries on the year-wise relations. Most accesses are to records of the current year and include a selection based on the year. Query op- timizers can rewrite such a query to only access the smaller relation corresponding to the requested year, and they can avoid reading records corresponding to other years.
For example, a query

select * from ***transaction*** where ***year*=2019**

would only access the relation *transaction 2019*, ignoring the other relations, while a query without the selection condition would read all the relations.
The cost of some operations, such as finding free space for a record, increase with relation size; by reducing the size of each relation, partitioning helps reduce such over- heads. Partitioning can also be used to store different parts of a relation on different storage devices; for example, in the year 2019, *transaction 2018* and earlier year transactions can which are infrequently accessed could be stored on magnetic disk, while *transaction 2019* could be stored on SSD, for faster access.

4. Explain about Data-Dictionary Storage.

A relational database system needs to maintain data *about* the relations, such as the schema of the relations. In general, such "data about data" are referred to as **metadata**. Relational schemas and other metadata about relations are stored in a structure called the **data dictionary** or **system catalog**. Among the types of information that the system must store are these:

- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of views defined on the database, and definitions of those views
- Integrity constraints (e.g., key constraints)

In addition, many systems keep the following data on users of the system:

- Names of users, the default schemas of the users, and passwords or other informa- tion to authenticate users
- Information about authorizations for each user

Further, the database may store statistical and descriptive data about the relations and attributes, such as the number of tuples in each relation, or the number of distinct values for each attribute.
The data dictionary may also note the storage organization (heap, sequential, hash, etc.) of relations, and the location where each relation is stored:

- If relations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation.
- If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.

There is a need to store information about each index on each of the relations:

- Name of the index
- Name of the relation being indexed
- Attributes on which the index is defined

- Type of index formed

All this metadata information constitutes, in effect, a miniature database. Some database systems store such metadata by using special-purpose data structures and code. It is generally preferable to store the data about the database as relations in the database itself. By using database relations to store system metadata, we simplify the overall structure of the system and harness the full power of the database for fast access to system data.

The exact choice of how to represent system metadata by relations must be made by the system designers. We show the schema diagram of a toy data dictionary in Figure, storing part of the information mentioned above. The schema is only illustrative; real implementations store far more information than what the figure shows.

In the metadata representation shown, the attribute *index attributes* of the relation *Index metadata* is assumed to contain a list of one or more attributes, which can be represented by a character string such as "*dept name, building*". The *Index metadata* relation is thus not in first normal form; it can be normalized, but the preceding representation is likely to be more efficient to access. The data dictionary is often stored in a non normalized form to achieve fast access.

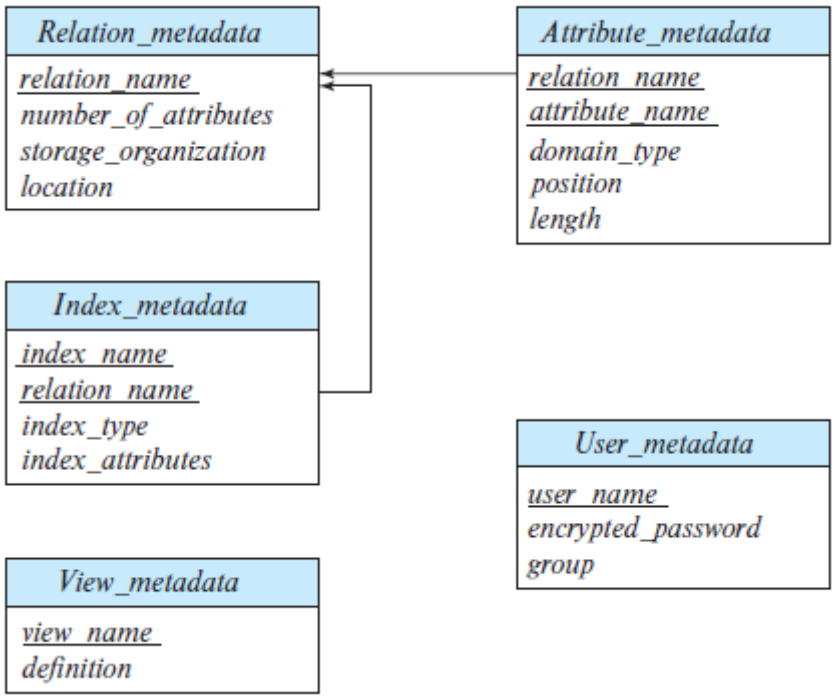


Figure. Relational schema representing part of the system metadata.

- Whenever the database system needs to retrieve records from a relation, it must first consult the *Relation metadata* relation to find the location and storage organization of the relation, and then fetch records using this information.
- However, the storage organization and location of the *Relation metadata* relation itself must be recorded elsewhere (e.g., in the database code itself, or in a fixed location in the database), since we need this information to find the contents of *Relation metadata*.
- Since system metadata are frequently accessed, most databases read it from the database into in-memory data structures that can be accessed very efficiently. This is done as part of

the database startup, before the database starts processing any queries.

5. Discuss about Column-Oriented Storage.

- Databases traditionally store all attributes of a tuple together in a record, and tuples are stored in a file. Such a storage layout is referred to as a *row-oriented storage*.
- In contrast, in **column-oriented storage**, also called a **columnar storage**, each attribute of a relation is stored separately, with values of the attribute from successive tuples stored at successive positions in the file. Figure shows how the *instructor* relation would be stored in column-oriented storage, with each attribute stored separately.
- In the simplest form of column-oriented storage, each attribute is stored in a separate file. Further, each file is *compressed*, to reduce its size.
- If a query needs to access the entire contents of the i^{th} row of a table, the values at the i^{th} position in each of the columns are retrieved and used to reconstruct the row. Column-oriented storage thus has the drawback that fetching multiple attributes of a single tuple requires multiple I/O operations. Thus, it is not suitable for queries that fetch multiple attributes from a few rows of a relation.
- However, column-oriented storage is well suited for data analysis queries, which process many rows of a relation, but often only access some of the attributes.

BENEFITS:

1) **Reduced I/O**.: When a query needs to access only a few attributes of a relation with a large number of attributes, the remaining attributes need not be fetched from disk into memory. In contrast, in row-oriented storage, irrelevant attributes are fetched into memory from disk. The reduction in I/O can lead to significant reduction in query execution cost.

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstei	Physics	95000
32343	n El	History	60000
33456	Said	Physics	87000
45565	Gold	Comp.	75000
58583	Katz	Sci.	62000
76543	Califieri	History	80000
76766	Singh	Finance	72000
83821	Crick	Biology	92000
98345	Brandt	Comp.	80000

Figure Columnar representation of the *instructor* relation.

2) **Improved CPU cache performance**. When the query processor fetches the con- tents of a particular attribute, with modern CPU architectures multiple consecutive bytes, called a cache line, are fetched from memory to CPU cache. If these bytes are accessed later, access is much faster if they are in cache than if they have to be fetched from main memory. However, if these adjacent bytes contain values for attributes that are not be

needed by the query, fetching them into cache wastes memory bandwidth and uses up cache space that could have been used for other data. Column-oriented storage does not suffer from this problem, since adjacent bytes are from the same column, and data analysis queries usually access all these values consecutively.

3) Improved compression. Storing values of the same type together significantly increases the effectiveness of compression, when compared to compressing data stored in row format; in the latter case, adjacent attributes are of different types, reducing the efficiency of compression. Compression significantly reduces the time taken to retrieve data from disk, which is often the highest-cost component for many queries. If the compressed files are stored in memory, the in-memory stor- age space is also reduced correspondingly, which is particularly important since main memory is significantly more expensive than disk storage.

4) Vector processing. Many modern CPU architectures support **vector processing**, which allows a CPU operation to be applied in parallel on a number of elements of an array. Storing data columnwise allows vector processing of operations such as comparing an attribute with a constant, which is important for applying selection conditions on a relation. Vector processing can also be used to compute an aggregate of multiple values in parallel, instead of aggregating the values one at a time.

As a result of these benefits, column-oriented storage is increasingly used in data- warehousing applications, where queries are primarily data analysis queries. It should be noted that indexing and query processing techniques need to be carefully designed to get the performance benefits of column-oriented storage.

Databases that use column-oriented storage are referred to as **column stores**, while databases that use row-oriented storage are referred to as **row stores**.

DRAWBACKS:

1) **Cost of tuple reconstruction**: Reconstructing a tuple from the individual columns can be expensive, negating the benefits of columnar represen- tation if many columns need to be reconstructed. While tuple reconstruction is common in transaction-processing applications, data analysis applications usual output only a few columns out of many that are stored in "fact tables" in data warehouses.
2) **Cost of tuple deletion and update**. Deleting or updating a single tuple in a com- pressed representation would require rewriting the entire sequence of tuples that are compressed as one unit. Since updates and deletes are common in transaction- processing applications, column-oriented storage would result in a high cost for these operations if a large number of tuples were compressed as one unit. In contrast, data-warehousing systems typically do not support updates to tu- ples, and instead support only insert of new tuples and bulk deletes of a large number of old tuples at a time. Inserts are done at the end of the relation repre- sentation, that is, new tuples are appended to the relation. Since small deletes and updates do not occur in a data warehouse, large sequences of attribute values can be stored and compressed together as one unit, allowing for better compression than with small sequences.
3) **Cost of decompression**. Fetching data from a compressed representation requires *decompression*, which in the simplest compressed representations requires reading all the data from the beginning of a file. Transaction processing queries usually only need to fetch a few records; sequential access is expensive in such a scenario, since many irrelevant records may have to be decompressed to access a few relevant records.

- Since data analysis queries tend to access many consecutive records, the time spent on decompression is typically not wasted. However, even data analysis queries do not need to access records that fail selection conditions, and attributes of such records should be skipped to reduce disk I/O.
- To allow skipping of attribute values from such records, compressed representations for column stores allow decompression to start at any of a number of points in the file, skipping earlier parts of the file. This could be done by starting compression afresh after every 10,000 values (for example). By keeping track of where in the file the data start for each group of 10,000 values, it is possible to access the *i*th value by going to the start of the group [i / 1 0 0 0 0] and starting decompression from there.
- **ORC (Optimized Row Columnar)** and Parquet are columnar file representations used in many big-data processing applications. In ORC, a row-oriented representation is converted to column- oriented representation as follows: A sequence of tuples occupying several hundred megabytes is broken up into a columnar representation called a **stripe**. An ORC file contains several such stripes, with each stripe occupying around 250 megabytes.
- Figure illustrates some details of the ORC file format. Each stripe has index data followed by row data. The row data area stores a compressed representation of the sequence of value for the first column, followed by the compressed representation of the second column, and so on. The index data region of a stripe stores for each attribute the starting point within the stripe for each group of (say) 10,000 values of that attribute. The index is useful for quick access to a desired tuple or sequence of tuples; the index also allows queries containing selections to skip groups of tuples if the query determines that no tuple in those groups satisfies the selections. ORC files store several other pieces of information in the stripe footer and file footer, which we skip here.
- Some column-store systems allow groups of columns that are often accessed together to be stored together, instead of breaking up each column into a different file. Such systems thus allow a spectrum of choices that range from pure column-oriented storage, where every column is stored separately, to pure row-oriented storage, where all columns are stored together. The choice of which attributes to store together depends on the query workload.

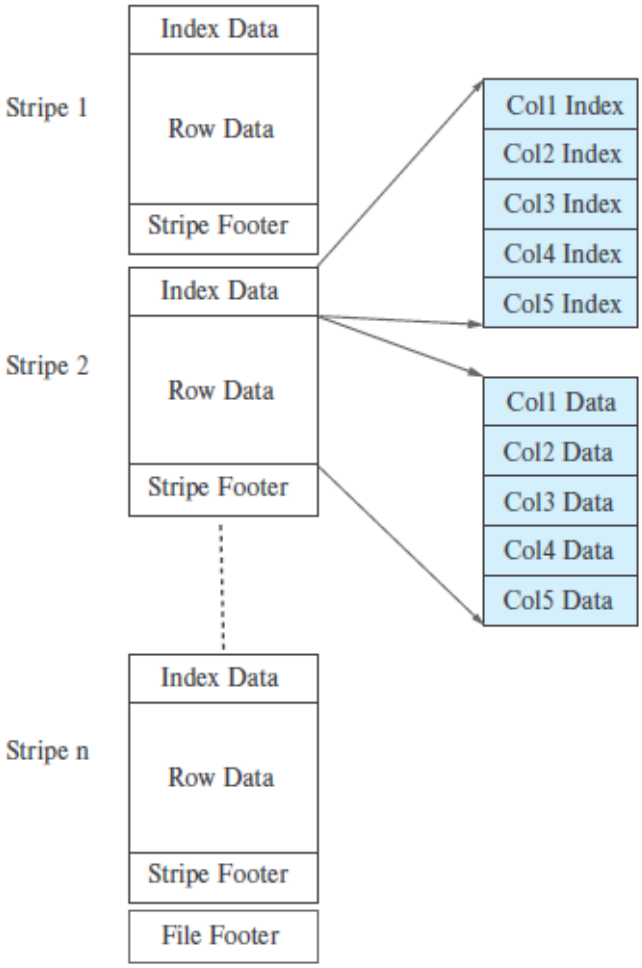


Figure. Columnar data representation in the ORC file format.

- Some of the benefits of column-oriented storage can be obtained even in a row- oriented storage system by logically decomposing a relation into multiple relations. For example, the *instructor* relation could be decomposed into three relations, containing (*ID*, *name*), (*ID*, *dept name*) and (*ID*, *salary*), respectively. Then, queries that access only the name do not have to fetch the *dept name* and *salary* attributes. However, in this case the same *ID* attribute occurs in three tuples, resulting in wasted space.
- Some database systems use a column-oriented representation for data within a disk block, without using compression.[8] Thus, a block contains data for a set of tuples, and all attributes for that set of tuples are stored in the same block. Such a scheme is useful in transaction-processing systems, since retrieving all attribute values does not require multiple disk accesses.
- At the same time, using column-oriented storage within the block provides the benefits of more efficient memory access and cache usage, as well as the potential for using vector processing on the data. However, this scheme does not allow irrelevant disk blocks to be skipped when only a few attributes are retrieved, nor does it give the benefits of compression. Thus, it represents a point in the space between pure row-oriented storage and pure column-oriented storage.
- Some databases, such as SAP HANA support two underlying storage systems, one a row-oriented one designed for transaction processing, and the second a column- oriented one,

designed for data analysis.

- Tuples are normally created in the row- oriented store but are later migrated to the column-oriented store when they are no longer likely to be accessed in a row-oriented manner. Such systems are called **hybrid row/column stores**.
- In other cases, applications store transactional data in a row-oriented store, but copy data periodically (e.g., once a day or a few times a day) to a data warehouse, which may use a column-oriented storage system.
- Sybase IQ was one of the early products to use column-oriented storage, but there are now several research projects and companies that have developed database systems based on column stores, including C-Store, Vertica, MonetDB, Vectorwise, among others.

6. Explain about indexing in detail. (or)
Describe the ordered indices with example.

- An index for a file in a database system works in much the same way as the index in the textbook.
- 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.

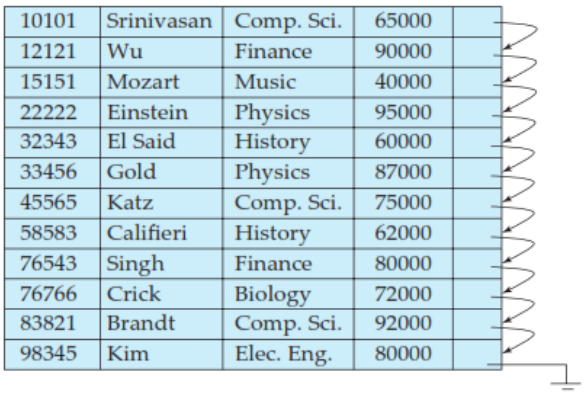
4) Deletion time: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.

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**.

ORDERED INDICES:

- 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.

DENSE AND SPARSE INDICES :

- 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.

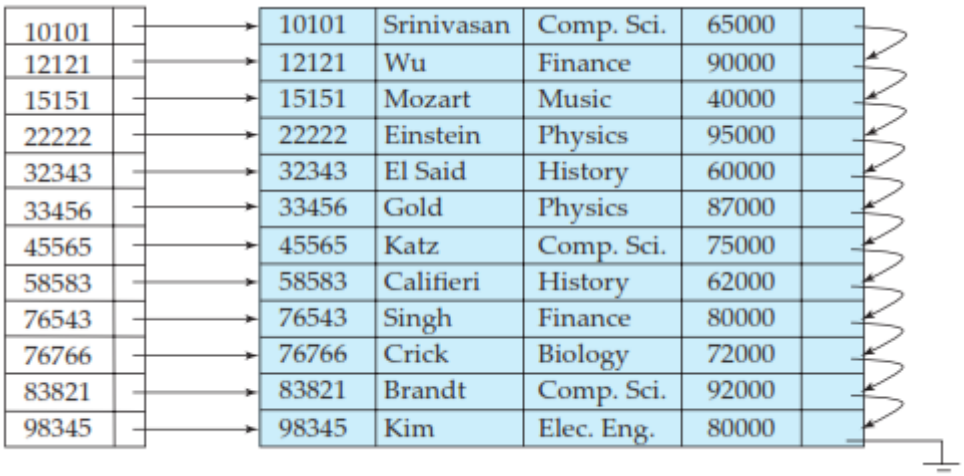


Figure. Dense index.

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.

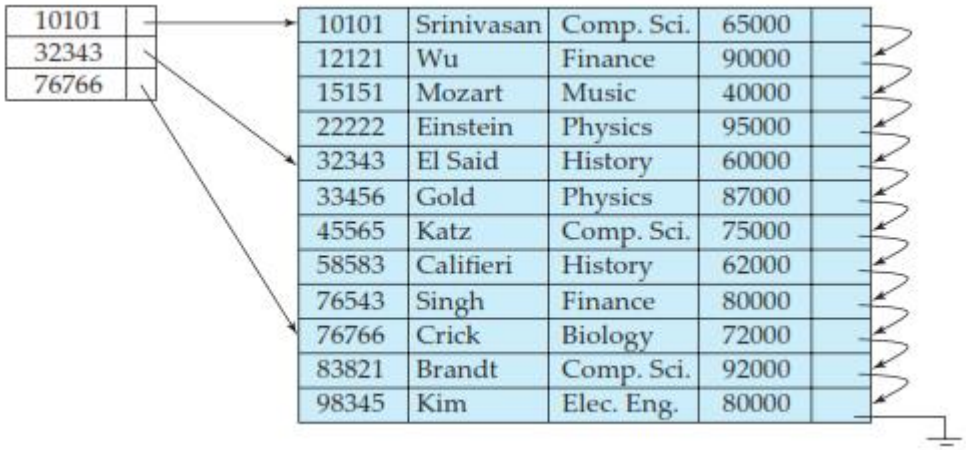


Figure. Sparse index.

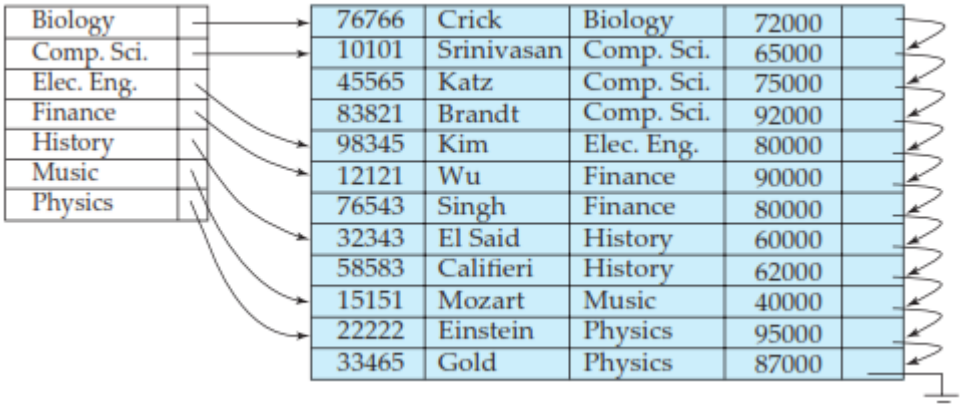


Figure. Dense index with search key *dept name*.

- 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.

- As another example, suppose that the search-key value is 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.

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.

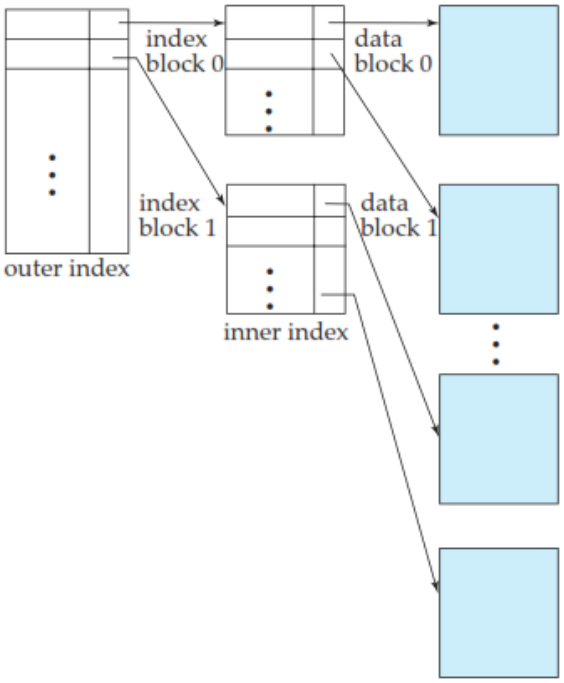


Figure. Two-level sparse index.

- 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.

Index Update:

- 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. Sūch 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:

Dense indices:

- 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.

Sparse indices:

- 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:

Dense indices:

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.

Sparse indices:

- 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. The same technique applies to further levels of the index, if there are any.

SECONDARY INDICES:

- 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. 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.
- 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.

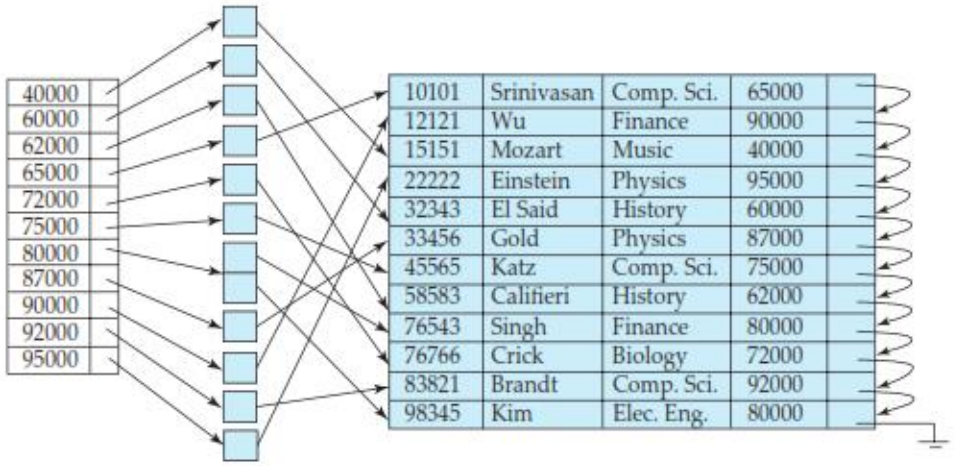


Figure . Secondary index on *instructor* file, on noncandidate key *salary*.

- 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.

Disadvantages:

- 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.
- 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.

7. Discuss in detail about B+ tree and B tree index files.

- 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.
- 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.

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.

- 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.

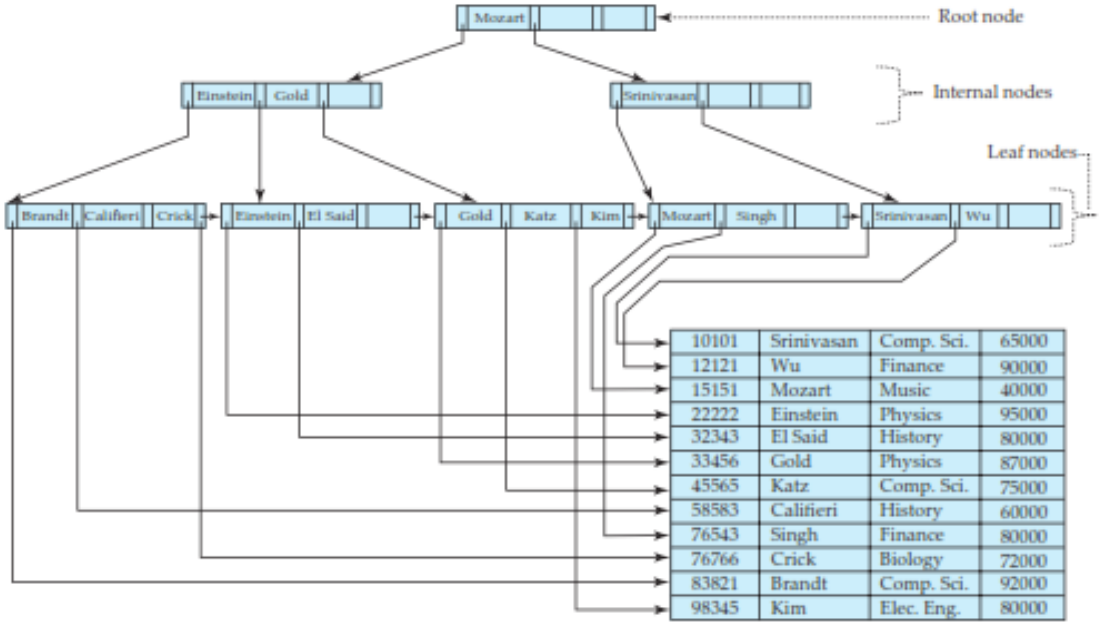


Figure . B^+ -tree for *instructor* file (n =4).

- 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.

Queries on B^+-Trees:

1) To find records with a search-key value

- 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.
- 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 non null 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.

2) Updates on B^+-Trees:

- When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly. 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, 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.

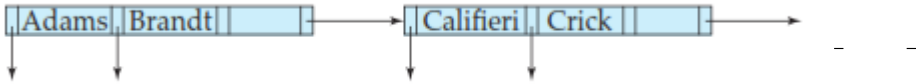


Figure. Split of leaf node on insertion of "Adams"

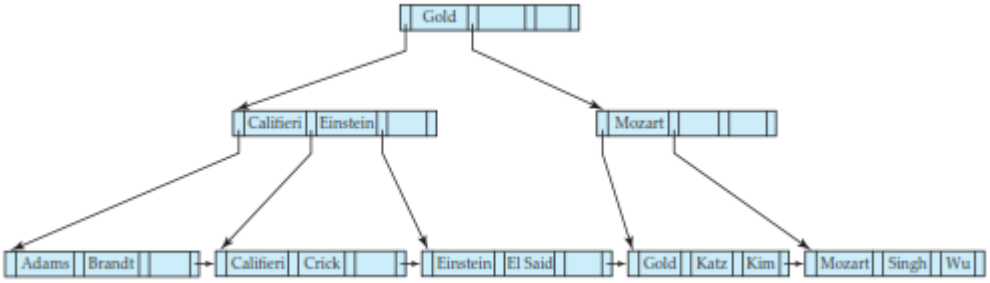


Figure. Deletion of "Srinivasan" from the B^+ -tree of Figure 11.13.

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.

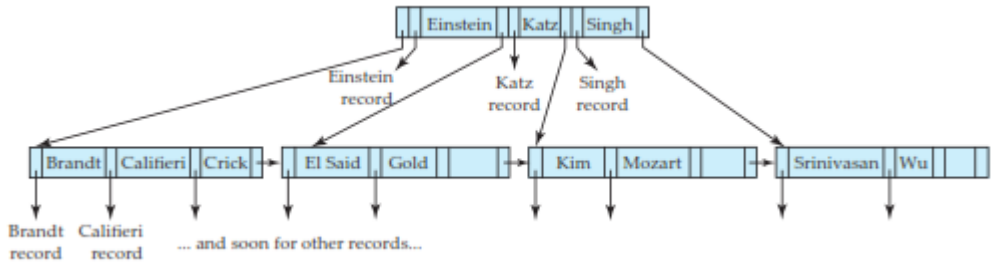


Figure. B-tree equivalent of B^+ -tree

- 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.

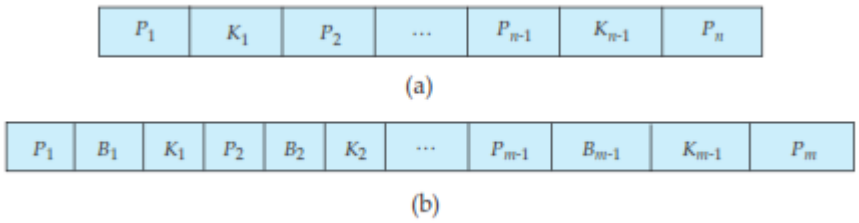


Figure . Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

- 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.

8. Explain about static hashing and dynamic hashing.

STATIC HASHING:

- 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.

1) 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.

2) In a **hash index organization** we organize the search keys, with their associated pointers, into a hash file structure.

Hash Functions:

- 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.

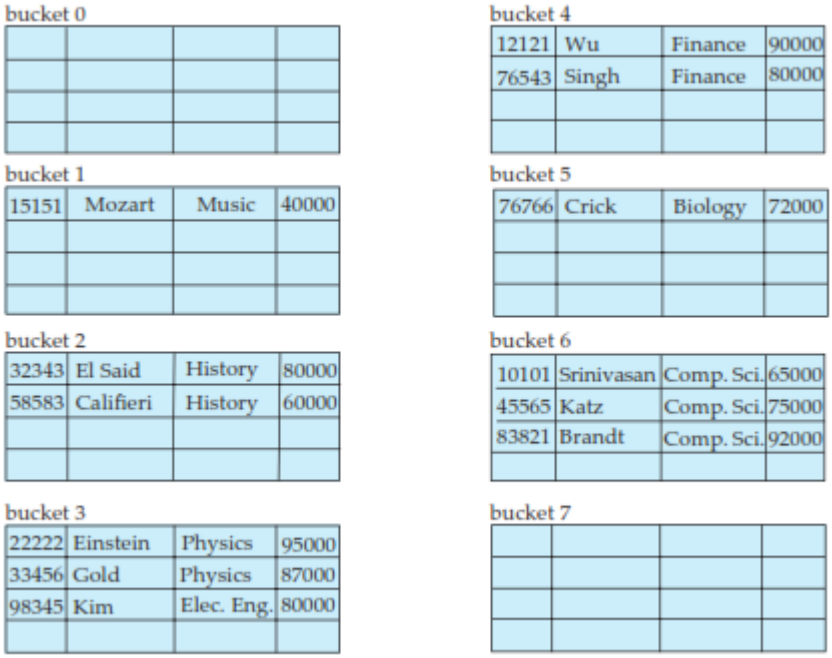


Figure. Hash organization of *instructor* file, with *dept name* as the key.

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:

$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$

- 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 non uniform 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**.

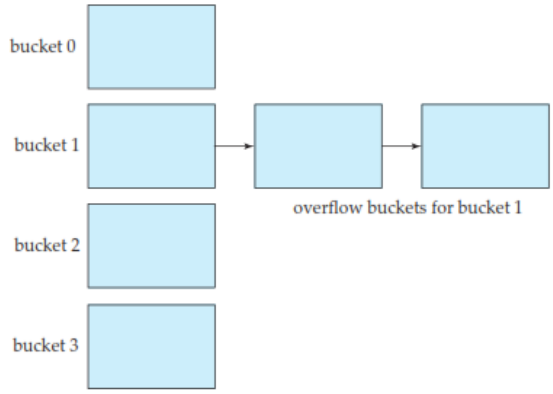


Figure . Overflow chaining in a hash structure.

- 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.
- 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.

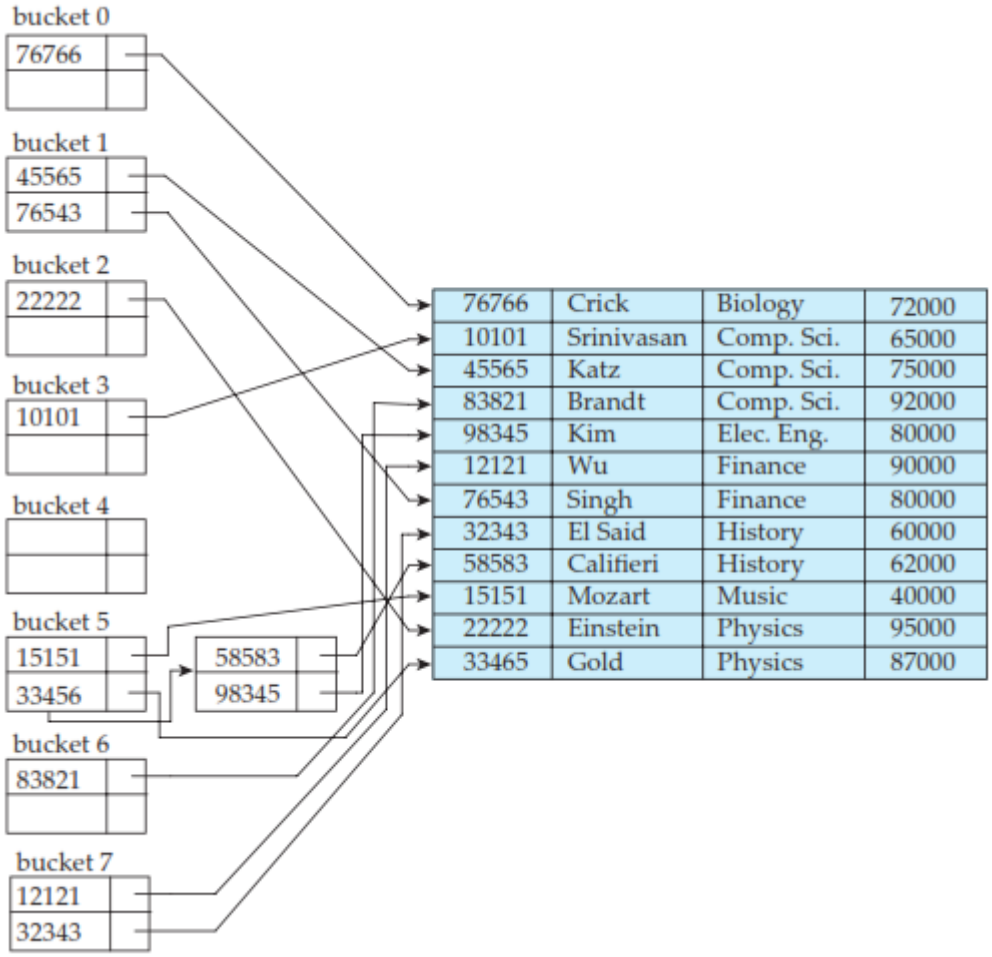


Figure. Hash index on search key *ID* of *instructor* file.

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.
- D**ynamic 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.

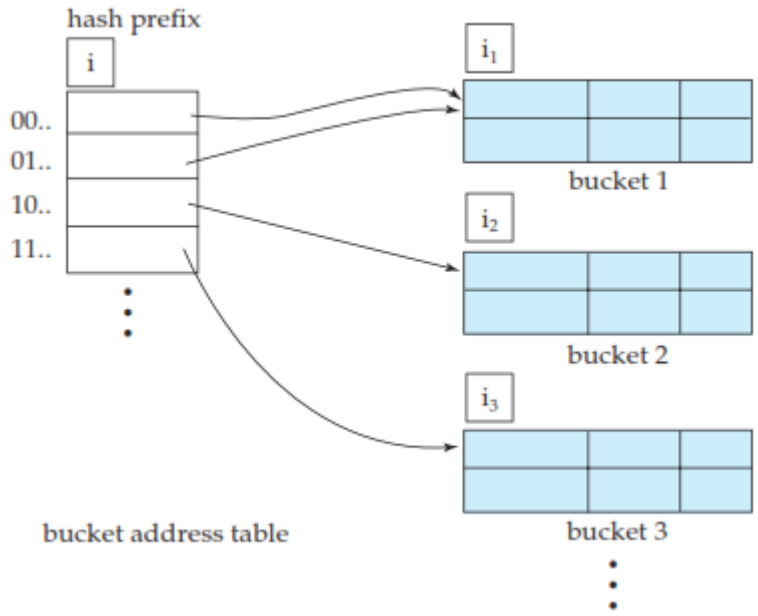


Figure. General extendable hash structure.

- 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.

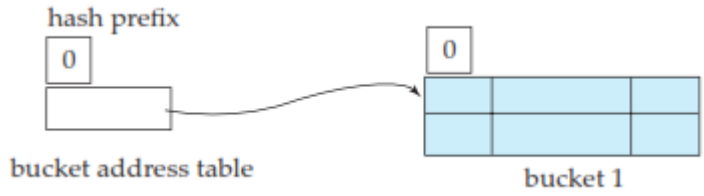


Figure. Initial extendable hash structure.

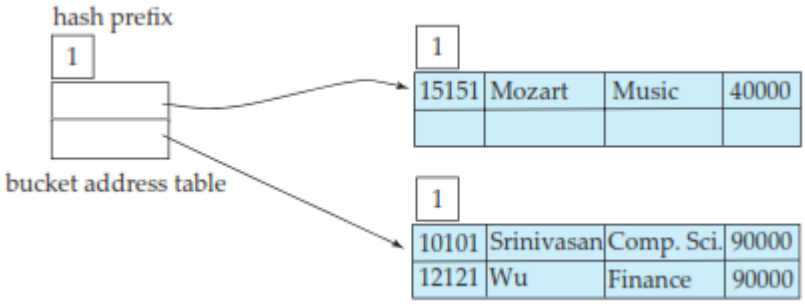


Figure. Hash structure after three insertions.

- 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 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 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.

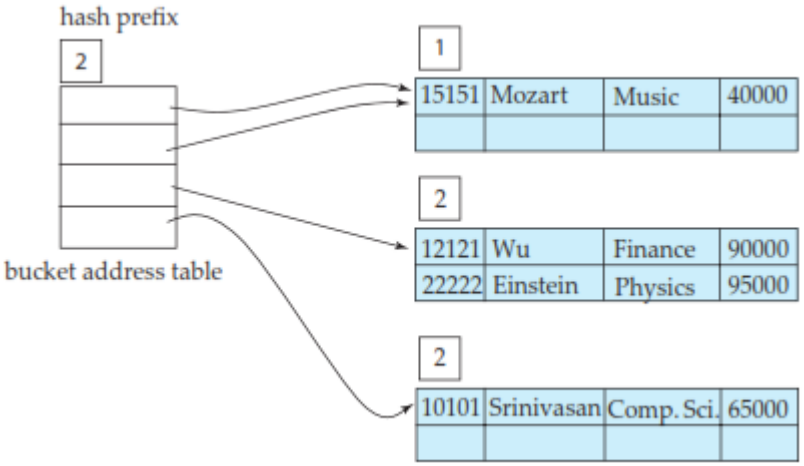


Figure . Hash structure after four insertions.

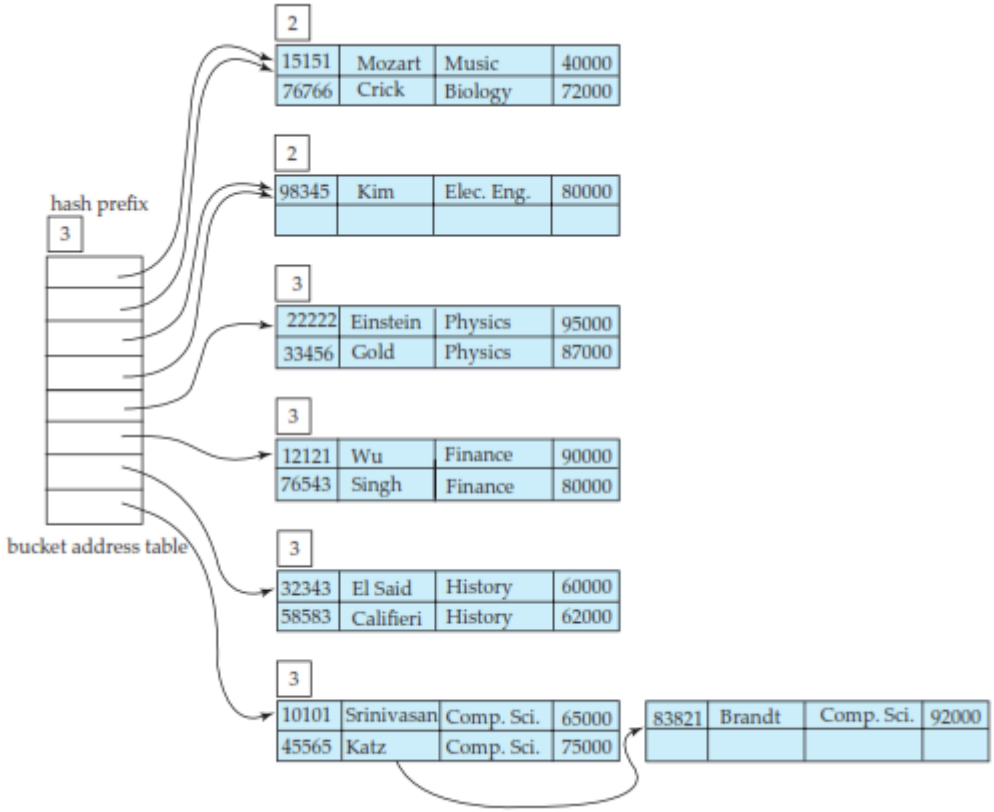


Figure. Extendable hash structure for the *instructor* file.

- 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.

Static Hashing versus Dynamic Hashing:

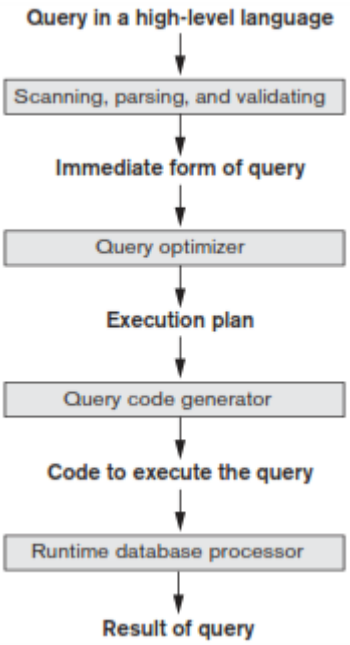
- We now examine the advantages and disadvantages of extendable hashing, com-pared with static hashing. The main advantage of extendable hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Although the bucket address table incurs additional overhead, 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 performance.
- Thus, extendable hashing appears to be a highly attractive technique, pro-vided that we are willing to accept the added complexity involved in its implementation. 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 associated with extendable hashing, at the possible cost of more overflow buckets.

9. Give a detailed description about Query Processing.

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.
- 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.



Code can be:
Executed directly (interpreted mode)
Stored and executed later whenever needed (compiled mode)
Figure . Typical steps when processing a high-level query.

10. Explain about algorithm for select and join operations.

- 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.

OP1: $\sigma_{Ssn = '123456789'}(EMPLOYEE)$
OP2: $\sigma_{Dnumber > 5}(DEPARTMENT)$
OP3: $\sigma_{Dno = 5}(EMPLOYEE)$
OP4: $\sigma_{Dno = 5 \text{ AND } Salary > 30000 \text{ AND } Sex = 'F'}(EMPLOYEE)$
OP5: $\sigma_{Essn='123456789' \text{ AND } Pno = 10}(WORKS\_ON)$

- 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.

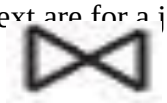
S9—Conjunctive selection by intersection of record pointers. If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if the indexes include record pointers (rather than block pointers), then each 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.

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 $\bowtie_{Dno=Dnumber}$ DEPARTMENT

OP7: DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ 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. 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 non key attributes, in which case the method needs to be modified slightly. A sketch of the sort-merge join algorithm is given in Figure (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. 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.

11. Explain about query optimization using heuristics rules.

- 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:

$$\pi_{Pnumber,\ Dnum,\ Lname,\ Address,\ Bdate}\ (((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT)) \bowtie_{Mgr\_ssn=Ssn}(EMPLOYEE))$$

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’;

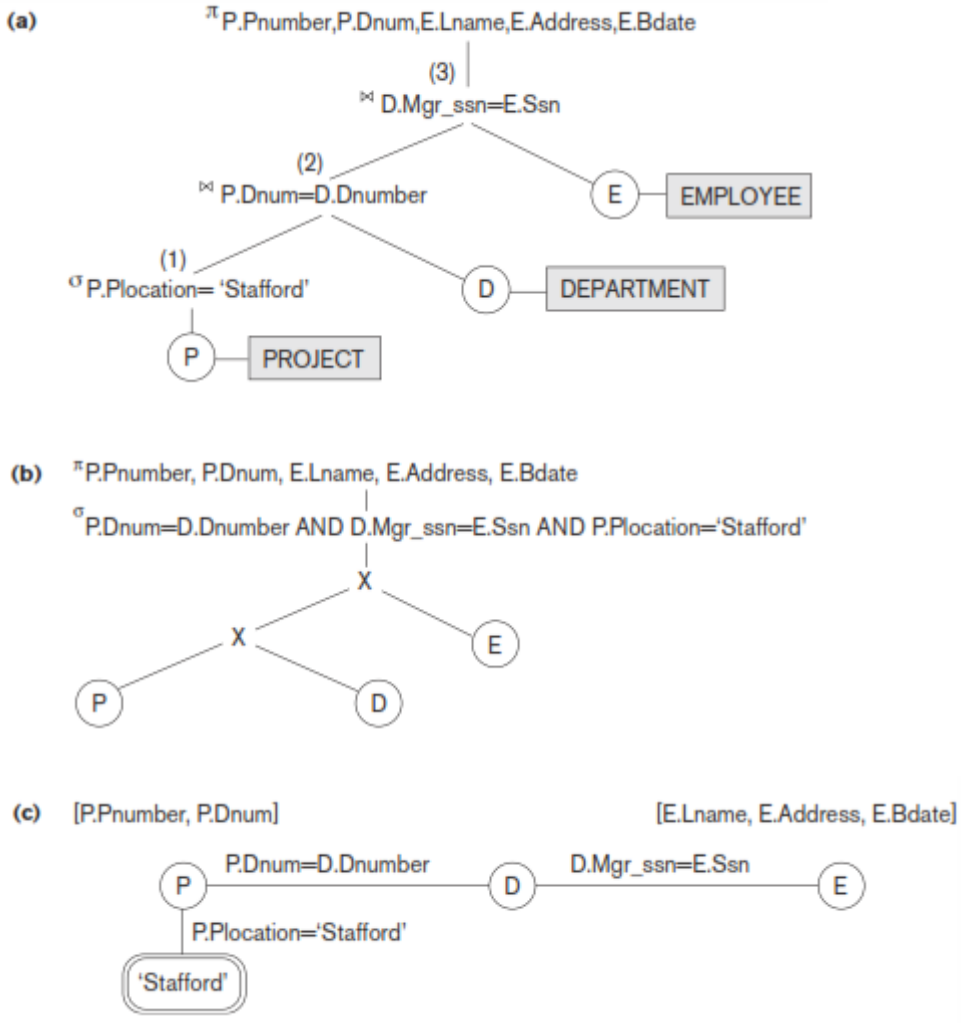


Figure. Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

- 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.

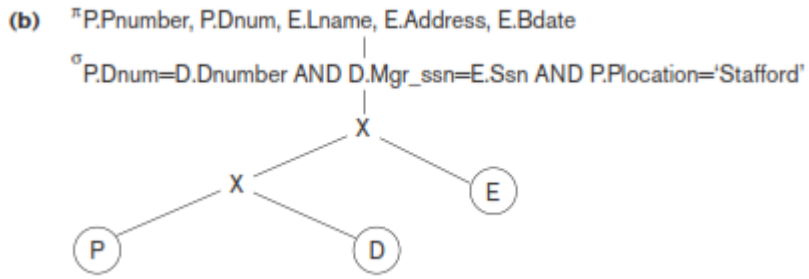
- As we can see, the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the **query graph** notation. Figure (c) shows the query graph for query Q2. Relations in the query are represented by **relation nodes**, which 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.



- 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. :

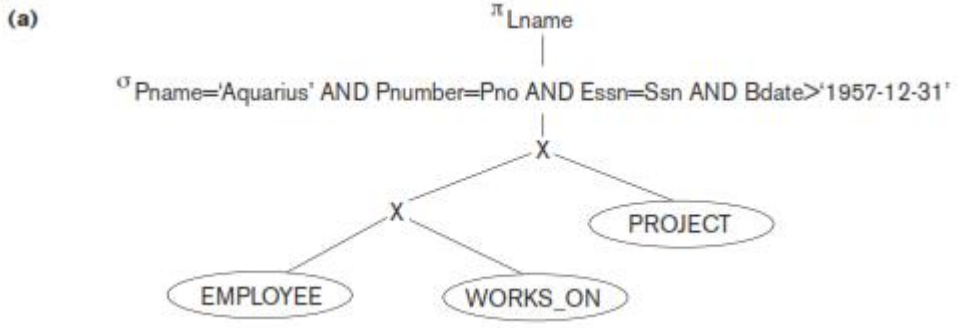
- Consider the following query Q: *Find the last names of employees born after 1957 who work on a project named 'Aquarius'*. This query can be specified in SQL as follows:

SELECT Lname FROM EMPLOYEE, WORKS_ON, PROJECT WHERE Pname='Aquarius'
AND Pnumber=Pno AND Essn=Ssn AND Bdate > '1957-12-31';

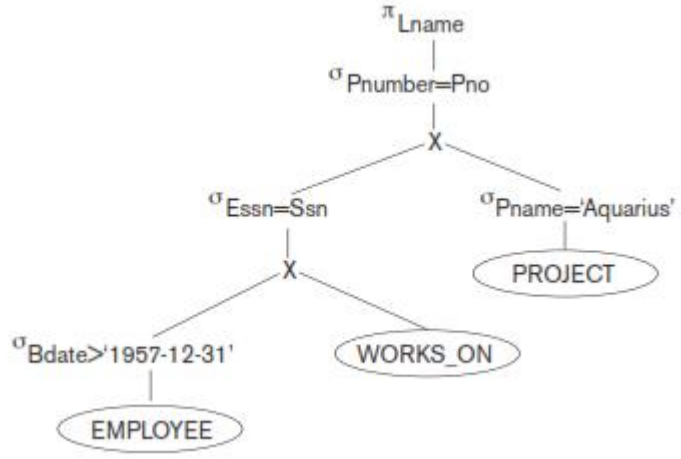
- The initial query tree for Q is shown in Figure(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree 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.

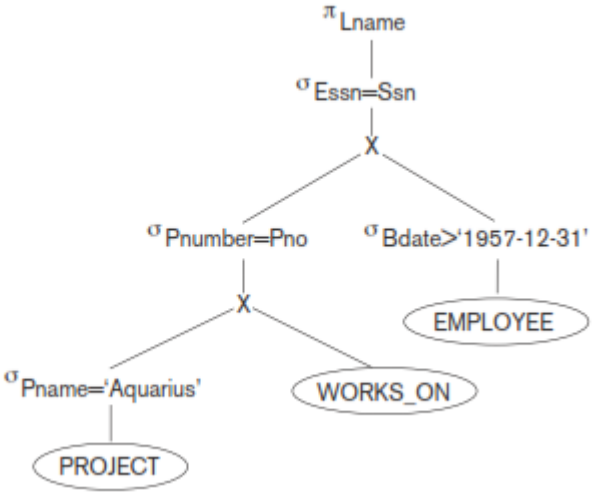
a) Initial (canonical) query tree for SQL query Q.
b) Moving SELECT operations down the query tree.
c) Applying the more restrictive SELECT operation first.
d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
e) Moving PROJECT operations down the query tree.



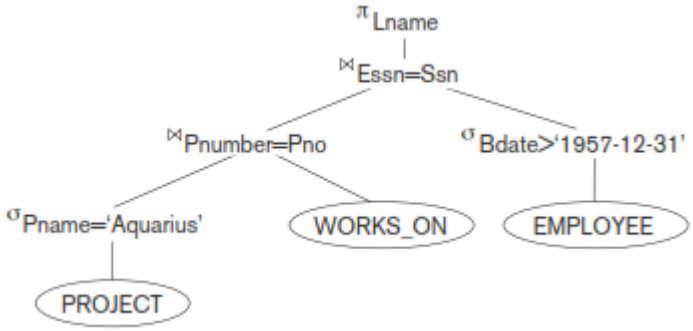
(b)


πLname
σPnumber=Pno
X
σEssn=Ssn
σPname='Aquarius'
X
PROJECT
σBdate>'1957-12-31'
WORKS_ON
EMPLOYEE

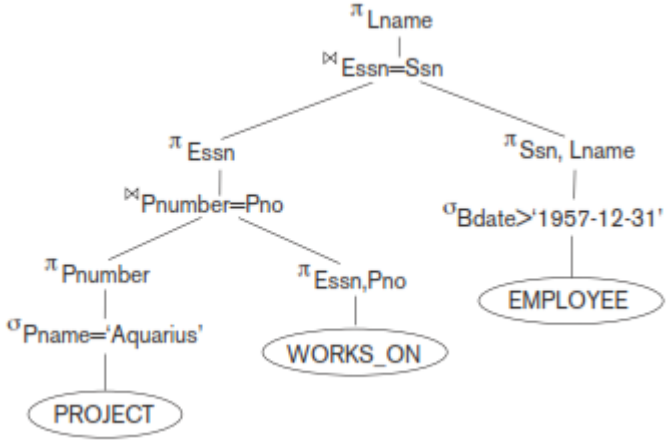
(c)


πLname
σEssn=Ssn
X
σPnumber=Pno
σBdate>'1957-12-31'
X
EMPLOYEE
σPname='Aquarius'
WORKS_ON
PROJECT

(d)


πLname
⋈Essn=Ssn
⋈Pnumber=Pno
σBdate>'1957-12-31'
σPname='Aquarius'
WORKS_ON
EMPLOYEE
PROJECT

(e)


πLname
⋈Essn=Ssn
πEssn
πSsn, Lname
⋈Pnumber=Pno
σBdate>'1957-12-31'
πPnumber
πEssn,Pno
EMPLOYEE
σPname='Aquarius'
WORKS_ON
PROJECT

- 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 operations 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).

- 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*.

- 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 $\bowtie$ (and $\times$).** The join operation is commutative, as is the $\times$ 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 $\bowtie$ (or $\times$).** 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 $(c_1 \text{ AND } c_2)$, 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 $\bowtie$ is replaced by a $\times$ operation.

7. **Commuting π with $\bowtie$ (or $\times$).** Suppose that the projection list is $L = \{A_1, ..., A_n, B_1, ..., B_m\}$, where $A_1, ..., A_n$ are attributes of R and $B_1, ..., B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

   $\pi_L (R \bowtie_c S) \equiv (\pi_{A_1, ..., A_n} (R)) \bowtie_c (\pi_{B_1, ..., B_m} (S))$

   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 $A_{n+1}, ..., A_{n+k}$ of R and $B_{m+1}, ..., B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

   $\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, ..., A_n, A_{n+1}, ..., A_{n+k}} (R)) \bowtie_c (\pi_{B_1, ..., B_m, B_{m+1}, ..., B_{m+p}} (S)))$

   For $\times$, there is no condition c, so the first transformation rule always applies by replacing $\bowtie_c$ with $\times$.

8. **Commutativity of set operations.** The set operations $\cup$ and $\cap$ 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:

    $(\sigma_c (R \times S)) \equiv (R \bowtie_c S)$

12. Explain about algorithm External sorting in DBMS.

- Sorting is one of the primary algorithms used in query processing. For example, whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted. Sorting is also a key component in sort-merge algorithms used for JOIN and other operations (such as UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause). We will discuss one of these algorithms in this sec- tion. Note that sorting of a particular file may be avoided if an appropriate index—such as a primary or clustering index—exists on the desired file attribute to allow ordered access to the records of the file.
- **External sorting** refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most data- base files.[6] The typical external sorting algorithm uses a **sort-merge strategy**, which starts by sorting small subfiles—called **runs**—of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn. The sort-merge algorithm, like other database algorithms, requires *buffer space* in main memory, where the actual sorting and merging of the runs is performed. The basic algorithm, outlined in Figure 18.2, consists of two phases: the sorting phase and the merging phase. The buffer space in main memory is part of the **DBMS cache**—an area in the computer's main memory that is controlled by the DBMS. The buffer space is divided into individual buffers, where each **buffer** is the same size in bytes as the size of one disk block. Thus, one buffer can hold the contents of exactly *one disk block.*
- In the **sorting phase**, runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an *internal* sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of each run and the **number of initial runs (n_R)** are dictated by the **number of file blocks (b)** and the **available buffer space (n_B)**. For example, if the number of available main memory buffers $n_B = 5$ disk blocks and the size of the file $b = 1{,}024$ disk blocks, then $n_R = \lceil (b/n_B) \rceil$ or 205 initial runs each of size 5 blocks (except the last run, which will have only 4 blocks). Hence, after the sorting phase, 205 sorted runs (or 205 sorted subfiles of the original file) are stored as temporary subfiles on disk.
- In the **merging phase**, the sorted runs are merged during one or more **merge passes**. Each merge pass can have one or more merge steps. The **degree of merging (d_M)** is the number of sorted subfiles that can be merged in each merge step. During each merge step, one buffer block is needed to hold one disk block from each of the sorted subfiles being merged, and one additional buffer is needed for containing one disk block of the merge result, which will produce a larger sorted file that is the result of merging several smaller sorted subfiles. Hence, d_M is the smaller of $(n_B - 1)$ and n_R, and the number of merge passes is $\lceil (\log_{dM}(n_R)) \rceil$. In our example, where $n_B = 5$, $d_M = 4$ (four-way merging), so the 205 initial sorted runs would be merged 4 at a time in each step into 52 larger sorted subfiles at the end of the first merge pass. These 52 sorted files are then merged 4 at a time into 13 sorted files, which are then merged into 4 sorted files, and then finally into 1 fully sorted file, which means that *four passes* are needed.

- Th

e performance of the sort-merge algorithm can be measured in terms of the number of disk block reads and writes (between the disk and main memory) before the sorting of

the whole file is completed. The following formula approximates this cost:

$$(2 * b) + (2 * b * (\log_{dM} n_R))$$

- The first term $(2 * b)$ represents the number of block accesses for the sorting phase, since each file block is accessed twice: once for reading into a main memory buffer and once for writing the sorted records back to disk into one of the sorted subfiles. The second term represents the number of block accesses for the merging phase. During each merge pass, a number of disk blocks approximately equal to the original file blocks b is read and written. Since the number of merge passes is $(\log_{dM} n_R)$, we get the total merge cost of $(2 * b * (\log_{dM} n_R))$.
- The minimum number of main memory buffers needed is $n_B = 3$, which gives a d_M of 2 and an n_R (of $b/3$. The minimum) d_M of 2 gives the worst-case performance of the algorithm, which is:

$$(2 * b) + (2 * (b * (\log_2 n_R))).$$

13. Explain about query optimization using cost estimation.

- 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.

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-IV

PART- A

1. What is RAID? (NOV 2014)(R)

A variety of disk organization techniques, collectively called redundant arrays of independent disks (RAID), have been proposed to achieve improvedperformance and reliability.

2. What are the factors to be taken into account in choosing a RAID level?

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 date in a failed diskare being rebuilt on a new disk)

3. What is heap file organization?

Any record can be placed anywhere in the file where there is space for therecord. There is no ordering of records. Typically, there is a single file for each relation.

4. What is sequential file organization?

Records are stored in sequential order, according to the value of a ―search key‖ of each record.

5. What is hashing file organization?

A hash function is computed on some attribute of each record. The result ofthe hash function specifies in which block of the file the record should be placed,

6. What is a multitable clustering file organization?

A multitable clustering file organization is a file organization thatstores 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.

7. What are the two basic kinds of indices?

The Two basic kinds of indices are

- Ordered indices
- Hash indices

8. What are access types?

The types of access that are supported efficiently. Access types can include finding records with a specified attribute vale and finding records whose attribute values fall in a specified range.

9. What is access time?

The time it takes to find a particular data item, or set of items, using thetechnique in question.

10.What is 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.

11.What is 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.

12.What is space overhead?

The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worth-while to sacrifice the space to achieve improved performance.

13.What is a search key?

An attribute or set of attributes used to look up records in a file is called a searchkey.

14.What is an clustering index (or) primary index? (NOV 2014)

Clustering index is an index whose search key also defines the sequential orderof the file.

15.What is an nonclustering index (or) secondary index?

Indices whose search key specifies an order different from the sequential order of the file are called nonclustering indices.

16.What is an index-sequential file?

All files are ordered sequentially on some search key. Such files, with a clustering index on the search key, are called index-sequential files.

17.What is a dense index? (MAY 2012)

An index record 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 thefirst record, since, because the index is a clustering one, records are stored on the same search key. Dense index implementations may store a list of pointers to all records with the same search-key value; doing so is not essential for clustering indices.

18.What is a sparse index? (MAY 2012)

An index record appears for only some of the search-key values. as is true in dense indices, each index record 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.

19.What is a multilevel indices?

Indices with two or more levels are called multilevel indices.

20.What is a balanced tree?

Balanced tree in which every path from the root of the tree to the leaf ofthe tree is of the same length.

21.Define hashing.

Hashing allow us to avoid accessing an index structure. It provides a way of constructing indices.

22.Define a bucket.

The term bucket to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosento be a smaller or larger than a disk block.

23.Define hash function.

A hash function h is a function from K to B. Let h denote a hash function

24.What are the two different purposes of hashing?

Hashing can be used for two different purposes.

- Hash file organization
- Hash index organization

25.What is a hash index?

A hash index organizes the search keys, with their associated pointers, into ahash file structure.

26.What is dynamic hashing? (NOV 2014)

Dynamic hashing techniques allow the hash function to be modifieddynamically to accommodate the growth or shrinkage of the database.

27.What are the steps involved in query processing?

The steps involved in processing a query are

- Parsing and translation
- Optimization
- Evaluation

28.What is an indexed nested-loop join?

Indexed nested loop join can be used with existing indices, as well as withtemporary indices created for the sole purpose of evaluating the join.

29.What is a merge join?

The merge-join algorithm can be used to compute natural joins and equi-joins.

30.What is an index record?

An index record, or index entry, consists of a search – key vale and pointers to one or more records with that value as their search-key value.

31.Define mean time to failure(MTTF)

Mean time to failure is a measure of the reliability of the disk. Themean time to failure of a disk is the amount of time that, on average, we can expect the system to run continuously without any failure.

32.What is mirroring?

The simplest approach to introducing redundancy is to duplicate every disk. This technique is called mirroring.

33.How dynamic hashing differ from statichashing? (DEC2015)

Static Hashing

- A bucket is a unit of storage containing one or more records (a bucket istypically a disk block).
- The file blocks are divided into M equal-sized buckets, numbered bucket0, bucket1... bucketM- 1.Typically, a bucket corresponds to one(or a fixed number of) disk block.
- In a hash file organization we obtain the bucket of a record directlyfrom its search-key value using a hash

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
- Hash function generates values over a large range —typically *b*-bit integers,with $b = 32$.
- At any time use only a prefix of the hash function to index into a table of bucketaddresses.
- Let the length of the prefix be *i* bits, $0 \_ i \_ 32$.

34.Disadvantages of B TREE over B+ TREE. (NOV 2016)

In a B tree search keys and data stored in internal or leaf nodes. But in B+-tree data store only leaf nodes. Searching any data in a B+ tree is very easy because all data are found in leafnodes. In a B tree, data cannot be found in leaf nodes.In a B tree, data may be found in leaf nodes or internal nodes. Deletion of internal nodes is very complicated. In a B+ tree, data is only found inleaf nodes. Deletion of leaf nodes is easy. Insertion in B tree is more complicated than B+ tree.B+ trees store redundant search key but B tree has no redundant value. In a B+ tree, leaf nodes data are ordered as a sequential linked list but in Btree the leaf node cannot be stored using a linked list. Many database systems' implementations prefer the structural simplicity of a B+ tree.

35. What is a query execution plan?**(Apr/May-2017)**

Execution plan will be generated by Query optimizer with the help of statisticsand Algebrizer processor tree.

It is the result of Query optimizer and tells how to do\perform yourwork\requirement.

There are two different execution plans - Estimated and Actual.

Estimated execution plan indicates optimizer view.

Actual execution plan indicates what executed the query and how was it done.

36. Which cost component are used most often as the basis for cost function**?(Apr/May-2017)**

(i) *Access cost to secondary storage*

(ii) *Memory usage of cost*

(iii)Storage cost

37.What is replication transparency?(Apr/May-2017)

Distribution Transparency. **Distribution transparency** is the property of **distributed databases** by the virtue of which the internal details of the **distribution** are hidden from the users. The DDBMS designer may choose to fragment tables, **replicate** the fragments and store themat different sites.

38.State the Need for Query Optimization.

- Process of selecting an efficient execution plan for evaluating the query.
- refers to the process of finding lowest cost method of evaluating a given query.
- Need for being logically consistent because the least cost plan will alwaysbe consistently low.

PART B

1. Explain various level of RAID in detail.
2. Discuss about File Organization with neat diagram
3. Explain about Organization of Records in Files
4. Elaborate about Data dictionary Storage.
5. Explain about Column Oriented Storage
6. Explain two types of ordered indices.
7. Discuss about B+ tree Index Files.
8. Elaborate about B tree Index Files
9. Explain about static hashing and Dynamic Hashing
10. Discuss about various steps in query processing
11. Explain about various Algorithms for Selection.
12. Elaborate about algorithms for Sortingand join operations
13. Discuss about Query optimization using Heuristics
14. Explain about Cost Estimation in query execution.

UNIT V ADVANCED TOPICS

Distributed Databases: Architecture, Data Storage, Transaction Processing, Query processing and optimization - NOSQL Databases: Introduction - CAP Theorem - Document Based systems - Key value Stores - Column Based Systems - Graph Databases. Database Security: Security issues - Access control based on privileges - Role Based access control - SQL Injection - Statistical Database security - Flow control - Encryption and Public Key infrastructures - Challenges

1. Explain about Distributed Database Concepts, characteristics and the advantages.

Distributed database (DDB) is a collection of multiple logically interrelated databases distributed over a computer network. D**istributed database management system (DDBMS)** is a software system that manages a distributed database while making the distribution transparent to the user.

For a database to be called distributed, the following minimum conditions should be satisfied:

- **Connection of database nodes over a computer network.** There are multiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **network** to transmit data and commands among sites.
- **Logical interrelation of the connected databases.** It is essential that the information in the various database nodes be logically related.
- **Possible absence of homogeneity among connected nodes.** It is not necessary that all nodes be identical in terms of data, hardware, and software.

The sites may all be located in physical proximity–say, within the same building or a group of adjacent buildings–and connected via a **local area network**, or they may be geographically distributed over large distances and connected via a **long-haul** or **wide area network**. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines, cables, wireless communication infrastructures, or satellites. It is common to have a combination of various types of networks.

- Networks may have different **topologies** that define the direct communication paths among sites. The type and topology of the network used may have a significant impact on the performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter what type of network is used; what matters is that each site be able to communicate, directly or indirectly, with every other site. We assume that some type of network exists among nodes, regardless of any particular topology.

Characteristics:

1) Transparency

The concept of transparency extends the general idea of hiding implementation details from end users. A highly transparent system offers a lot of flexibility to the end user/application developer since it requires little or no awareness of underlying details on their part. In the case of a traditional centralized database, transparency simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple nodes connected by a computer network, so additional types of transparencies are introduced.

The following types of transparencies are possible:

i) **Data organization transparency (also known as *distribution* or *network transparency*).** This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency. **Location transparency** refers to the fact that the command used to performa task is independent of the location of the data and the location of the node where the command was issued. **Naming transparency** implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.

ii) **Replication transparency. C**opies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.

iii) **Fragmentation transparency.** Two types of fragmentation are possible. **Horizontal fragmentation** distributes a relation (table) into subrelations that are subsets of the tuples (rows) in the original relation; this is also known as **sharding** in the newer big data and cloud computing systems. **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. Fragmentation transparency makes the user unaware of the existence of fragments.

iv) **Design transparency** and **execution transparency**–which refer, respectively, to freedom from knowing how the distributed database is designed and where a transaction executes.

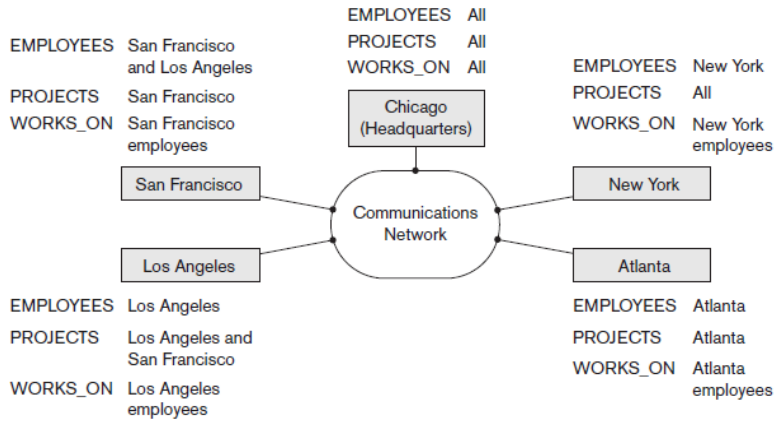


Figure. Data distribution and replication among distributed databases.

2) Availability and Reliability

- Reliability and availability are two of the most common potential advantages citedfor distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas **availability** is the probability that the system is continuously available during a time interval. We candirectly relate reliability and availability of the database to the faults, errors, and failures associated with it. A failure can be described as a deviation of a system's behavior from that which is specified in order to ensure correct execution of operations. **Errors** constitute that subset of system states that causes the failure. **Fault** is the cause of an error.
- To construct a system that is reliable, we can adopt several approaches. One common approach stresses *fault tolerance*; it recognizes that faults will occur, and it designs mechanisms that can detect and remove faults before they can result in a system failure. Another more stringent approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing. A reliable DDBMS tolerates failures of underlying components, and it processes user requests as long as data- base consistency is not violated. A DDBMS recovery manager has to deal with failures arising from transactions, hardware, and communication networks. Hardware failures can either be those that result in loss of main memory contents or loss of secondary storage contents. Network failures occur due to errors associated with messages and line failures. Message errors can include their loss, corruption, or out-of-order arrival at destination.

3) Scalability and Partition Tolerance

Scalability determines the extent to which the system can expand its capacity while continuing to operate without interruption. There are two types of scalability:

1. **Horizontal scalability:** This refers to expanding the number of nodes in the distributed system. As nodes are added to the system, it should be possible to distribute some of the data and processing loads from existing nodes to the new nodes.
2. **Vertical scalability:** This refers to expanding the capacity of the individualnodes in the system, such as expanding the storage capacity or the process- ing power of a node.

As the system expands its number of nodes, it is possible that the network, which connects the nodes, may have faults that cause the nodes to be partitioned into groups of nodes. The nodes within each partition are still connected by a subnetwork, but communication among the partitions is lost. The concept of **partition tolerance** states that the system should have the capacity to continue operating while the network is partitioned.

Autonomy

- **Autonomy** determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution. **Design autonomy** refers to independence of data model usage and transaction management techniques among nodes. **Communication autonomy** determines the extent to which each node can decide on sharing of information with other nodes. **Execution autonomy** refers to independence of users to act as they please.

Advantages of Distributed Databases

1. **Improved ease and flexibility of application development**. Developing and maintaining applications at geographically distributed sites of an organization is facilitated due to transparency of data distribution and control.
2. **Increased availability.** This is achieved by the isolation of faults to their site of origin without affecting the other database nodes connected to the network. When the data and DDBMS software are distributed over many sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. Further improvement is achieved by judiciously replicating data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site has been replicated at another site prior to the failure, then the user will not be affected at all. The ability of the system to survive network partitioning also contributes tohigh availability.
3. **Improved performance.** A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local data- bases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breakingup a query into a number of subqueries that execute in parallel. This contributes to improved performance.
4. **Easier expansion via scalability**. In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more nodes is much easier than in centralized (non-distributed) systems.

2. Discuss about Data Fragmentation, Replication, and Allocation Techniques for DistributedDatabase Design.

(or)

Explain about data storage in DistributedDatabase.

- Data Fragmentation techniques are used to break up the database into logical units, called **fragments**, which may be assigned for storage at the various nodes. **data replication** permits certain data to be stored in more than one site to increase availability and reliability; and the process of **allocating** fragments–or replicas of fragments–for storage at the various nodes. These techniques are used during the process of **distributed database design**. The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.

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.

- **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.

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 $\sigma_{Ci}(R)$ 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 $\pi_{Li}(R)$ 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.

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* ≠ ATTRS(*R*), we get a vertical fragment, and if *C* ≠ TRUE and *L* = ATTRS(*R*), we get a horizontal fragment. Finally, if

≠ TRUE and *L* ≠ 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.

Example:

customer_id	Name	Area	Payment Type	Gender
1	Bob	London	Credit card	Male
2	Mike	Manchester	Cash	Male
3	Ruby	London	Cash	Female

Horizontal Fragmentation are subsets of tuples (rows)

Fragment 1

customer_id	Name	Area	Payment Type	Gender
1	Bob	London	Credit card	Male
2	Mike	Manchester	Cash	Male

Fragment 2

customer_id	Name	Area	Payment Type	Gender
3	Ruby	London	Cash	Female

Vertical fragmentation are subset of attributes

Fragment 1

customer_id	Name	Area	Gender
1	Bob	London	Male
2	Mike	Manchester	Male
3	Ruby	London	Female

Fragment 2

customer_id	Payment Type
1	Credit card
2	Cash
3	Cash

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 **non redundant 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.

3. Explain about 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.
- An additional component called the **global transaction manager** is introduced for 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 , namely BEGIN_TRANSACTION, READ or WRITE, END_TRANSACTION,
- COMMIT_TRANSACTION, and ROLLBACK (or ABORT). The manager stores book- keeping 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 module the database operations 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 blocked 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

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. Other types of problems may also occur that make the outcome of the transaction nondeterministic.
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 prepared to commit 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[2] to guarantee mutually exclu sive 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 locked resource get blocked. 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 mostcommon 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 tohave this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

4. **Explain about Query Processing and Optimization in Distributed Databases.**

Various stages of distributed database query processing are:

1. **Query Mapping.** The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replication of data. Hence, this translation is largely identical to the one performedin a centralized DBMS. It is first normalized, analyzed for semantic errors, simplified, and finally restructured into an algebraic query.
2. **Localization.** In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.
3. **Global Query Optimization.** Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a network, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).
4. **Local Query Optimization.** This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

The first three stages discussed above are performed at a central control site, 3whereas the last stage is performed locally.

Data Transfer Costs of Distributed Query Processing

In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred tothe site where the query result is needed. Although these costs may not be very highif the sites are connected via a high-performance local area network, they become significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the *semijoin operation* is to reduce the number of tuples in a relation before transferring it to another site. Intuitively, the idea is to send the *joining column* of one relation *R* to the site where the other relation *S* is located; this column is then joined with *S*. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with *R*. Hence, only the joining column of *R* is transferred in one direction, and a subset of *S* with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in *S* participate in the join, this can be an efficient solution to minimizing data transfer.

Query and Update Decomposition

In a DDBMS with *no distribution transparency,* the user phrases a query directly in terms of specific fragments. The user must also maintain consistency of replicated data items when updating a DDBMS with *no replication transparency.*

On the other hand, a DDBMS that supports *full distribution, fragmentation,* and *replication transparency* allows the user to specify a query or update request onthe schema, just as though the DBMS were centralized. For updates, the DDBMS is responsible for maintaining *consistency among replicated items* by using one of the distributed concurrency control algorithms. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. Additionally, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard**, is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment; it is called a guard because *only tuples that satisfy this condition* are permitted to be stored in the fragment. For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

5. Explain about Distributed Database Architectures.

Distributed database (DDB) is a collection of multiple logically interrelated databases distributed over a computer network. D**istributed database management system (DDBMS)** is a software system that manages a distributed database while making the distribution transparent to the user.

- Figure 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.

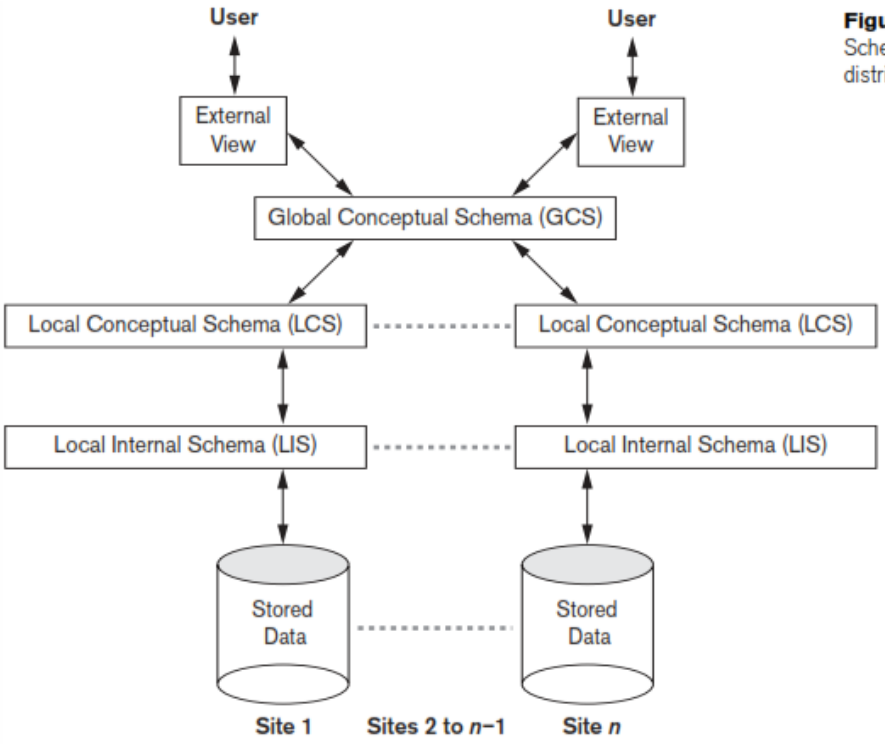


Figure 25.4
Schema architecture of distributed databases.

Federated Database Schema Architecture:

Typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure.

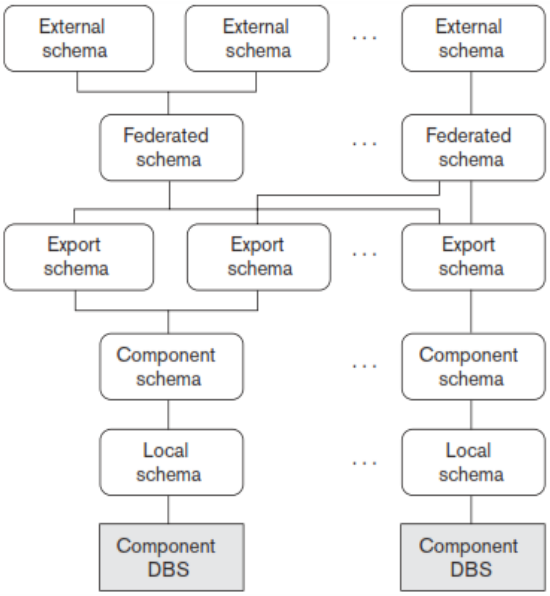


Figure . The five-level schema architecture in a federated database system (FDBS).

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.

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.

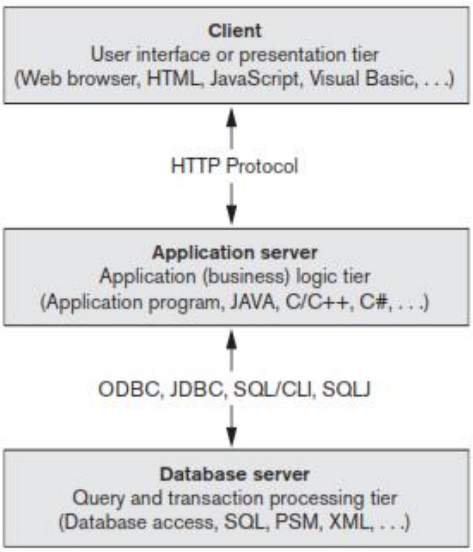


Figure . The three-tier client-server architecture.

- 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.

6. Explain about Emergence of NOSQL Systems, Characteristics of NOSQL Systems and types of NOSQL Systems.

- Many companies and organizations are faced with applications that store vast amounts of data. Consider a free e-mail application, such as Google Mail or Yahoo Mail or other similar service–this application can have millions of users, and each user can have thousands of e-mail messages. There is a need for a storage system that can manage all these e-mails; a structured relational SQL system may not be appropriate because

  (1) SQL systems offer too many services (powerful query language, concurrency control, etc.), which this application may not need;

  (2) A structured data model such the traditional relational model may be too restrictive. Although newer relational systems do have more complex object-relational modeling options, they still require schemas, which are not required by many of the NOSQL

systems.

- As another example**, consider an application such as Facebook**, with millions of users who submit posts, many with images and videos; then these posts must be displayed on pages of other users using the social media relationships among the users. User profiles, user relationships, and posts must all be stored in a huge collection of data stores, and the appropriate posts must be made available to the sets of users that have signed up to see these posts. Some of the data for this type of application is not suitable for a traditional relational system and typically needs multiple types of databases and data storage systems.
- Some of the organizations that were faced with these data management and storage applications decided to develop their own systems:

Google developed a proprietary NOSQL system known as BigTable, which is used in many of Google's applications that require vast amounts of data storage, such as Gmail, Google Maps, and Web site indexing.

Apache Hbase is an open source NOSQL system based on similar concepts. Google's innovation led to the category of NOSQL systems known as column-based or wide column stores; they are also sometimes referred to as column family stores.

Amazon developed a NOSQL system called DynamoDB that is available through Amazon's cloud services. This innovation led to the category known as key-value data stores or sometimes key-tuple or key-object data stores.

Facebook developed a NOSQL system called Cassandra, which is now open source and known as Apache Cassandra. This NOSQL system uses concepts from both key-value stores and column-based systems.

Other software companies started developing their own solutions and making them available to users who need these capabilities—for example, **MongoDB and CouchDB, which are classified as document-based NOSQL systems or document stores.**

Another category of NOSQL systems is the **graph-based NOSQL systems, or graph databases; these include Neo4J and GraphBase**, among others.

Some NOSQL systems, such as OrientDB, combine concepts from many of the categories discussed above.

In addition to the newer types of NOSQL systems listed above, it is also possible to classify database systems based on the object model or on the native XML model as NOSQL systems, althoughthey may not have the high-performance and replication characteristics of the other types of NOSQL systems.

These are just a few examples of NOSQL systems that have been developed.

Characteristics of NOSQL Systems

The characteristics are divided into two categories–those related to distributed databases and distributed systems, and those related to data models and query languages.

1) **NOSQL characteristics related to distributed databases and distributed systems.**

NOSQL systems emphasize high availability, so replicating the data is inherent in many of these systems. Scalability is another important characteristic, because many of the applications that use NOSQL systems tend to have data that keeps growing in volume. High performance is another required characteristic, whereas serializable consistency may not be as important for some of the NOSQL applications. We discuss some of these characteristics next.

1. **Scalability:** There are two kinds of scalability in distributed systems: horizontal and vertical. In NOSQL systems.

a) **Horizontal scalability** is generally used, where the distributed system is expanded by adding more nodes for data storage and processing as the volume of data grows.

b) **Vertical scalability** refers to expanding the storage and computing power of existing nodes.

In NOSQL systems, horizontal scalability is employed while the system is operational, so techniques for distributing the existing data among new nodes without interrupting system operation are necessary.

2. **Availability, Replication and Eventual Consistency:** Many applications that use NOSQL systems require continuous system availability. To accomplish this, data is replicated over two or more nodes in a transparent manner, so that if one node fails, the data is still available on other nodes. Replication improves data availability and can also improve read performance, because read requests can often be serviced from any of the replicated data nodes. However, write performance becomes more cumbersome because an update must be applied to every copy of the replicated data items; this can slow down write performance if serializable consistency is required Many NOSQL applications do not require serializable consistency, so more relaxed forms of consistency known as **eventual consistency** are used.

3. **Replication Models:** Two major replication models are used in NOSQL sytems: master-slave and master-master replication. **Master-slave replication** requires one copy to be the master copy; all write operations must be applied to the master copy and then propagated to the slave copies, usually using eventual consistency (the slave copies will *eventually* be the same as the master copy). For read, the master-slave paradigm can be configured in various ways. One configuration requires all reads to also be at the master copy, so this would be similar to the primary site or primary copy methods of distributed concurrency control, with similar advantages and disadvantages. Another configuration would allow reads at the slave copies but would not guarantee that the values are the latest writes, since writes to the slave nodes can be done after they are applied to the master copy. The **master-master replication** allows reads and writes at any of the replicas but may not guarantee that reads at nodes that store different copies see the same values. Different users may write the same data item concurrently at different nodes of the system, so the values of the item will be temporarily inconsistent. A reconciliation method to resolve conflicting write operations of the same data item at different nodes must be implemented as part of the master-master replication scheme.

4. **Sharding of Files:** In many NOSQL applications, files (or collections of data objects) can have many millions of records (or documents or objects), and these records can be accessed concurrently by thousands of users. So it is not practical to store the whole file in one node. **Sharding** (also known as **horizontal partitioning**) of the file records is often employed in NOSQL systems. This serves to distribute the load of accessing the file records to multiple nodes. The combination of sharding the file records and replicating the shards works in tandem to improve load balancing as well as data availability.

5. **High-Performance Data Access:** In many NOSQL applications, it is necessary to find individual records or objects (data items) from among the millions of data records or objects in a file. To achieve this, most systems use one of two techniques: hashing or range partitioning on object keys. The majority of accesses to an object will be by providing the key value rather than by using complex query conditions. The object key is similar to the concept of object id (see Section 12.1). In **hashing**, a hash function *h(K)* is applied to the key *K*, and the location of the object with key *K* is determined by the value of *h(K)*. In **range partitioning**, the location is determined via a range of key values; for example, location *i* would hold the objects whose key values K are in the range $Ki_{min} \leq K \leq Ki_{max}$. In applications that require range queries, where multiple objects within a range of key values are retrieved, range partitioned is preferred. Other indexes can also be used to locate objects based on attribute conditions different from the key K.

2) NOSQL characteristics related to data models and query languages.

NOSQL systems emphasize performance and flexibility over modeling power and complex querying.

1. **Not Requiring a Schema:** The flexibility of not requiring a schema is achieved in many NOSQL systems by allowing semi-structured, self- describing data. The users can specify a partial schema in some systems to improve storage efficiency, but it is *not required to have a schema* in most of the NOSQL systems. As there may not be a schema to specify constraints, any constraints on the data would have to be programmed in the application programs that access the data items. There are various languages for describing semistructured data, such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language;). JSON is used in several NOSQL systems, but other methods for describing semi-structured data can also be used.
2. **Less Powerful Query Languages:** Many applications that use NOSQL systems may not require a powerful query language such as SQL, because search (read) queries in these systems often locate single objects in a single file based on their object keys. NOSQL systems provide a set of functions and operations as a programming API (application programming interface), so reading and writing the data objects is accomplished by calling the appropriate operations by the programmer. In many cases, the operations are called **CRUD operations**, for Create, Read, Update, and Delete. In other cases, they are known as **SCRUD** because of an added Search (or Find) operation. Some NOSQL systems also provide a high-level query language, but it may not have the full power of SQL; only a subset of SQL querying capabilities would be provided. In particular, many NOSQL systems do not provide join operations as part of the query language itself; the joins need to be implemented in the application programs.
3. **Versioning:** Some NOSQL systems provide storage of multiple versions of the data items, with the timestamps of when the data version was created.

Categories of NOSQL Systems

NOSQL systems have been characterized into four major categories, with some additional categories that encompass other types of systems. The most common categorization lists the following four major categories:

1. **Document-based NOSQL systems:** These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation). Documents are accessible via their document id, but can also be accessed rapidly using other indexes.
2. **NOSQL key-value stores:** These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.
3. **Column-based or wide column NOSQL systems:** These systems partition a table by column into column families (a form of vertical partitioning), where each column family is stored in its own files. They also allow versioning of data values.
4. **Graph-based NOSQL systems:** Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions.

Additional categories can be added as follows to include some systems that are not easily categorized into the above four categories, as well as some other types of systems that have been available even before the term NOSQL became widely used.

5. **Hybrid NOSQL systems:** These systems have characteristics from two or more of the above four categories.
6. **Object databases:**
7. **XML databases:**

Even keyword-based search engines store large amounts of data with fast search access, so the stored data can be considered as large NOSQL big data stores.

7. Discuss about the CAP Theorem.

- The distributed database system (DDBS) is required to enforce the ACID properties (atomicity, consistency, isolation, durability) of transactions that are running concurrently. In a system with data replication, concurrency control becomes more complex because there can be multiple copies of each data item. So if an update is applied to one copy of an item, it must be applied to all other copies in a consistent manner. The possibility exists that one copy of an item X is updated by a transaction T_1 whereas another copy is updated by a transaction T_2, so two inconsistent copies of the same item exist at two different nodes in the distributed system. If two other transactions T_3 and T_4 want to read X, each may read a different copy of item X.

- There are distributed concurrency control methods that do not allow this inconsistency among copies of the same data item, thus enforcing serializability and hence the isolation property in the presence of replication. However, these techniques often come with high overhead, which would defeat the purpose of creating multiple copies to improve performance and availability in distributed database systems such as NOSQL. In the field of distributed systems, there are various levels of consistency among replicated data items, from weak consistency to strong consistency. Enforcing serializability is considered the strongest form of consistency, but it has high overhead so it can reduce performance of read and write operations and hence adversely affect system performance.

- The CAP theorem, which was originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication. The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **consistency** (among replicated copies), **availability** (of the system for read and write operations) and **partition tolerance** (in the face of the nodes in the system being partitioned by a network fault).

- ***Availability*** means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed.

- ***Partition tolerance*** means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.

- ***Consistency*** means that the nodes will have the same copies of a replicated data item visible for various transactions.

- It is important to note here that the use of the word *consistency* in CAP and its use in ACID *do not refer to the same identical concept*. In CAP, the term *consistency* refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema. However, if we consider that the consistency of replicated copies is a *specified constraint*, then the two uses of the term *consistency* would be related.

- **The CAP theorem states that it *is not possible to guarantee all three* of the desirable properties—consistency, availability, and partition tolerance—at the same time in a distributed system with data replication.** If this is the case, then the distributed system designer would have to choose two properties out of the three to guarantee. It is generally assumed that in many traditional (SQL) applications, guaranteeing consistency through the ACID properties is important.

- On the other hand, in a NOSQL distributed data store, a weaker consistency level is often acceptable, and guaranteeing the other two properties (availability, partition tolerance) is important. Hence, weaker consistency levels are often used in NOSQL system instead of guaranteeing serializability. In particular, a form of consistency known as **eventual consistency** is often adopted in NOSQL systems.

8. Explain about Document-Based NOSQL Systemsand MongoDB.

- Document-based or document-oriented NOSQL systems typically store data as **collections** of similar **documents**. These types of systems are also sometimes known as **document stores**. The individual documents somewhat resemble *complex objects* or XML documents, but a major difference between document-based systems versus object and object-relational systems and XML is that there is no requirement to specify a schema–rather, the documents are specified as **self-describing data**. Although the documents in a collection should be *similar*, they can have different data elements (attributes), and new documents can have new data elements that do not exist in any of the current documents in the collection. The system basically extracts the data element names from the self-describing documents in the collection, and the user can request that the system create indexes on some of the data elements. Documents can be specified in various formats, such as XML. A popular language to specify documents in NOSQL systems is **JSON** (JavaScript Object Notation).
- There are many document-based NOSQL systems, including MongoDB and CouchDB, among many others.

MongoDB Data Model

- MongoDB documents are stored in BSON (Binary JSON) format, which is a variation of JSON with some additional data types and is more efficient for storage than JSON. Individual **documents** are stored in a **collection**. The operation create Collection is used to create each collection. For example, the following command can be used to create a collection called **project** to hold PROJECTobjects from the COMPANY database.

  db.createCollection("project", { capped : true, size : 1310720, max : 500 })

- ***The first parameter "project" is the*** *name* ***of the collection, which is followed by an optional document that specifies*** *collection options****. In our example, the collectionis*** *capped****; this means it has upper limits on its storage space (****size****) and number of documents (****max****). The capping parameters help the system choose the storage options for each collection.***

- For our example, we will create another document collection called **worker** to hold information about the EMPLOYEEs who work on each project; for example:

  db.createCollection("worker", { capped : true, size : 5242880, max : 2000 }))

- Each document in a collection has a unique **ObjectId** field, called **_id**, which is automatically indexed in the collection unless the user explicitly requests no index for the _id field.

- The value of ObjectId can be *specified by the user*, or it can be *system-generated* if the user does not specify an _id field for a particular document.

- *System-generated* ObjectIds have a specific format, which combines the timestamp when the object is created (4 bytes, in an internal MongoDB format), the node id (3 bytes), the process id (2 bytes), and a counter (3 bytes) into a 16-byte Id value.

- *User-generated* ObjectsIds can have any value specified by the user as long as it

uniquely identifies the document and so these Ids are similar to primary keys in relational systems.

- A collection does not have a schema. The structure of the data fields in documents is chosen based on how documents will be accessed and used, and the user can choose a normalized design (similar to normalized relational tuples) or a denormalized design (similar to XML documents or complex objects).
- Interdocument references can be specified by storing in one document the ObjectId or ObjectIds ofother related documents.
- Figure (a) shows a simplified MongoDB document showing some of the data from the COMPANY database. In our example, the _id values are user-defined, and the documents whose _id starts with P (for project) will be stored in the "project" collection, whereas those whose _id starts with W (for worker) will be stored in the"worker" collection.
- In Figure (a), the workers information is *embedded in the project document*; sothere is no need for the "worker" collection. This is known as the ***denormalized pat tern***, which is similar to creating a complex object or an XML document . A list of values that is enclosed in *square brackets* […]within a document represents a field whose value is an **array**.
- Another option is to use the design in Figure (b), where *worker references* are embedded in the project document, but the worker documents themselves are stored in a separate "worker" collection.
- A third option in Figure (c) would use a normalized design, similar to First Normal Form relations. The choice of which design option to use depends on how the data will be accessed.
- It is important to note that the simple design in Figure (c) *is not the general normalized design* for a many-to-many relationship, such as the one between employees and projects; rather, we would need three collections for "project", "employee", and "works_on".
- In the design in Figure (c), an EMPLOYEE who works on several projects would be represented by *multiple worker documents* with different _id values; each document would represent the employee *as worker for a particular project*.
- This is similar tothe design decisions for XML schema design. However, it is again important to note that the typical document-based system *does not have a schema*, so the design rules would have to be followed whenever individual documents are inserted into a collection.

(a) project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                { Ename: "John Smith",
                  Hours: 32.5
                },
                { Ename: "Joyce English",
                  Hours: 20.0
                }
            ]
);
```

(b) project document with an embedded array of worker ids:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    WorkerIds:      [ "W1", "W2" ]
}
    { _id:          "W1",
    Ename:          "John Smith",
    Hours:          32.5
}
    { _id:          "W2",
    Ename:          "Joyce English",
    Hours:          20.0
}
```

(c) normalized project and worker documents (not a fully normalized design for M:N relationships):

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire"
}
{   _id:            "W1",
    Ename:          "John Smith",
    ProjectId:      "P1",
    Hours:          32.5
}
```

```
{   _id:        "W2",
    Ename:      "Joyce English",
    ProjectId:  "P1",
    Hours:      20.0
}
```

(d) inserting the documents in (c) into their collections "project" and "worker":

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```


```
{   _id:        "W2",
    Ename:      "Joyce English",
    ProjectId:  "P1",
    Hours:      20.0
}
(d) Inserting the documents in (c) into their collections "project" and "worker":
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```


Figure. Example of simpledocuments in MongoDB.
Denormalized document designwith embedded subdocuments.
Embedded array ofdocument references.
Normalizeddocuments.

MongoDB CRUD Operations

- MongoDb has several **CRUD operations**, where CRUD stands for (create, read, update, delete). Documents can be *created* and inserted into their collections using the **insert** operation, whose format is:

  db.<collection_name>.insert(<document(s)>)

- The parameters of the insert operation can include either a single document or an array of documents, as shown in Figure (d). The *delete* operation is called **remove**, and the format is:

  db.<collection_name>.remove(<condition>)

- The documents to be removed from the collection are specified by a Boolean condition on some of the fields in the collection documents. There is also an **update** operation, which has a condition to select certain documents, and a *$set* clause to specify the update. It is also possible to use the update operation to replace an existing document with another one but keep the same ObjectId.

- For *read* queries, the main command is called **find**, and the format is:

  db.<collection_name>.find(<condition>)

- General Boolean conditions can be specified as <condition>, and the documents inthe collection that return **true** are selected for the query result. For a full discussion of the MongoDb CRUD

operations, see the MongoDB online documentation in thechapter references.

MongoDB Distributed Systems Characteristics

- Most MongoDB updates are atomic if they refer to a single document, but MongoDB also provides a pattern for specifying transactions on multiple documents. Since MongoDB is a distributed system, the **two-phase commit** method is used to ensure atomicity and consistency of multidocument transactions.
- **Replication in MongoDB**. The concept of **replica set** is used in MongoDB to create multiple copies of the same data set on different nodes in the distributed system, and it uses a variation of the **master-slave** approach for replication. For example, suppose that we want to replicate a particular document collection C. A replica set will have one **primary copy** of the collection C stored in one node N1, and at least one **secondary copy** (replica) of C stored at another node N2. Additional copies can be stored in nodes N3, N4, etc., as needed, but the cost of storage and update (write) increases with the number of replicas. The total number of participants in a replica set must be at least three, so if only one secondary copy is needed, a participant in the replica set known as an **arbiter** must run on the third node N3. The arbiter does not hold a replica of the collection but participates in **elections** to choose a new primary if the node storing the current primary copy fails. If the total number of members in a rep- lica set is n (one primary plus i secondaries, for a total of $n = i + 1$), then n must be an odd number; if it is not, an *arbiter* is added to ensure the election process works correctly if the primary fails.
- In MongoDB replication, all write operations must be applied to the primary copy and then propagated to the secondaries. For read operations, the user can choose the particular **read preference** for their application. The *default read preference* processes all reads at the primary copy, so all read and write operations are performed at the primary node. In this case, secondary copies are mainly to make surethat the system continues operation if the primary fails, and MongoDB can ensure that every read request gets the latest document value. To increase read performance, it is possible to set the read preference so that *read requests can be processedat any replica* (primary or secondary); however, a read at a secondary is not guaranteed to get the latest version of a document because there can be a delay in propagating writes from the primary to the secondaries.
- **Sharding in MongoDB**. When a collection holds a very large number of documents or requires a large storage space, storing all the documents in one node can lead to performance problems, particularly if there are many user operationsaccessing the documents concurrently using various CRUD operations. **Sharding** of the documents in the collection—also known as *horizontal partitioning*— divides the documents into disjoint partitions known as **shards**. This allows the system to add more nodes as needed by a process known as **horizontal scaling** of the distributed system, and to store the shards of the collection on different nodes to achieve load balancing. Each node will process only those operations pertaining to the documents in the shard stored at that node. Also, each shard will contain fewer documents than if the entire collection were stored at one node, thus further improving performance.
- There are two ways to partition a collection into shards in MongoDB—**range partitioning** and **hash partitioning**. Both require that the user specify a particular document field to be used as the basis for partitioning the documents into shards. The *partitioning field*—known as the **shard key** in MongoDB—must have two characteristics: it must exist in *every document* in the collection, and it must have an *index*. The ObjectId can be used, but any other field possessing these two character- istics can also be used as the basis for sharding. The values of the shard key are divided into **chunks** either through range partitioning or hash partitioning, and the documents are partitioned based on the chunks of shard key values.
- Range partitioning creates the chunks by specifying a range of key values; for example, if

the shard key values ranged from one to ten million, it is possible to create ten ranges—1 to 1,000,000; 1,000,001 to 2,000,000; … ; 9,000,001 to 10,000,000—and each chunk would contain the key values in one range. Hash partitioning applies a hash function h(K) to each shard key K, and the partitioning of keys into chunks is based on the hash values. In general, if **range queries** are commonly applied to a collection (for example, retrieving all documents whose shard key value is between 200 and 400), then range partitioning is preferred because each range query will typically be submit- ted to a single node that contains all the required documents in one shard. If most searches retrieve one document at a time, hash partitioning may be preferable because it randomizes the distribution of shard key values into chunks.

- When sharding is used, MongoDB queries are submitted to a module called the **query router**, which keeps track of which nodes contain which shards based on the particular partitioning method used on the shard keys. The query (CRUD operation) will be routed to the nodes that contain the shards that hold the documents that the query is requesting. If the system cannot determine which shards hold the required documents, the query will be submitted to all the nodes that hold shards of the collection. Sharding and replication are used together; sharding focuses on improving performance via load balancing and horizontal scalability, whereas replication focuses on ensuring system availability when certain nodes fail in the distributed system.
- There are many additional details about the distributed system architecture and components of MongoDB, but a full discussion is outside the scope of our presentation. MongoDB also provides many other services in areas such as system administration, indexing, security, and data aggregation, but we will not discuss these features here. Full documentation of MongoDB is available online.

9. Elaborate about NOSQL Key-Value Stores

- **Key-value stores** focus on high performance, availability, and scalability by storing data in a distributed storage system. The data model used in key-value stores is relatively simple, and in many of these systems, there is no query language but rather a set of operations that can be used by the application programmers.
- The **key** is a unique identifier associated with a data item and is used to locate this data item rapidly.
- The **value** is the data item itself, and it can have very different formats for different key-value storage systems. In some cases, the value is just a *string of bytes* or an *array of bytes*, and the application using the key-value store has to interpret the structure of the data value. In other cases, some standard formatted data is allowed; for example, structured data rows (tuples) similar to relational data, or semistructured data using JSON or some other self-describing data format. Different key-value stores can thus store unstructured, semistructured, or structured data items. The main characteristic of key-value stores is the fact that every value (data item) must be associated with a unique key, and that retrieving the value by supplying the key must be very fast.

DynamoDB Overview

- The DynamoDB system is an Amazon product and is available as part of Amazon's **AWS/SDK** platforms (Amazon Web Services/Software Development Kit). It can be used as part of Amazon's cloud computing services, for the data storage component.
- DynamoDB data model. The basic data model in DynamoDB uses the concepts of tables, items, and attributes. A **table** in DynamoDB *does not have* a **schema**; it holds a collection of *self-describing items*. Each **item** will consist of a number of (attribute, value) pairs, and

attribute values can be single-valued or multivalued. So basically, a table will hold a collection of items, and each item is a self-describing record (or object). DynamoDB also allows the user to specify the items in JSON for- mat, and the system will convert them to the internal storage format of DynamoDB.

- When a table is created, it is required to specify a **table name** and a **primary key**; the primary key will be used to rapidly locate the items in the table. Thus, the primary key is the **key** and the item is the **value** for the DynamoDB key-value store. The primary key attribute must exist in every item in the table. The primary key can be one of the following two types:

- **A single attribute.** The DynamoDB system will use this attribute to build a hash index on the items in the table. This is called a *hash type primary key*. The items are not ordered in storage on the value of the hash attribute.
- **A pair of attributes.** This is called a *hash and range type primary key*. The primary key will be a pair of attributes (A, B): attribute A will be used for hashing, and because there will be multiple items with the same value of A, the B values will be used for ordering the records with the same A value. A table with this type of key can have additional secondary indexes defined on its attributes. For example, if we want to store multiple versions of some type of items in a table, we could use ItemID as hash and Date or Timestamp (when the version was created) as range in a hash and range type primary key.

- DynamoDB Distributed Characteristics. Because DynamoDB is proprietary, in the next subsection we will discuss the mechanisms used for replication, sharding, and other distributed system concepts in an open source key-value system called Voldemort. Voldemort is based on many of the techniques proposed for DynamoDB.

Voldemort Key-Value Distributed Data Store

- Voldemort is an open source system available through Apache 2.0 open source licensing rules. It is based on Amazon's DynamoDB. The focus is on high performance and horizontal scalability, as well as on providing replication for high availability and sharding for improving latency (response time) of read and write requests. All three of those features—replication, sharding, and horizontal scalability—are realized through a technique to distribute the key-value pairs among the nodes of a distributed cluster; this distribution is known as **consistent hashing**. Voldemort has been used by LinkedIn for data storage. Some of the features of Voldemort are as follows:

- **Simple basic operations.** A collection of (key, value) pairs is kept in a Voldemort **store**. In our discussion, we will assume the store is called *s*. The basic interface for data storage and retrieval is very simple and includes three operations: get, put, and delete. The operation s.put(k, v) inserts an item as a key-value pair with key k and value v. The operation s.delete(k) deletes the item whose key is k from the store, and the operation v = s.get(k) retrieves the value v associated with key k. The application can use these basic operations to build its own requirements. At the basic storage level, both keys and values are arrays of bytes (strings).
- **High-level formatted data values.** The values v in the (k, v) items can be specified in JSON (JavaScript Object Notation), and the system will convert between JSON and the internal storage format. Other data object formats can also be specified if the application provides the conversion (also known as **serialization**) between the user format and the storage format as a *Serializer class*. The Serializer class must be provided by the user and will include oper- ations to convert the user format into a string of bytes for storage as a value, and to convert back a string (array of bytes) retrieved via *s.get(k)* into the user format. Voldemort has some built-in serializers for formats other than JSON.
- **Consistent hashing for distributing (key, value) pairs.** A variation of the data distribution algorithm

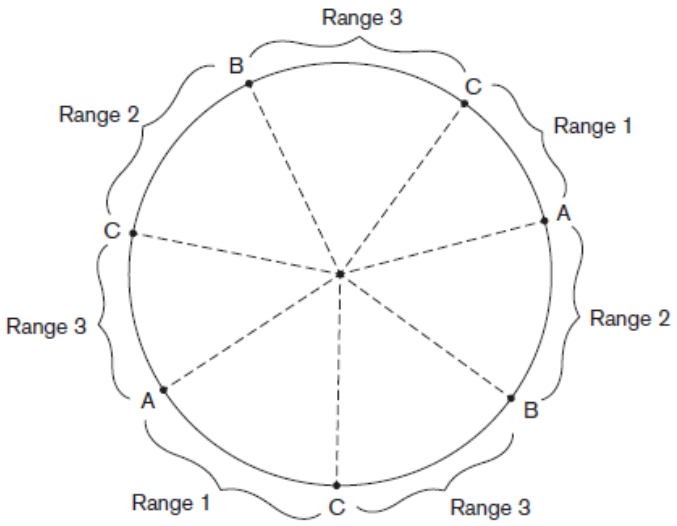
known as **consistent hashing** is used in Volde- mort for data distribution among the nodes in the distributed cluster of nodes. A hash function $h(k)$ is applied to the key k of each (k, v) pair, and $h(k)$ determines where the item will be stored. The method assumes that $h(k)$ is an integer value, usually in the range *0* to $Hmax = 2^{n-1}$, where *n* is chosen based on the desired range for the hash values. This method is best visualized by considering the range of all possible integer hash values *0* to *Hmax* to be evenly distributed on a circle (or ring). The nodes in the distributed system are then also located on the same ring; usually each node willhave several locations on the ring. The positioning of the points on the ring that represent the nodes is done in a pseudo random manner.

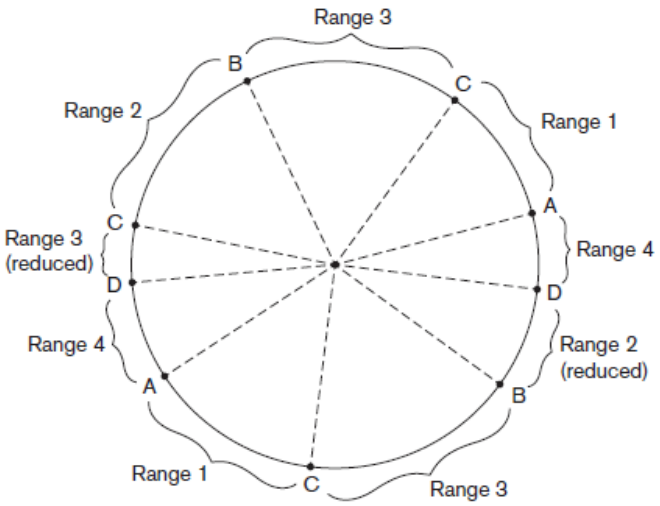
An item (*k, v*) will be stored on the node whose position in the ring *follows* the position of $h(k)$ on the ring *in a clockwise direction*. In Figure (a), weassume there are three nodes in the distributed cluster labeled A, B, and C, where node C has a bigger capacity than nodes A and B. In a typical system,there will be many more nodes. On the circle, two instances each of A and Bare placed, and three instances of C (because of its higher capacity), in apseudorandom manner to cover the circle. Figure (a) indicates which (k, v) items are placed in which nodes based on the *h(k)* values.

Figure . Example of consistent hashing. (a) Ring having three nodes A, B, and C, with C havinggreater capacity. The *h*(*K*) values that map tothe circle points in *range 1* have their (k, v)items stored in node A,*range 2* in node B, *range 3* in node C.

(b) Adding a node D tothe ring. Items in

range 4 are moved to the node D from nodeB (*range 2* is reduced)and node C (*range 3* isreduced).





- The *h*(*k*) values that fall in the parts of the circle marked as *range 1* in Fig- ure (a) will have their (k, v) items stored in node A because that is the node whose label follows *h*(*k*) on the ring in a clockwise direction; those in *range 2* are stored in node B; and those in *range 3* are stored in node C. This scheme allows *horizontal scalability* because when a new node is added to the distributed system, it can be added in one or more locations on the ring depending on the node capacity. Only a limited percentage of the (k, v) items will be reas- signed to the new node from the existing nodes based on the consistent hash- ing placement algorithm. Also, those items assigned to the new node may not all come from only one of the existing nodes because the new node can have multiple locations on the ring. For example, if a node D is added and it has two placements on the ring as shown in Figure (b), then some of the items from nodes B and C would be moved to node D. The items whose keys hash to *range 4* on the circle would be migrated to node D. This scheme also allows *replication* by placing the number of specified replicas of an item on successive nodes on the ring in a clockwise direction. The *sharding* is built into the method, and different items in the store (file) are located on different nodes in the distributed cluster, which means the items are horizon- tally partitioned (sharded) among the nodes in the distributed system. When a node fails, its load of data items can be distributed to the other existing nodes whose labels follow the labels of the failed node in the ring. And nodes with higher capacity can have more locations on the ring, as illustrated by node C in Figure 24.2(a), and thus store more items than smaller-capacity nodes.
- **Consistency and versioning.** Voldemort uses a method similar to the one developed for DynamoDB for consistency in the presence of replicas. Basically, concurrent write operations are allowed by different processes so there could exist two or more different values associated with the same key at dif- ferent nodes when items are replicated. Consistency is achieved when the item is read by using a technique known as *versioning and read repair*. Con- current writes are allowed, but each write is associated with a *vector clock* value. When a read occurs, it is possible that different versions of the same value (associated with the same key) are read from different nodes. If the system can reconcile to a single final value, it will pass that value to the read; otherwise, more than one version can be passed back to the application, which will reconcile the various versions into one version based on the application semantics and give this reconciled value back to the nodes.

Examples of Other Key-Value Stores

Three other key-value stores are

- Oracle key-value store. Oracle has one of the well-known SQL relational database systems, and Oracle also offers a system based on the key-value store concept; this system is called the **Oracle NoSQL Database**.
- Redis key-value cache and store. **Redis** differs from the other systems dis- cussed here because it caches its data in main memory to further improve perfor- mance. It offers master-slave replication and high availability, and it also offers persistence by backing up the cache to disk.
- Apache Cassandra. **Cassandra** is a NOSQL system that is not easily categorized into one category; it is sometimes listed in the column-based NOSQL category or in the key-value category. If offers features from several NOSQL categories and is used by Facebook as well as many other customers.

10. Explain about Column-Based or Wide Column NOSQL Systems.

- Another category of NOSQL systems is known as **column-based** or **wide column** systems. The Google distributed storage system for big data, known as **BigTable**, is a well-known example of this class of NOSQL systems, and it is used in many Google applications that require large amounts of data storage, such as Gmail. Big- Table uses the **Google File System** (**GFS**) for data storage and distribution. An open source system known as **Apache Hbase** is somewhat similar to Google Big- Table, but it typically uses **HDFS** (**Hadoop Distributed File System**) for data storage. HDFS is used in many cloud computing applications. Hbase can also use Amazon's **Simple Storage System** (known as **S3**) for data storage. Another well-known example of column-based NOSQL systems is Cassandra because it can also be characterized as a key-value store.
- BigTable (and Hbase) is sometimes described as a *sparse multidimensional distributed persistent sorted map*, where the word *map* means a *collection of (key, value) pairs* (the key is *mapped* to the value). One of the main differences that distinguish column-based systems from key-value stores is the *nature of the key*. In column-based systems such as Hbase, the key is *multidimensional* and so has several components: typically, a combination of table name, row key, column, and timestamp. As we shall see, the column is typically composed of two compo- nents: column family and column qualifier. We discuss these concepts in more detail next as they are realized in Apache Hbase.

Hbase Data Model and Versioning

- Hbase data model. The data model in Hbase organizes data using the concepts of *namespaces*, *tables*, *column families*, *column qualifiers*, *columns*, *rows*, and *data cells*. A column is identified by a combination of (column family:column qualifier). Data is stored in a self-describing form by associating columns with data values, where data values are strings. Hbase also stores *multiple versions* of a data item, with a *timestamp* associated with each version, so versions and timestamps are also part of the Hbase data model (this is similar to the concept of attribute versioning intemporal databases.
- As with other NOSQL systems, unique keys are associated with stored data items for fast access, but the keys identify *cells* in the storage system. Because the focus is on high performance when storing huge amounts of data, the data model includes some storage-related concepts.

- **Tables and Rows.** Data in Hbase is stored in **tables**, and each table name. Data in a table is stored as self-describing **rows**. Each row has a unique **row key**, and row keys are strings that must have the property that they can be lexicographically ordered, so characters that do not have a lexi-cographic order in the character set cannot be used as part of a row key.
- **Column Families, Column Qualifiers, and Columns.** A table is associatedwith one or more **column families**. Each column family will have a name, and the column families associated with a table *must be specified* when the table is created and cannot be changed later. Figure (a) shows how a table may be created; the table name is followed by the names of the column families associated with the table. When the data is loaded into a table, each column family can be associated with many **column qualifiers**, but the column qualifiers *are not specified* as part of creating a table. So the column qualifiers make the model a self-describing data model because the qualifiers can be dynamically specified as new rows are created and inserted into the table. A **column** is specified by a combination of ColumnFamily:ColumnQualifier. Basically, column families are a way of grouping together related columns (attributes in relational terminology) for storage purposes, except that the column qualifier names are not specified during table creation. Rather, theyare specified when the data is created and stored in rows, so the data is *self-describing* since any column qualifier name can be used in a new row of data. However, it is important that the application programmers know which column qualifiers belong to each column family, even though they have the flexibility to create new column qualifiers on the fly when new data rows are created. The concept of column family is somewhat similar to *vertical partitioning*, because columns (attributes) that are accessed together because they belong to the same column family are stored in the same files. Each column family of a table is stored in its own files using the HDFS file system.

Versions and Timestamps. Hbase can keep several versions of a data item, along with the timestamp associated with each version. The timestamp is along integer number that represents the system time when the version was created, so newer versions have larger timestamp values. Hbase uses mid- night 'January 1, 1970 UTC' as timestamp value zero, and uses a long integer that measures the number of milliseconds since that time as the system timestamp value (this is similar to the value returned by the Java utility java.util.Date.getTime() and is also used in MongoDB). It is also possible for the user to define the timestamp value explicitly in a Date format rather thanusing the system-generated timestamp.

Figure. Examples in Hbase.

(a) Creating a table called EMPLOYEE with three column families: Name, Address, and Details.

(b) Inserting some in the EMPLOYEE table; different rows can have different self-describing column qualifiers (Fname, Lname, Nickname, Mname, Minit, Suffix, … for column family Name; Job, Review, Supervisor, Salary for column family Details).

(c) Some CRUD operations of Hbase.

(a) creating a table:

```
create 'EMPLOYEE', 'Name', 'Address', 'Details'
```

(b) inserting some row data in the EMPLOYEE table:

```
put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'
```

(c) Some Hbase basic CRUD operations:

```
Creating a table: create <tablename>, <column family>, <column family>, ...
Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
Reading Data (all data in a table): scan <tablename>
Retrieve Data (one item): get <tablename>,<rowid>
```

- **Cells.** A **cell** holds a basic data item in Hbase. The key (address) of a cell is specified by a combination of (table, rowid, columnfamily, columnqualifier, timestamp). If timestamp is left out, the latest version of the item is retrievedunless a default number of versions is specified, say the latest three versions.The default number of versions to be retrieved, as well as the default numberof versions that the system needs to keep, are parameters that can be specified during table creation.

Namespaces. A **namespace** is a collection of tables. A namespace basicallyspecifies a collection of one or more tables that are typically used together by user applications, and it corresponds to a database that contains a collectionof tables in relational terminology.

Hbase CRUD Operations

- Hbase has low-level CRUD (create, read, update, delete) operations, as in many of the NOSQL systems. The formats of some of the basic CRUD operations in Hbaseare shown in Figure (c).
- Hbase only provides low-level CRUD operations. It is the responsibility of the application programs to implement more complex operations, such as joins between rows in different tables. The *create* operation creates a new table and specifies one or more column families associated with that table, but it does not specify the column qualifiers, as we discussed earlier. The *put* operation is used for inserting new data or new versions of existing data items. The get operation is for retrieving the data associated with a single row

in a table, and the *scan* operation retrievesall the rows.

Hbase Storage and Distributed System Concepts

- Each Hbase table is divided into a number of **regions**, where each region will hold a *range* of the row keys in the table; this is why the row keys must be lexicographically ordered. Each region will have a number of **stores**, where each column family is assigned to one store within the region. Regions are assigned to **region servers** (storage nodes) for storage. A **master server** (master node) is responsible for monitoring the region servers and for splitting a table into regions and assigning regionsto region servers.
- Hbase uses the **Apache Zookeeper** open source system for services related to man-aging the naming, distribution, and synchronization of the Hbase data on the distributed Hbase server nodes, as well as for coordination and replication services. Hbase also uses Apache HDFS (Hadoop Distributed File System) for distributed file services. So Hbase is built on top of both HDFS and Zookeeper. Zookeeper can itself have several replicas on several nodes for availability, and it keeps the data it needs in main memory to speed access to the master servers and region servers.

11. Discuss about NOSQL Graph Databases and Neo4j.

In **graph databases** or **graph- oriented NOSQL** systems, the data is represented as a graph, which is a collection of vertices (nodes) and edges. Both nodes and edges can be labeled to indicate the types of entities and relationships they represent, and it is generally possible to store data associated with both individual nodes and individual edges. Many systems can be categorized as graph databases. We will focus our discussion on one particular system, Neo4j, which is used in many applications. Neo4j is an open source system, and it is implemented in Java.

Neo4j Data Model

- The data model in Neo4j organizes data using the concepts of **nodes** and **relation-ships**. Both nodes and relationships can have **properties**, which store the data items associated with nodes and relationships. Nodes can have **labels**; the nodes that have the *same label* are grouped into a collection that identifies a subset of the nodes in the database graph for querying purposes. A node can have zero, one, or several labels. Relationships are directed; each relationship has a *start node* and *end node* as well as a **relationship type**, which serves a similar role to a node label by identifying similar relationships that have the same relationship type. Properties can be specified via a **map pattern**, which is made of one or more “name : value” pairs enclosed in curly brackets; for example {Lname : ‘Smith’, Fname : ‘John’, Minit : ‘B’}.
- In conventional graph theory, nodes and relationships are generally called *vertices* and *edges*. The Neo4j graph data model somewhat resembles how data is repre- sented in the ER and EER models, but with some notable differences. Comparing the Neo4j graph model with ER/EER concepts, nodes cor- respond to *entities*, node labels correspond to *entity types and subclasses*, relation- ships correspond to *relationship instances*, relationship types correspond to *relationship types*, and properties correspond to *attributes*. One notable difference is that a relationship is *directed* in Neo4j, but is not in ER/EER. Another is that a node may have no label in Neo4j, which is not allowed in ER/EER because every entity must belong to an entity type. A third crucial difference is that the graph model of Neo4j is used as a basis for an actual high-performance distributed data- base system whereas the ER/EER model is mainly used for database design.

- Figure (a) shows how a few nodes can be created in Neo4j. There are various ways in which nodes and relationships can be created; for example, by calling appro- priate Neo4j operations from various Neo4j APIs. We will just show the high-level syntax for creating nodes and relationships; to do so, we will use the Neo4j CREATE command, which is part of the high-level declarative query language Cypher. Neo4j has many options and variations for creating nodes and relationships using various scripting interfaces, but a full discussion is outside the scope of our presentation.

Labels and properties. When a node is created, the node label can be specified. It is also possible to create nodes without any labels. In Figure (a), the node labels are EMPLOYEE, DEPARTMENT, PROJECT, and LOCATION, and the created nodes correspond to some of the data from the COMPANY database in Figure 5.6 with a few modifications; for example, we use EmpId instead of SSN, and we only include a small subset of the data for illustration purposes. Properties are enclosed in curly brackets { … }. It is possible that some nodes have multiple labels; for example the same node can be labeled as PERSON and EMPLOYEE and MANAGER by listing all the labels separated by the colon symbol as follows: PERSON:EMPLOYEE:MANAGER. Having multiple labels is similar to an entity belonging to an entity type (PERSON)

plus some subclasses of PERSON (namely EMPLOYEE and MANAGER) in the EER model but can also be used for other purposes.

Relationships and relationship types. Figure (b) shows a few example relationships in Neo4j based on the COMPANY database in Figure 5.6. The □ specifies the direction of the relationship, but the relationship can be traversed in either direction. The relationship types (labels) in Figure (b) are WorksFor, Manager, LocatedIn, and WorksOn; only relationships with the relationship type WorksOn have properties (Hours) in Figure (b).

Paths. A **path** specifies a traversal of part of the graph. It is typically used as part of a query to specify a pattern, where the query will retrieve from the graph data that matches the pattern. A path is typically specified by a start node, followed by one or more relationships, leading to one or more end nodes that satisfy the pattern.

Optional Schema. A **schema** is optional in Neo4j. Graphs can be created and used without a schema, but in Neo4j version 2.0, a few schema-related functions were added. The main features related to schema creation involve creating indexes and constraints based on the labels and properties. For example, it is possible to create the equivalent of a key constraint on a property of a label, so all nodes in the collection of nodes associated with the label must have unique values for that property.

Indexing and node identifiers. When a node is created, the Neo4j system creates an internal unique system-defined identifier for each node. To retrieve individual nodes using other properties of the nodes efficiently, the user can create **indexes** for the collection of nodes that have a particular label. Typically, one or more of the properties of the nodes in that collection can be indexed. For example, Empid can be used to index nodes with the EMPLOYEE label, Dno to index the nodes with the DEPARTMENT label, and Pno to index the nodes with the PROJECT label.

The Cypher Query Language of Neo4j

- Neo4j has a high-level query language, Cypher. There are declarative commands for creating nodes and relationships, as well as for finding nodes and relationships based on specifying patterns. Deletion and modification of data is also possible in Cypher. We introduced the CREATE command in the previous section, so we will now give a brief overview of some of the other features of Cypher.
- A Cypher query is made up of *clauses*. When a query has several clauses, the result from one clause can be the input to the next clause in the query. We will give a flavor of the language by discussing some of the clauses using examples. Our presentation is not meant to be a detailed presentation on Cypher, just an introduction to some of the languages features. Figure (c) summarizes some of the main clauses that can be part of a Cyber query. The Cyber language can specify complex queries and updates on a graph database. We will give a few of

examples to illustrate simpleCyber queries in Figure (d).

Figure.

Examples in Neo4j using the Cypher language. (a) Creating some nodes. (b) Creating some relationships.

(a) creating some nodes for the COMPANY data (from Figure 5.6):

```
CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})
...
CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})
...
CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})
CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})
CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})
CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})
...
CREATE (loc1: LOCATION, {Lname: 'Houston'})
CREATE (loc2: LOCATION, {Lname: 'Stafford'})
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})
...
```

(b) creating some relationships for the COMPANY data (from Figure 5.6):

```
CREATE (e1) – [ : WorksFor ] –> (d1)
CREATE (e3) – [ : WorksFor ] –> (d2)
...
CREATE (d1) – [ : Manager ] –> (e2)
CREATE (d2) – [ : Manager ] –> (e4)
...
CREATE (d1) – [ : LocatedIn ] –> (loc1)
CREATE (d1) – [ : LocatedIn ] –> (loc3)
CREATE (d1) – [ : LocatedIn ] –> (loc4)
CREATE (d2) – [ : LocatedIn ] –> (loc2)
...
CREATE (e1) – [ : WorksOn, {Hours: '32.5'} ] –> (p1)
CREATE (e1) – [ : WorksOn, {Hours: '7.5'} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p1)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p3)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p4)
...
```

(c) **Basic simplified syntax of some common Cypher clauses:**

Finding nodes and relationships that match a pattern: MATCH <pattern>
Specifying aggregates and other query variables: WITH <specifications>
Specifying conditions on the data to be retrieved: WHERE <condition>
Specifying the data to be returned: RETURN <data>
Ordering the data to be returned: ORDER BY <data>
Limiting the number of returned data items: LIMIT <max number>
Creating nodes: CREATE <node, optional labels and properties>
Creating relationships: CREATE <relationship, relationship type and optional properties>
Deletion: DELETE <nodes or relationships>
Specifying property values and labels: SET <property values and labels>
Removing property values and labels: REMOVE <property values and labels>

(d) **Examples of simple Cypher queries:**

1. MATCH (d : DEPARTMENT {Dno: '5'}) − [: LocatedIn] → (loc)
   RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: '2'}) − [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e) − [w: WorksOn] → (p: PROJECT {Pno: 2})
   RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) − [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
5. MATCH (e) − [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
   LIMIT 10
6. MATCH (e) − [w: WorksOn] → (p)
   WITH e, COUNT(p) AS numOfprojs
   WHERE numOfprojs > 2
   RETURN e.Ename , numOfprojs
   ORDER BY numOfprojs
7. MATCH (e) − [w: WorksOn] → (p)
   RETURN e , w, p
   ORDER BY e.Ename
   LIMIT 10
8. MATCH (e: EMPLOYEE {Empid: '2'})
   SET e.Job = 'Engineer'

Query 1 in Figure 24.4(d) shows how to use the MATCH and RETURN clauses in a query, and the query retrieves the locations for department number 5. Match specifies the *pattern* and the *query variables* (*d* and *loc*) and RETURN specifies the query result to be retrieved by refering to the query variables. Query 2 has three variables (*e*, *w*, and *p*), and returns the projects and hours per week that the employee with

Empid = 2 works on. Query 3, on the other hand, returns the employees and hours per week who work on the project with Pno = 2. Query 4 illustrates the ORDER BY clause and returns all employees and the projects they work on, sorted by Ename. It is also possible to limit the number of returned results by using the LIMIT clause as in query 5, which only returns the first 10 answers.

Query 6 illustrates the use of WITH and aggregation, although the WITH clause can be used to separate clauses in a query even if there is no aggregation. Query 6 also illus- trates the WHERE clause to specify additional conditions, and the query returns the employees who work on more than

two projects, as well as the number of projects each employee works on. It is also common to return the nodes and relationships them- selves in the query result, rather than the property values of the nodes as in the previ- ous queries. Query 7 is similar to query 5 but returns the nodes and relationships only, and so the query result can be displayed as a graph using Neo4j's visualization tool. It is also possible to add or remove labels and properties from nodes. Query 8 shows how to add more properties to a node by adding a Job property to an employee node.

Neo4j Interfaces and Distributed System Characteristics

Neo4j has other interfaces that can be used to create, retrieve, and update nodes and relationships in a graph database. It also has two main versions: the enterprise edi- tion, which comes with additional capabilities, and the community edition. We dis-cuss some of the additional features of Neo4j in this subsection.

- **Enterprise edition vs. community edition.** Both editions support the Neo4jgraph data model and storage system, as well as the Cypher graph query language, and several other interfaces, including a high-performance nativeAPI, language drivers for several popular programming languages, such as Java, Python, PHP, and the REST (Representational State Transfer) API. Inaddition, both editions support ACID properties. The enterprise edition supports additional features for enhancing performance, such as caching and clustering of data and locking.
- **Graph visualization interface.** Neo4j has a graph visualization interface, so that a subset of the nodes and edges in a database graph can be displayed as a graph. This tool can be used to visualize query results in a graph representation.
- **Master-slave replication.** Neo4j can be configured on a cluster of distrib- uted system nodes (computers), where one node is designated the master node. The data and indexes are fully replicated on each node in the cluster. Various ways of synchronizing the data between master and slave nodes canbe configured in the distributed cluster.
- **Caching.** A main memory cache can be configured to store the graph data for improved performance.
- **Logical logs.** Logs can be maintained to recover from failures.

12. Explain about Database Security Issues.

i) Types of Security

Database security is a broad area that addresses many issues, including the following:

- Various legal and ethical issues regarding the right to access certain information—for example, some information may be deemed to be private and cannot be accessed legally by unauthorized organizations or persons. In theUnited States, there are numerous laws governing privacy of information.
- Policy issues at the governmental, institutional, or corporate level regarding what kinds of information should not be made publicly available—for example, credit ratings and personal medical records.
- System-related issues such as the system levels at which various security functions should be enforced—for example, whether a security function should be handled at the physical hardware level, the operating system level,or the DBMS level.
- The need in some organizations to identify multiple security levels and to categorize the data and users based on these classifications—for example, top secret, secret, confidential, and

unclassified. The security policy of the organization with respect to permitting access to various classifications of data must be enforced.

Threats to Databases. Threats to databases can result in the loss or degradation of some or all of the following commonly accepted security goals: integrity, availability, and confidentiality.

- **Loss of integrity.** Database integrity refers to the requirement that information be protected from improper modification. Modification of data includescreating, inserting, and updating data; changing the status of data; and deleting data. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts. If the loss of system or data integrity isnot corrected, continued use of the contaminated system or corrupted data could result in inaccuracy, fraud, or erroneous decisions.
- **Loss of availability.** Database availability refers to making objects available to a human user or a program who/which has a legitimate right to those data objects. Loss of availability occurs when the user or program cannot access these objects.
- **Loss of confidentiality.** Database confidentiality refers to the protection of data from unauthorized disclosure. The impact of unauthorized disclosure of confidential information can range from violation of the Data Privacy Act to the jeopardization of national security. Unauthorized, unanticipated, or unintentional disclosure could result in loss of public confidence, embarrassment, or legal action against the organization.
- Database Security: Not an Isolated Concern. When considering the threats facing databases, it is important to remember that the database management system alone cannot be responsible for maintaining the confidentiality, integrity, and availability of the data. Rather, the database works as part of a network of services, including applications, Web servers, firewalls, SSL terminators, and security monitoring systems. Because security of an overall system is only as strong as its weakestlink, a database may be compromised even if it would have been perfectly secure onits own merits.
- To protect databases against the threats discussed above, it is common to implement four kinds of control measures: access control, inference control, flow control,and encryption.
- In a multiuser database system, the DBMS must provide techniques to enable certain users or user groups to access selected portions of a database without gaining access to the rest of the database. This is particularly important when a large integrated database is to be used by many different users within the same organization. For example, sensitive information such as employee salaries or performance reviews should be kept confidential from most of the database system's users. A DBMS typically includes a database security and authorization subsystem that is responsible for ensuring the security of portions of a database against unauthorized access. It is now customary to refer to two types of database security mechanisms:
- **Discretionary security mechanisms.** These are used to grant privileges to users, including the capability to access specific data files, records, or fieldsin a specified mode (such as read, insert, delete, or update).
- **Mandatory security mechanisms.** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organi zation. For example, a typical security policy is to permit users at a certain classification (or clearance) level to see only the data items classified at the user's own (or lower) classification level. An extension of this is role-basedsecurity, which enforces policies and privileges based on the concept of organizational roles.

ii) Control Measures

Four main control measures are used to provide security of data in databases:

- Access control
- Inference control
- Flow control
- Data encryption

- A security problem common to computer systems is that of preventing unauthorized persons from accessing the system itself, either to obtain information or to make malicious changes in a portion of the database. The security mechanism of a DBMS must include provisions for restricting access to the database system as a whole. This function, called **access control**, is handled by creating user accounts and passwords to control the login process by the DBMS.

- **Statistical databases** are used to provide statistical information or summaries of values based on various criteria. For example, a database for population statistics may provide statistics based on age groups, income levels, household size, education levels, and other criteria. Statistical database users such as government statisticians or market research firms are allowed to access the database to retrieve statistical information about a population but not to access the detailed confidential information about specific individuals. Security for statistical databases must ensure that information about individuals cannot be accessed. It is sometimes possible to deduce or infer certain facts concerning individuals from queries that involve only summary statistics on groups; consequently, this must not be per- mitted either. This problem, called **statistical database security.** The corresponding control measures are called **inference control** measures.

- Another security issue is that of **flow control**, which prevents information from flowing in such a way that it reaches unauthorized users. **Covert channels** are pathways on which information flows implicitly in ways that violate the security policy of an organization.

- A final control measure is **data encryption**, which is used to protect sensitive data(such as credit card numbers) that is transmitted via some type of communicationsnetwork. Encryption can be used to provide additional protection for sensitive portions of a database as well. The data is **encoded** using some coding algorithm. An unauthorized user who accesses encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher the data. Encrypting techniques that are very difficult to decode without a keyhave been developed for military applications. However, encrypted database records are used today in both private organizations and governmental and military applications. In fact, state and federal laws prescribe encryption for any systemthat deals with legally protected personal information. For example, according to Georgia Law (OCGA 10-1-911):

- "Personal information" means an individual's first name or first initial and last name in combination with any one or more of the following data elements, when either the name or the data elements are not encrypted or redacted:

  - Social security number;
  - Driver's license number or state identification card number;
  - Account number, credit card number, or debit card number, if circum- stances exist wherein such a number could be used without additional identifying information, access codes, or passwords;
  - Account passwords or personal identification numbers or other access codes.

Because laws defining what constitutes personal information vary from state to state, systems must protect individuals' privacy and enforce privacy measures adequately. Discretionary access control alone may not suffice. .

iii) Database Security and the DBA

The database administrator (DBA) is the central authority for managing a database system. The DBA's responsibilities include granting privileges to users who need to use the system and

classifying users and data in accordance with the policy of the organization. The DBA has a **DBA account** in the DBMS, sometimes called a **system** or **super user account**, which provides powerful capabilities that are not made available to regular database accounts and users. DBA-privileged commands include commands for granting and revoking privileges to individual accounts, users, or user groups and for performing the following types of actions:

1. **Account creation.** This action creates a new account and password for a user or a group of users to enable access to the DBMS.
2. **Privilege granting.** This action permits the DBA to grant certain privileges to certain accounts.
3. **Privilege revocation.** This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts.
4. **Security level assignment.** This action consists of assigning user accounts to the appropriate security clearance level.

The DBA is responsible for the overall security of the database system. Action 1 in the preceding list is used to control access to the DBMS as a whole, whereas actions 2 and 3 are used to control discretionary database authorization, and action 4 is used to control mandatory authorization.

iv) Access Control, User Accounts, and Database Audits

Whenever a person or a group of persons needs to access a database system, the individual or group must first apply for a user account. The DBA will then create a new **account number** and **password** for the user if there is a legitimate need to access the database. The user must **log in** to the DBMS by entering the account number and password whenever database access is needed. The DBMS checks that the account number and password are valid; if they are, the user is permitted to use the DBMS and to access the database. Application programs can also be considered users and are required to log in to the database.

It is straightforward to keep track of database users and their accounts and pass- words by creating an encrypted table or file with two fields: AccountNumber and Password. This table can easily be maintained by the DBMS. Whenever a new account is created, a new record is inserted into the table. When an account is canceled, the corresponding record must be deleted from the table.

The database system must also keep track of all operations on the database that are applied by a certain user throughout each **login session**, which consists of the sequence of database interactions that a user performs from the time of logging in to the time of logging off. When a user logs in, the DBMS can record the user's account number and associate it with the computer or device from which the user logged in. All operations applied from that computer or device are attributed to the user's account until the user logs off. It is particularly important to keep track of update operations that are applied to the database so that, if the database is tampered with, the DBA can determine which user did the tampering.

To keep a record of all updates applied to the database and of particular users who applied each update, we can modify the system log. The **system log** includes an entry for each operation applied to the database that may be required for recovery from a transaction failure or system crash. We can expand the log entries so that they also include the account number of the user and the online computer or device ID that applied each operation recorded in the log. If any tampering with the database is suspected, a **database audit** is performed, which consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period. When an illegal or unauthorized operation is found, the DBA can determine the account number used to perform the operation. Database audits are particularly important for sensitive databases that are updated by many transactions and users, such as a banking database that can be updated by thousands of bank tellers. A database log that is used mainly for security purposes serves as an **audit trail**.

v) Sensitive Data and Types of Disclosures

Sensitivity of data is a measure of the importance assigned to the data by its owner for the purpose of denoting its need for protection. Some databases contain only sensitive data whereas other databases may contain no sensitive data at all. Handling databases that fall at these two extremes is relatively easy because such databases can be covered by access control, which is explained in the next section. The situation becomes tricky when some of the data is sensitive whereas other data is not.

Several factors can cause data to be classified as sensitive:

1. **Inherently sensitive.** The value of the data itself may be so revealing or confidential that it becomes sensitive—for example, a person's salary or who apatient has HIV/AIDS.
2. **From a sensitive source.** The source of the data may indicate a need for secrecy—for example, an informer whose identity must be kept secret.
3. **Declared sensitive.** The owner of the data may have explicitly declared it assensitive.
4. **A sensitive attribute or sensitive record.** The particular attribute or record may have been declared sensitive—for example, the salary attribute of an employee or the salary history record in a personnel database.
5. **Sensitive in relation to previously disclosed data.** Some data may not be sensitive by itself but will become sensitive in the presence of some other data—for example, the exact latitude and longitude information for a location where some previously recorded event happened that was later deemed sensitive.

It is the responsibility of the database administrator and security administrator to collectively enforce the security policies of an organization. This dictates whether access should or should not be permitted to a certain database attribute (also knownas a table column or a data element) for individual users or for categories of users. Several factors must be considered before deciding whether it is safe to reveal the data. The three most important factors are data availability, access acceptability, and authenticity assurance.

1. **Data availability.** If a user is updating a field, then this field becomes inaccessible and other users should not be able to view this data. This blocking is only temporary and only to ensure that no user sees any inaccurate data. This is typically handled by the concurrency control mechanism.
2. **Access acceptability.** Data should only be revealed to authorized users. A database administrator may also deny access to a user request even if the request does not directly access a sensitive data item, on the grounds that the requested data may reveal information about the sensitive data that the useris not authorized to have.
3. **Authenticity assurance.** Before granting access, certain external character-istics about the user may also be considered. For example, a user may only be permitted access during working hours. The system may track previous queries to ensure that a combination of queries does not reveal sensitive data. The latter is particularly relevant to statistical database queries.

The term precision, when used in the security area, refers to allowing as much as possible of the data to be available, subject to protecting exactly the subset of data that is sensitive. The definitions of security versus precision are as follows:

- **Security:** Means of ensuring that data is kept safe from corruption and that access to it is suitably controlled. To provide security means to disclose only nonsensitive data and to reject any query that references a sensitive field.
- **Precision:** To protect all sensitive data while disclosing or making avail- able as much nonsensitive data as possible. Note that this definition of precision is not related to the precision of information retrieval.

The ideal combination is to maintain perfect security with maximum precision. If we want to maintain security, precision must be sacrificed to some degree. Hence there is typically a tradeoff between security and precision.

vi) Relationship between Information Securityand Information Privacy

The rapid advancement of the use of information technology (IT) in industry, government, and academia raises challenging questions and problems regarding the protection and use of personal information. Questions of who has what rights to information about individuals for which purposes become more important as we move toward a world in which it is technically possible to know just about anything about anyone.

Deciding how to design privacy considerations in technology for the future includes philosophical, legal, and practical dimensions. There is a considerable overlap between issues related to access to resources (security) and issues related to appropriate use of information (privacy).

Security in information technology refers to many aspects of protecting a system from unauthorized use, including authentication of users, information encryption, access control, firewall policies, and intrusion detection. For our purposes here, we will limit our treatment of security to the concepts associated with how well a sys- tem can protect access to information it contains. The concept of **privacy** goes beyond security. Privacy examines how well the use of personal information that the system acquires about a user conforms to the explicit or implicit assumptions regarding that use. From an end user perspective, privacy can be considered from two different perspectives: preventing storage of personal information versus ensuring appropriate use of personal information.

For the purposes of this chapter, a simple but useful definition of **privacy** is the ability of individuals to control the terms under which their personal information is acquired and used. In summary, security involves technology to ensure that information is appropriately protected. Security is a required building block for privacy. Privacy involves mechanisms to support compliance with some basic principles and other explicitly stated policies. One basic principle is that people should be informed about information collection, told in advance what will be done with their information, and given a reasonable opportunity to approve or disapprove of such use of the information. A related concept, **trust**, relates to both security and privacy and is seen as increasing when it is perceived that both security and privacy are provided for.

13. Discuss about Discretionary Access Control Basedon Granting and Revoking Privileges.

The typical method of enforcing **discretionary access control** in a database systemis based on the granting and revoking of **privileges**. Let us consider privileges in thecontext of a relational DBMS. Many current relational DBMSs use some variation of this technique. The main idea is to include statements in the query language that allow the DBA and selected users to grant and revoke privileges.

Types of Discretionary Privileges

In SQL2 and later versions, the concept of an **authorization identifier** is used to refer, roughly speaking, to a user account (or group of user accounts). For simplic- ity, we will use the words user or account interchangeably in place of authorization identifier. The DBMS must provide selective access to each relation in the database based on specific accounts. Operations may also be controlled; thus, having an account does not necessarily entitle the account holder to all the functionality provided by the DBMS. Informally, there are two levels for assigning privileges to usethe database system:

- **The account level.** At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- **The relation (or table) level.** At this level, the DBA can control the privilege to access each individual relation or view in the database.
- The privileges at the **account level** apply to the capabilities provided to the account itself and can include the CREATE SCHEMA or CREATE TABLE privilege, to create a schema or base relation; the CREATE VIEW privilege; the ALTER privilege, to apply schema changes such as adding or removing attributes from relations; the DROP privilege, to delete relations or views; the MODIFY privilege, to insert, delete, or update tuples; and the SELECT privilege, to retrieve information from the database by using a SELECT query. Notice that these account privileges apply to the account in general. If a certain account does not have the CREATE TABLE privilege, no rela- tions can be created from that account. Account-level privileges are not defined as part of SQL2; they are left to the DBMS implementers to define. In earlier versions of SQL, a CREATETAB privilege existed to give an account the privilege to create tables (relations).

- The second level of privileges applies to the **relation level**, which includes base relations and virtual (view) relations. These privileges are defined for SQL2. In the following discussion, the term relation may refer either to a base relation or to a view, unless we explicitly specify one or the other. Privileges at the relation level specify for each user the individual relations on which each type of command can be applied. Some privileges also refer to individual columns (attributes) of relations. SQL2 commands provide privileges at the relation and attribute level only. Although this distinction is general, it makes it difficult to create accounts with limited privileges. The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the **access matrix model**, where the rows of a matrix M represent subjects (users, accounts, programs) and the columns represent objects (relations, records, columns, views, operations). Each position M(i, j) in the matrix represents the types of privileges (read, write, update) that subject i holds on object j.
- To control the granting and revoking of relation privileges, each relation R in a database is assigned an **owner account**, which is typically the account that was used when the relation was created in the first place. The owner of a relation is given all privileges on that relation. In SQL2, the DBA can assign an owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the CREATE SCHEMA command. The owner account holder can pass privileges on any of the owned relations to other users by **granting** privileges to their accounts. In SQL, the following types of privileges can be granted on each individual relation R:
- SELECT **(retrieval or read) privilege on** R. Gives the account retrieval privilege. In SQL, this gives the account the privilege to use the SELECT statement to retrieve tuples from R.
- **Modification privileges on** R. This gives the account the capability to modify the tuples of R. In SQL, this includes three privileges: UPDATE, DELETE, and INSERT. These correspond to the three SQL commands for modifying a table R. Additionally, both the INSERT and UPDATE privileges can specify that only certain attributes of R can be modified by the account.
- **References privilege on** R. This gives the account the capability to reference (or refer to) a relation R when specifying integrity constraints. This privilege can also be restricted to specific attributes of R.

Notice that to create a view, the account must have the SELECT privilege on all relations involved in the view definition in order to specify the query that corresponds to the view.

Specifying Privileges through the Use of Views

The mechanism of **views** is an important discretionary authorization mechanism in its own right. For example, if the owner A of a relation R wants another account B to be able to retrieve only some fields of R, then A can create a view V of R that

includes only those attributes and then grant SELECT on V to B. The same applies can be to limiting B to retrieving only certain tuples of R; a view V created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access.

Revoking of Privileges

- In some cases, it is desirable to grant a privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for **revoking** privileges is needed. In SQL, a REVOKE command is included for the purpose of canceling privileges.

Propagation of Privileges Using the GRANT OPTION

- Whenever the owner A of a relation R grants a privilege on R to another account B, the privilege can be given to B with or without the GRANT OPTION. If the GRANT OPTION is given, this means that B can also grant that privilege on R to other accounts. Suppose that B is given the GRANT OPTION by A and that B then grants the privilege on R to a third account C, also with the GRANT OPTION. In this way, privileges on R can **propagate** to other accounts without the knowledge of the owner of R. If the owner account A now revokes the privilege granted to B, all the privileges that B propagated based on that privilege should automatically be revoked by the system.
- It is possible for a user to receive a certain privilege from two or more sources. For example, A4 may receive a certain UPDATE R privilege from both A2 and A3. In such a case, if A2 revokes this privilege from A4, A4 will still continue to have the privilege by virtue of having been granted it from A3. If A3 later revokes the privilege from A4, A4 totally loses the privilege. Hence, a DBMS that allows propagation of privileges must keep track of how all the privileges were granted in the form of some internal log so that revoking of privileges can be done correctly and completely.

An Example to Illustrate Granting and Revoking of Privileges

Suppose that the DBA creates four accounts–A1, A2, A3, and A4–and wants only A1 to be able to create base relations. To do this, the DBA must issue the following GRANT command in SQL:

```
GRANT CREATETAB TO A1;
```

The CREATETAB (create table) privilege gives account A1 the capability to create new database tables (base relations) and is hence an account privilege. This privilege was part of earlier versions of SQL but is now left to each individual system implementation to define. Note that A1 , A2, and so forth may be individuals, like John in IT department or Mary in marketing; but they may also be applications or programs that want to access a database.

In SQL2, the same effect can be accomplished by having the DBA issue a CREATE SCHEMA command, as follows:

```
CREATE SCHEMA EXAMPLE AUTHORIZATION A1;
```

User account A1 can now create tables under the schema called EXAMPLE. To continue our example, suppose that A1 creates the two base relations EMPLOYEE and DEPARTMENT shown in Figure ; A1 is then the **owner** of these two relations andhence has all the relation privileges on each of them.

Next, suppose that account A1 wants to grant to account A2 the privilege to insert and delete tuples in both of these relations. However, A1 does not want A2 to be able to propagate these privileges to additional accounts. A1 can issue the following command:

GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;

Notice that the owner account A1 of a relation automatically has the GRANT OPTION, allowing it to grant privileges on the relation to other accounts. However, account A2 cannot grant INSERT and DELETE privileges on the EMPLOYEE and DEPARTMENT tables because A2 was not given the GRANT OPTION in the preceding command.

Next, suppose that A1 wants to allow account A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts. A1 can issue the following command:

GRANT SELECT ON EMPLOYEE, DEPARTMENT TO A3 WITH GRANT OPTION;

The clause WITH GRANT OPTION means that A3 can now propagate the privilege to other accounts by using GRANT. For example, A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing the following command:

GRANT SELECT ON EMPLOYEE TO A4;

Notice that A4 cannot propagate the SELECT privilege to other accounts because the GRANT OPTION was not given to A4.

Now suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 then can issue this command:

REVOKE SELECT ON EMPLOYEE FROM A3;

Figure . Schemas for the two relations EMPLOYEE and DEPARTMENT.

EMPLOYEE

Name	Ssn	Bdate	Address	Sex	Salary	Dno

DEPARTMENT

Dnumber	Dname	Mgr_ssn

The DBMS must now revoke the SELECT privilege on EMPLOYEE from A3, and it must also automatically revoke the SELECT privilege on EMPLOYEE from A4. This is because A3 granted that privilege to A4, but A3 does not have the privilege any more.

Next, suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege. The limitation is to retrieve only the Name, Bdate, and Address attributes and only for the tuples with Dno = 5. A1

then can create the following view:

```
CREATE VIEW A3EMPLOYEE AS SELECT Name, Bdate, Address

FROM EMPLOYEE WHERE Dno = 5;
```

After the view is created, A1 can grant SELECT on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT OPTION;
```

Finally, suppose that A1 wants to allow A4 to update only the Salary attribute of EMPLOYEE; A1 can then issue the following command:

```
GRANT UPDATE ON EMPLOYEE (Salary) TO A4;
```

The UPDATE and INSERT privileges can specify particular attributes that may be updated or inserted in a relation. Other privileges (SELECT, DELETE) are not attribute specific, because this specificity can easily be controlled by creating the appropriate views that include only the desired attributes and granting the corresponding privileges on the views. However, because updating views is not always possible, the UPDATE and INSERT privileges are given the option to specify the particular attributes of a base relation that may be updated.

Specifying Limits on Propagation of Privileges

- Techniques to limit the propagation of privileges have been developed, although they have not yet been implemented in most DBMSs and are not a part of SQL. Limiting **horizontal propagation** to an integer number i means that an account B given the GRANT OPTION can grant the privilege to at most i other accounts. **Vertical propagation** is more complicated; it limits the depth of the granting of privileges. Granting a privilege with a vertical propagation of zero is equivalent to granting the privilege with no GRANT OPTION. If account A grants a privilege to account B with the vertical propagation set to an integer number j > 0, this means that the account B has the GRANT OPTION on that privilege, but B can grant the privilege to other accounts only with a vertical propagation less than j. In effect, vertical propagation limits the sequence of GRANT OPTIONS that can be given from one account to the next based on a single original grant of the privilege.
- We briefly illustrate horizontal and vertical propagation limits–which are not available currently in SQL or other relational systems–with an example. Suppose that A1 grants SELECT to A2 on the EMPLOYEE relation with horizontal propagation equal to 1 and vertical propagation equal to 2. A2 can then grant SELECT to at most one account because the horizontal propagation limitation is set to 1. Additionally, A2 cannot grant the privilege to another account except with vertical propagation set to 0 (no GRANT OPTION) or 1; this is because A2 must reduce the vertical propagation by at least 1 when passing the privilege to others. In addition, the horizontal propagation must be less than or equal to the originally granted horizontal propagation. For example, if account A grants a privilege to account B with the horizontal propagation set to an integer number j > 0, this means that B can grant the privilege to other accounts only with a horizontal propagation less than or equal to j. As this example shows, horizontal and vertical propagation techniques are designed to limit the depth and breadth of propagation of privileges.

14. Elaborate about Mandatory Access Control and Role-BasedAccess Control for Multilevel Security.

- The discretionary access control technique of granting and revoking privileges on relations has traditionally been the main security mechanism for relational data- base systems. This is an all-or-nothing method: A user either has or does not have a certain

privilege. In many applications, an additional security policy is needed that classifies data and users based on security classes. This approach, known as **mandatory access control (MAC)**, would typically be combined with the discretionary access control mechanisms. It is important to note that most mainstream RDBMSs currently provide mechanisms only for discretionary access control. However, the need for multilevel security exists in government, military, and intelligence applications, as well as in many industrial and corporate applications. Because of the overriding concerns for privacy, in many systems the levels are determined by who has what access to what private information (also called personally identifiable information). Some DBMS vendors—for example, Oracle—have released special versions of their RDBMSs that incorporate mandatory access control for government use.

- Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest. Other more complex security classification schemes exist, in which the security classes are organized in a lattice. For simplicity, we will use the system with four security classification levels, where TS ≥ S ≥ C ≥ U, to illustrate our discussion. The commonly used model for multilevel security, known as the Bell-LaPadula model, classifies each **subject** (user, account, program) and **object** (relation, tuple, column, view, operation) into one of the security classifications TS, S, C, or U. We will refer to the **clearance** (classification) of a subject S as **class(S)** and to the **classification** of an object O as **class(O)**. Two restrictions are enforced on data access based on the subject/object classifications:

1. A subject S is not allowed read access to an object O unless class(S) ≥ class(O). This is known as the **simple security property**.
2. A subject S is not allowed to write an object O unless class(S) ≤ class(O). This is known as the **star property** (or *-property).

The first restriction is intuitive and enforces the obvious rule that no subject can read an object whose security classification is higher than the subject's security clearance. The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance. Violation of this rule would allow information to flow from higher to lower classifications, which violates a basic tenet of multilevel security. For example, a user (subject) with TS clearance may make a copy of an object with classification TS and then write it back as a new object with classification U, thus making it visible throughout the system.

To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute A is associated with a **classification attribute** C in the schema, and each attribute value in a tuple is associated with a corresponding security classification. In addition, in some models, a **tuple classification** attribute TC is added to the relation attributes to provide a classification for each tuple as a whole. The model we describe here is known as the multilevel model, because it allows classifications at multiple security levels. A **multilevel relation** schema R with n attributes would be represented as:

$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$

where each C_i represents the classification attribute associated with attribute A_i.

The value of the tuple classification attribute TC in each tuple t—which is the highest of all attribute classification values within t—provides a general classification for the tuple itself. Each attribute classification Ci provides a finer security classification for each attribute value within the tuple. The value of TC in each tuple t is the highest of all attribute classification values Ci within t.

The **apparent key** of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels. In some cases, it is possible to store a single tuple in the relation at a higher classifica-tion level and produce the corresponding tuples

at a lower-level classification through a process known as **filtering**. In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the apparent key. This leads to the concept of **polyinstantiation**,[5] where several tuples can have the same apparent key value but have different attribute values for users at different clearance levels.

We illustrate these concepts with the simple example of a multilevel relation shown in Figure (a), where we display the classification attribute values next to each

(a) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

(b) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	NULL C	C
Brown C	NULL C	Good C	C

(c) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	NULL U	NULL U	U

(d) EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Smith U	40000 C	Excellent C	C
Brown C	80000 S	Good C	S

Figure . A multilevel relation to illustrate multilevel security.

(a) The original EMPLOYEE tuples. (b) Appearance of EMPLOYEE after filtering for classification C users.
(c) Appearance of EMPLOYEE after filtering for classification U users.
(d) Polyinstantiation of the Smith tuple.

attribute's value. Assume that the Name attribute is the apparent key, and consider the query SELECT * FROM EMPLOYEE. A user with security clearance S would see the same relation shown in Figure (a), since all tuple classifications are less than or equal to
S. However, a user with security clearance C would not be allowed to see the values for Salary of 'Brown' and Job_performance of 'Smith', since they have higher classification. The tuples would be **filtered** to appear as shown in Figure 30.2(b), with Salary and Job_performance appearing as null. For a user with security clearance U, the filtering allows only the Name attribute of 'Smith' to appear, with all the other attributes appearing as null . Thus, filtering introduces null values for attribute values whose security classification is higher than the user's security clearance.

In general, the **entity integrity** rule for multilevel relations states that all attributes that are members of the apparent key must not be null and must have the same security classification within each individual tuple. Additionally, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key. This constraint ensures that a user

can see the key if the user is permitted to see any part of the tuple. Other integrity rules, called **null integrity** and **interinstance integrity**, informally ensure that if a tuple value at some security level can be filtered (derived) from a higher-classified tuple, then it is sufficient to store the higher-classified tuple in the multilevel relation.

To illustrate polyinstantiation further, suppose that a user with security clearance C tries to update the value of Job_performance of 'Smith' in Figure to 'Excellent'; this corresponds to the following SQL update being submitted by that user:

```
UPDATE EMPLOYEE SET Job_performance = 'Excellent' WHERE
        Name = 'Smith';
```

Since the view provided to users with security clearance C permits such an update, the system should not reject it; otherwise, the user could infer that some nonnull value exists for the Job_performance attribute of 'Smith' rather than the null value that appears. This is an example of inferring information through what is known as a **covert channel**, which should not be permitted in highly secure systems. However, the user should not be allowed to overwrite the existing value of Job_performance at the higher classification level. The solution is to create a **polyinstantiation** for the 'Smith' tuple at the lower classification level C, as shown in Figure (d). This is necessary since the new tuple cannot be filtered from the existing tuple at classification S.

The basic update operations of the relational model (INSERT, DELETE, UPDATE) must be modified to handle this and similar situations, but this aspect of the problem is outside the scope of our presentation. We refer the interested reader to the Selected Bibliography at the end of this chapter for further details.

Comparing Discretionary Access Control and Mandatory Access Control

Discretionary access control (DAC) policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains. The main drawback of DAC models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs. The reason for this vulnerability is that discretionary authorization models do not impose any control on how information is propagated and used once it has been accessed by users authorized to do so. By contrast, mandatory policies ensure a high degree of protection—in a way, they prevent any illegal flow of information. Therefore, they are suitable for military and high-security types of applications, which require a higher degree of protection. However, mandatory policies have the drawback of being too rigid in that they require a strict classification of subjects and objects into security levels, and therefore they are applicable to few environments and place an additional bur- den of labeling every object with its security classification. In many practical situations, discretionary policies are preferred because they offer a better tradeoff between security and applicability than mandatory policies.

Role-Based Access Control

- Role-based access control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise-wide systems.

- Its basic notion is that privileges and other permissions are associated with organizational **roles** rather than with individual users. Individual users are then assigned to appropriate roles. Roles can be created using the CREATE ROLE and DESTROY ROLE commands. The GRANT and REVOKE commands can then be used to assign and revoke privileges from roles, as well as for individual users when needed. For example, a company may have roles such as sales account manager, purchasing agent, mailroom clerk, customer service manager, and so on. Multiple individuals can be assigned to each

role. Security privileges that are common to a role are granted to the role name, and any individual assigned to this role would automatically have those privileges granted.

- RBAC can be used with traditional discretionary and mandatory access controls; it ensures that only authorized users in their specified roles are given access to certain data or resources. Users create sessions during which they may activate a subset of roles to which they belong. Each session can be assigned to several roles, but it maps to one user or a single subject only. Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.
- Separation of duties is another important requirement in various mainstream DBMSs. It is needed to prevent one user from doing work that requires the involve ment of two or more people, thus preventing collusion. One method in which separation of duties can be successfully implemented is with mutual exclusion of roles. Two roles are said to be **mutually exclusive** if both the roles cannot be used simultaneously by the user. **Mutual exclusion of roles** can be categorized into two types, namely authorization time exclusion (static) and runtime exclusion (dynamic). In authorization time exclusion, two roles that have been specified as mutually exclusive cannot be part of a user's authorization at the same time. In runtime exclusion, both these roles can be authorized to one user but cannot be activated by the user at the same time. Another variation in mutual exclusion of roles is that of complete and partial exclusion.
- The **role hierarchy** in RBAC is a natural way to organize roles to reflect the organization's lines of authority and responsibility. By convention, junior roles at the bottom are connected to progressively senior roles as one moves up the hierarchy. The hierarchic diagrams are partial orders, so they are reflexive, transitive, and antisymmetric. In other words, if a user has one role, the user automatically has roles lower in the hierarchy. Defining a role hierarchy involves choosing the type of hierarchy and the roles, and then implementing the hierarchy by granting roles to other roles. Role hierarchy can be implemented in the following manner:

```
GRANT ROLE full_time TO employee_type1

GRANT ROLE intern TO employee_type2
```

The above are examples of granting the roles full_time and intern to two types of employees.

- Another issue related to security is identity management. **Identity** refers to a unique name of an individual person. Since the legal names of persons are not necessarily unique, the identity of a person must include sufficient additional information to make the complete name unique. Authorizing this identity and managing the schema of these identities is called **identity management.** Identity management addresses how organizations can effectively authenticate people and manage their access to confidential information. It has become more visible as a business require- ment across all industries affecting organizations of all sizes. Identity management administrators constantly need to satisfy application owners while keeping expenditures under control and increasing IT efficiency.
- Another important consideration in RBAC systems is the possible temporal con- straints that may exist on roles, such as the time and duration of role activations and the timed triggering of a role by an activation of another role. Using an RBAC model is a highly desirable goal for addressing the key security requirements of Web-based applications. Roles can be assigned to workflow tasks so that a user with any of the roles related to a task may be authorized to execute it and may play a certain role only for a certain duration.
- RBAC models have several desirable features, such as flexibility, policy neutrality, better support for security management and administration, and a natural enforcement of the hierarchical organization structure within organizations. They also have other aspects that make them attractive candidates for developing secure Web-based applications. These

features are lacking in DAC and MAC models. RBAC models do include the capabilities available in traditional DAC and MAC policies. Furthermore, an RBAC model provides mechanisms for addressing the security issues related to the execution of tasks and workflows, and for specifying user-defined and organization-specific policies. Easier deployment over the Inter- net has been another reason for the success of RBAC models.

Label-Based Security and Row-Level Access Control

- Many mainstream RDBMSs currently use the concept of row-level access control, where sophisticated access control rules can be implemented by considering the data row by row. In row-level access control, each data row is given a label, which is used to store information about data sensitivity. Row-level access control pro- vides finer granularity of data security by allowing the permissions to be set for each row and not just for the table or column. Initially the user is given a default session label by the database administrator. Levels correspond to a hierarchy of data-sensitivity levels to exposure or corruption, with the goal of maintaining privacy or security. Labels are used to prevent unauthorized users from viewing or altering certain data. A user having a low authorization level, usually represented by a low number, is denied access to data having a higher-level number. If no such label is given to a row, a row label is automatically assigned to it depending upon the user's session label.
- A policy defined by an administrator is called a **label security policy**. Whenever data affected by the policy is accessed or queried through an application, the policy is automatically invoked. When a policy is implemented, a new column is added to each row in the schema. The added column contains the label for each row that reflects the sensitivity of the row as per the policy. Similar to MAC (mandatory access control), where each user has a security clearance, each user has an identity in label-based security. This user's identity is compared to the label assigned to each row to determine whether the user has access to view the contents of that row. However, the user can write the label value himself, within certain restrictions and guidelines for that specific row. This label can be set to a value that is between the user's current session label and the user's minimum level. The DBA has the privi- lege to set an initial default row label.
- The label security requirements are applied on top of the DAC requirements for each user. Hence, the user must satisfy the DAC requirements and then the label security requirements to access a row. The DAC requirements make sure that the user is legally authorized to carry on that operation on the schema. In most applications, only some of the tables need label-based security. For the majority of the application tables, the protection provided by DAC is sufficient.
- Security policies are generally created by managers and human resources personnel. The policies are high-level, technology neutral, and relate to risks. Policies are a result of management instructions to specify organizational procedures, guiding principles, and courses of action that are considered to be expedient, prudent, or advantageous. Policies are typically accompanied by a definition of penalties and countermeasures if the policy is transgressed. These policies are then interpreted and converted to a set of label-oriented policies by the **label security administrator**, who defines the security labels for data and authorizations for users; these labels and authorizations govern access to specified protected objects.
- Suppose a user has SELECT privileges on a table. When the user executes a SELECT statement on that table, label security will automatically evaluate each row returned by the query to determine whether the user has rights to view the data. For example, if the user has a sensitivity of 20, then the user can view all rows having a security level of 20 or lower. The level determines the sensitivity of the information con- tained in a row; the more sensitive the row, the higher its security label value. Such label security can be

configured to perform security checks on UPDATE, DELETE, and INSERT statements as well.

XML Access Control

- With the worldwide use of XML in commercial and scientific applications, efforts are under way to develop security standards. Among these efforts are digital signatures and encryption standards for XML. The XML Signature Syntax and Process- ing specification describes an XML syntax for representing the associations between cryptographic signatures and XML documents or other electronic resources. The specification also includes procedures for computing and verifying XML signatures. An XML digital signature differs from other protocols for message signing, such as **OpenPGP** (**Pretty Good Privacy**—a confidentiality and authentication ser- vice that can be used for electronic mail and file storage application), in its supportfor signing only specific portions of the XML tree rather than the complete document. Additionally, the XML signature specification defines mechanisms for countersigning and transformations—so-called canonicalization—to ensure that two instances of the same text produce the same digest for signing even if their representations differ slightly; for example, in typographic white space.
- The XML Encryption Syntax and Processing specification defines XML vocabulary and processing rules for protecting confidentiality of XML documents in whole or in part and of non-XML data as well. The encrypted content and additional processing information for the recipient are represented in well-formed XML so that the result can be further processed using XML tools. In contrast to other commonly used technologies for confidentiality, such as SSL (Secure Sockets Layer—a leading Internet security protocol) and virtual private networks, XML encryption also applies to parts of documents and to documents in persistent storage. Database systems such as PostgreSQL or Oracle support JSON (JavaScript Object Notation) objects as a data format and have similar facilities for JSON objects like those defined above for XML.

Access Control Policies for the Weband Mobile Appplications

Publicly accessible Web application environments present a unique challenge to database security. These systems include those responsible for handling sensitive orprivate information and include social networks, mobile application API servers, and e-commerce transaction platforms.

Electronic commerce (**e-commerce**) environments are characterized by any trans- actions that are done electronically. They require elaborate access control policies that go beyond traditional DBMSs. In conventional database environments, accesscontrol is usually performed using a set of authorizations stated by security officersor users according to some security policies. Such a simple paradigm is not well suited for a dynamic environment like e-commerce. Furthermore, in an e-commerce environment the resources to be protected are not only traditional data but also knowledge and experience. Such peculiarities call for more flexibility in specifying access control policies. The access control mechanism must be flexible enough to support a wide spectrum of heterogeneous protection objects.

Because many reservation, ticketing, payment, and online shopping systems process information that is protected by law, the security architecture that goes beyond simple database access control must be put in place to protect the information. When an unauthorized party inappropriately accesses protected information, it amounts to a data breach, which has significant legal and financial consequences. This unauthorized party could be an adversary that actively seeks to steal protectedinformation or may be an employee who overstepped his or her role or incorrectly distributed protected information to others. Inappropriate handling of credit card data, for instance, has led to significant data breaches at major retailers.

In conventional database environments, access control is usually performed using a set of authorizations stated by security officers. But in Web applications, it is all too common that the

Web application itself is the user rather than a duly authorized individual. This gives rise to a situation where the DBMS's access control mechanisms are bypassed and the database becomes just a relational data store to the sys- tem. In such environments, vulnerabilities like SQL injection become significantly more dangerous because it may lead to a total data breach rather than being limited to data that a particular account is authorized to access.

To protect against data breaches in these systems, a first requirement is a comprehensive information security policy that goes beyond the technical access control mechanisms found in mainstream DBMSs. Such a policy must protect not only traditional data, but also processes, knowledge, and experience.

A second related requirement is the support for content-based access control. **Content-based access control** allows one to express access control policies that take the protection object content into account. In order to support content-based access control, access control policies must allow inclusion of conditions based onthe object content.

A third requirement is related to the heterogeneity of subjects, which requires access control policies based on user characteristics and qualifications rather than on specific and individual characteristics (for example, user Ids). A possible solu- tion that will allow better accounting of user profiles in the formulation of access control policies, is to support the notion of credentials. A **credential** is a set of properties concerning a user that are relevant for security purposes (for example, age or position or role within an organization). For instance, by using credentials, one cansimply formulate policies such as Only permanent staff with five or more years of service can access documents related to the internals of the system.

XML is expected to play a key role in access control for e-commerce applications[6] because XML is becoming the common representation language for document interchange over the Web, and is also becoming the language for e-commerce. Thus, on the one hand, there is the need to make XML representations secure by providing access control mechanisms specifically tailored to the protection of XML documents. On the other hand, access control information (that is, access control policies and user credentials) can be expressed using XML itself. The **Directory Services Markup Language (DSML)** is a representation of directory service information in XML syntax. It provides a foundation for a standard for communicating with the directory services that will be responsible for providing and authenticating user credentials. The uniform presentation of both protection objects and access control policies can be applied to policies and credentials them- selves. For instance, some credential properties (such as the user name) may be accessible to everyone, whereas other properties may be visible only to a restricted class of users. Additionally, the use of an XML-based language for specifying credentials and access control policies facilitates secure credential submission and export of access control policies.

15. Explain in detail about SQL Injection.

SQL injection is one of the most common threats to a database system. Some of the other frequent attacks on databases are:

- **Unauthorized privilege escalation.** This attack is characterized by an indi-vidual attempting to elevate his or her privilege by attacking vulnerable points in the database systems.
- **Privilege abuse.** Whereas unauthorized privilege escalation is done by an unauthorized user, this attack is performed by a privileged user. For exam- ple, an administrator who is allowed to change student information can use this privilege to update student grades without the instructor's permission.
- **Denial of service.** A **denial of service (DOS) attack** is an attempt to makeresources unavailable to its intended users. It is a general attack category in which access to network applications or data is denied to intended users by overflowing the buffer or consuming resources.
- **Weak authentication.** If the user authentication scheme is weak, an attacker can impersonate the identity of a legitimate user by obtaining her login credentials.

SQL Injection Methods

Web programs and applications that access a database can send commands and data to the database, as well as display data retrieved from the database through the Web browser. In an **SQL injection attack**, the attacker injects a string input through the application, which changes or manipulates the SQL statement to the attacker's advantage. An SQL injection attack can harm the database in various ways, such as unauthorized manipulation of the data-base or retrieval of sensitive data. It can also be used to execute system-level commands that may cause the system to deny service to the application.

Types of injection attacks are

1) **SQL Manipulation**. A manipulation attack, which is the most common type of injection attack, changes an SQL command in the application–for example, by add- ing conditions to the WHERE-clause of a query, or by expanding a query with addi- tional query components using set operations such as UNION, INTERSECT, or MINUS. Other types of manipulation attacks are also possible. A typical manipulation attack occurs during database login. For example, suppose that a simplistic authentication procedure issues the following query and checks to see if any rows were returned:

```
SELECT * FROM users WHERE username = 'jake' and PASSWORD ='jakespasswd' ;
```

The attacker can try to change (or manipulate) the SQL statement by changing it asfollows:

```
SELECT * FROM users WHERE username = 'jake' and (PASSWORD ='jakespasswd' or 'x' = 'x') ;
```

As a result, the attacker who knows that 'jake' is a valid login of some user is able tolog into the database system as 'jake' without knowing his password and is able todo everything that 'jake' may be authorized to do to the database system.

2) **Code Injection**. : This type of attack attempts to add additional SQL statements or commands to the existing SQL statement by exploiting a computer bug, which is caused by processing invalid data. The attacker can inject or introduce code into a computer program to change the course of execution. Code injection is a popular technique for system hacking or cracking to gain information.

3) **Function Call Injection**. In this kind of attack, a database function or operating system function call is inserted into a vulnerable SQL statement to manipulate the data or make a privileged system call. For example, it is possible to exploit a func- tion that performs some aspect related to network communication. In addition, functions that are contained in a customized database package, or any custom data- base function, can be executed as part of an SQL query. In particular, dynamicallycreated SQL queries (see Chapter 10) can be exploited since they are constructedat runtime.

For example, the dual table is used in the FROM clause of SQL in Oracle when a userneeds to run SQL that does not logically have a table name. To get today's date, wecan use:

```
SELECT SYSDATE FROM dual;
```

The following example demonstrates that even the simplest SQL statements can bevulnerable.

```
SELECT TRANSLATE ('user input', 'from_string', 'to_string') FROM dual;
```

Here, TRANSLATE is used to replace a string of characters with another stringof characters. The TRANSLATE function above will replace the characters of the 'from_string' with the characters in the 'to_string' one by one. This means that thef will be replaced with the t, the r with the o, the o with the _, and so on.

This type of SQL statement can be subjected to a function injection attack. Con- sider the following example:

```
SELECT TRANSLATE (" || UTL_HTTP.REQUEST ('http://129.107.2.1/') || ",'98765432', '9876') FROM dual;
```

The user can input the string (" || UTL_HTTP.REQUEST ('http://129.107.2.1/') ||"), where || is the concatenate operator, thus requesting a page from a Web server. UTL_HTTP makes Hypertext Transfer Protocol (HTTP) callouts from SQL. The REQUEST object takes a URL ('http://129.107.2.1/' in this example) as a parameter, contacts that site, and returns the data (typically HTML) obtained from that site. The attacker could manipulate the string he inputs, as well as the URL, to include other functions and do other illegal operations. We just used a dummy example to show conversion of '98765432' to '9876', but the user's intent would beto access the URL and get sensitive information. The attacker can then retrieve useful information from the database server–located at the URL that is passed as a parameter–and send it to the Web server (that calls the TRANSLATE function).

Risks Associated with SQL Injection

SQL injection is harmful and the risks associated with it provide motivation for attackers. Some of the risks associated with SQL injection attacks are explained below.

- **Database fingerprinting.** The attacker can determine the type of database being used in the backend so that he can use database-specific attacks that correspond to weaknesses in a particular DBMS.
- **Denial of service.** The attacker can flood the server with requests, thus denying service to valid users, or the attacker can delete some data.
- **Bypassing authentication.** This is one of the most common risks, in whichthe attacker can gain access to the database as an authorized user and per- form all the desired tasks.
- **Identifying injectable parameters.** In this type of attack, the attacker gathers important information about the type and structure of the back-end database of a Web application. This attack is made possible by the fact that the default error page returned by application servers is often overly descriptive.
- **Executing remote commands.** This provides attackers with a tool to exe- cute arbitrary commands on the database. For example, a remote user can execute stored database procedures and functions from a remote SQL inter-active interface.
- **Performing privilege escalation.** This type of attack takes advantage of logical flaws within the database to upgrade the access level.

Protection Techniques against SQL Injection

Protection against SQL injection attacks can be achieved by applying certain programming rules to all Web-accessible procedures and functions. This section describes some of these techniques.

Bind Variables (Using Parameterized Statements). The use of bind variables (also known as parameters) protects against injection attacks and also improves performance.

Consider the following example using Java and JDBC: PreparedStatement stmt = conn.prepareStatement("SELECT * FROM
EMPLOYEE WHERE EMPLOYEE_ID=? AND PASSWORD=?");

stmt.setString(1, employee_id);stmt.setString(2, password);

Instead of embedding the user input into the statement, the input should be bound to a parameter. In this example, the input '1' is assigned (bound) to a bind variable

'employee_id' and input '2' to the bind variable 'password' instead of directly passing string parameters.

Filtering Input (Input Validation). This technique can be used to remove escape characters from input strings by using the SQL Replacefunction. For example, thedelimiter single quote (') can be replaced by two single quotes (''). Some SQL manipulation attacks can be prevented by using this technique, since escape characters can be used to inject manipulation attacks. However, because there can be a large number of escape characters, this technique is not reliable.

Function Security. Database functions, both standard and custom, should be restricted, as they can be exploited in the SQL function injection attacks.

16. Explain about StatisticalDatabase Security.

Statistical databases are used mainly to produce statistics about various populations. The database may contain confidential data about individuals; this information should be protected from user access. However, users are permitted to retrieve statistical information about the populations, such as averages, sums, counts, maximums, minimums, and standard deviations. The techniques that have been developed to protect the privacy of individual information are beyond the scope of this text. We will illustrate the problem with a very simple example, which refers to therelation shown in Figure. This is a PERSON relation with the attributes Name, Ssn, Income, Address, City, State, Zip, Sex, and Last_degree.

A **population** is a set of tuples of a relation (table) that satisfy some selection condition. Hence, each selection condition on the PERSON relation will specify a particular population of PERSON tuples. For example, the condition Sex = 'M' specifies the male population; the condition ((Sex = 'F') AND (Last_degree = 'M.S.' OR Last_degree= 'Ph.D.')) specifies the female population that has an M.S. or Ph.D. degree as theirhighest degree; and the condition City = 'Houston' specifies the population that lives in Houston.

Statistical queries involve applying statistical functions to a population of tuples. For example, we may want to retrieve the number of individuals in a population or the average income in the population. However, statistical users are not allowed to retrieve individual data, such as the income of a specific person. **Statistical database security** techniques must prohibit the retrieval of individual data. This can be achieved by prohibiting queries that retrieve attribute values and by allowing

---

Figure . The PERSON relation schema for illustrating statistical database security.

PERSON

Name	Ssn	Income	Address	City	State	Zip	Sex	Last_degree

only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called **statistical queries**.

It is the responsibility of a database management system to ensure the confidentiality of information about individuals while still providing useful statistical summaries of data about those individuals to users. Provision of **privacy protection** of users in astatistical database is paramount; its violation is illustrated in the following example.

In some cases it is possible to **infer** the values of individual tuples from a sequence of statistical queries. This is particularly true when the conditions result in a population consisting of a small number of tuples. As an illustration, consider the following statistical queries:

Q1: SELECT COUNT (*)FROM PERSON

WHERE <condition>;

Q2: SELECT AVG (Income) FROM PERSON

WHERE <condition>;

Now suppose that we are interested in finding the Salary of Jane Smith, and we know that she has a Ph.D. degree and that she lives in the city of Bellaire, Texas. We issue the statistical query Q1 with the following condition:

(Last_degree='Ph.D.' AND Sex='F' AND City='Bellaire' AND State='Texas')

If we get a result of 1 for this query, we can issue Q2 with the same condition and find the Salary of Jane Smith. Even if the result of Q1 on the preceding condition is not 1 but is a small number—say 2 or 3—we can issue statistical queries using the functions MAX, MIN, and AVERAGE to identify the possible range of values for the Salary of Jane Smith.

The possibility of inferring individual information from statistical queries is reduced if no statistical queries are permitted whenever the number of tuples in the population specified by the selection condition falls below some threshold. Another technique for prohibiting retrieval of individual information is to prohibit sequences of queries that refer repeatedly to the same population of tuples. It is also possible to introduce slight inaccuracies or noise into the results of statistical queries deliberately, to make it difficult to deduce individual information from the results. Another technique is partitioning of the database. Partitioning implies that records are stored in groups of some minimum size; queries can refer to any complete group or set of groups, but never to subsets of records within a group. The interested reader is referred to the bibliography at the end of this chapter for a discussion of these techniques.

17. Elaborate about Flow Control.

Flow control regulates the distribution or flow of information among accessible objects. A flow between object X and object Y occurs when a program reads values from X and writes values into Y. **Flow controls** check that information contained in some objects does not flow explicitly or implicitly into less protected objects. Thus, a user cannot get indirectly in Y what he or she cannot get directly in X. Active flow control began in the early 1970s. Most flow controls employ some concept of security class; the transfer of information from a sender to a receiver is allowed only if the receiver's security class is at least as privileged as the sender's. Examples of a flow control include preventing a service program from leaking a customer's confidential data, and blocking the transmission of secret military data to an unknown classified user.

A **flow policy** specifies the channels along which information is allowed to move. The simplest flow policy specifies just two classes of information—confidential (C) and nonconfidential (N)—and allows all flows except those from class C to class N. This policy can solve the confinement problem that arises when a service program handles data such as customer information, some of which may be confidential. For example, an income-tax-computing service might be allowed to retain a customer's address and the bill for services rendered, but not a customer's income or deductions.

Access control mechanisms are responsible for checking users' authorizations for resource access: Only granted operations are executed. Flow controls can be enforced by an extended access control mechanism, which involves assigning a security class (usually called the clearance) to each running program. The program is allowed to read a particular memory segment only if its security class is as high as that of the segment. It is allowed to write in a segment only if

its class is as low as that of the segment. This automatically ensures that no information transmitted by the person can move from a higher to a lower class. For example, a military program with a secret clearance can only read from objects that are unclassified and confidential and can only write into objects that are secret or top secret.

Two types of flow can be distinguished: explicit flows, which occur as a consequence of assignment instructions, such as $Y:= f(X_1,X_n,)$; and implicit flows, which are generated by conditional instructions, such as if $f(X_{m+1, \dots, } X_n)$ then $Y:= f (X_1,X_m)$.

Flow control mechanisms must verify that only authorized flows, both explicit and implicit, are executed. A set of rules must be satisfied to ensure secure information flows. Rules can be expressed using flow relations among classes and assigned to information, stating the authorized flows within a system. (An information flow from A to B occurs when information associated with A affects the value of information associated with B. The flow results from operations that cause information transfer from one object to another.) These relations can define, for a class, the set of classes where information (classified in that class) can flow, or can state the specific relations to be verified between two classes to allow information to flow from one to the other. In general, flow control mechanisms implement the controls by assigning a label to each object and by specifying the security class of the object. Labels are then used to verify the flow relations defined in the model.

Covert Channels

A covert channel allows a transfer of information that violates the security or the policy. Specifically, a **covert channel** allows information to pass from a higher classification level to a lower classification level through improper means. Covert channels can be classified into two broad categories: timing channels and storage. The distinguishing feature between the two is that in a **timing channel** the information is conveyed by the timing of events or processes, whereas **storage channels** do not require any temporal synchronization, in that information is conveyed by accessing system information or what is otherwise inaccessible to the user.

In a simple example of a covert channel, consider a distributed database system in which two nodes have user security levels of secret (S) and unclassified (U). In order for a transaction to commit, both nodes must agree to commit. They mutually can only do operations that are consistent with the *-property, which states that in any transaction, the S site cannot write or pass information to the U site. However, if these two sites collude to set up a covert channel between them, a transaction involving secret data may be committed unconditionally by the U site, but the S site may do so in some predefined agreed-upon way so that certain information may be passed from the S site to the U site, violating the *-property. This may be achieved where the transaction runs repeatedly, but the actions taken by the S site implicitly convey information to the U site. Measures such as locking, prevent concurrent writing of the information by users with different security levels into the same objects, preventing the storage-type covert channels. Operating systems and distributed databases provide control over the multiprogramming of operations, which allows a sharing of resources without the possibility of encroachment of one program or process into another's memory or other resources in the system, thus preventing timing-oriented covert channels. In general, covert channels are not a major problem in well-implemented robust data- base implementations. However, certain schemes may be contrived by clever users that implicitly transfer information.

Some security experts believe that one way to avoid covert channels is to disallow programmers to actually gain access to sensitive data that a program will process after the program has been put into operation. For example, a programmer for a bank has no need to access the names or balances in depositors' accounts. Programmers for brokerage firms do not need to know what buy and sell orders exist for clients. During program testing, access to a form of real data or some sample test data may be justifiable, but not after the program has been accepted for regular use.

18. Explain about Encryption and Public Key Infrastructures.

The previous methods of access and flow control, despite being strong control measures, may not be able to protect databases from some threats. Suppose we communicate data, but our data falls into the hands of a non legitimate user. In this situation, by using encryption we can disguise the message so that even if the transmission is diverted, the message will not be revealed. **Encryption** is the conversion of data into a form, called a **cipher text**, that cannot be easily understood by unauthorized persons. It enhances security and privacy when access controls are bypassed, because in cases of data loss or theft, encrypted data cannot be easily understood by unauthorized persons.

With this background, we adhere to following standard definitions:

- Cipher text: Encrypted (enciphered) data
- Plaintext (or clear text): Intelligible data that has meaning and can be read oracted upon without the application of decryption
- Encryption: The process of transforming plaintext into ciphertext
- Decryption: The process of transforming ciphertext back into plaintext

Encryption consists of applying an **encryption algorithm** to data using some pre- specified **encryption key**. The resulting data must be **decrypted** using a **decryption key** to recover the original data.

The Data Encryption and AdvancedEncryption Standards

The **Data Encryption Standard (DES)** is a system developed by the U.S. government for use by the general public. It has been widely accepted as a cryptographic standard both in the United States and abroad. DES can provide end-to-end encryption on the channel between sender A and receiver B. The DES algorithm is a careful and complex combination of two of the fundamental building blocks of encryption: substitution and permutation (transposition). The algorithm derives its strength from repeated application of these two techniques for a totalof 16 cycles. Plaintext (the original form of the message) is encrypted as blocks of 64 bits. Although the key is 64 bits long, in effect the key can be any 56-bit number. After questioning the adequacy of DES, the NIST introduced the **Advanced Encryption Standard (AES)**. This algorithm has a block size of 128 bits, compared with DES's 56-block size, and can use keys of 128, 192, or 256 bits, com- pared with DES's 56-bit key. AES introduces more possible keys, compared with DES, and thus takes a much longer time to crack. In present systems, AES is the default with large key lengths. It is also the standard for full drive encryption products, with both Apple FileVault and Microsoft BitLocker using 256-bit or 128-bit keys. Triple DES is a fallback option if a legacy system cannot use a modern encryption standard.

Symmetric Key Algorithms

A symmetric key is one key that is used for both encryption and decryption. By using a symmetric key, fast encryption and decryption is possible for routineuse with sensitive data in the database. A message encrypted with a secret key can be decrypted only with the same secret key. Algorithms used for symmetric key encryption are called **secret key algorithms**. Since secret-key algorithms are mostly used for encrypting the content of a message, they are also called **content-encryption algorithms**.

The major liability associated with secret-key algorithms is the need for sharing the secret key. A possible method is to derive the secret key from a user-supplied password string by applying the same function to the string at both the sender and receiver; this is known as a password-based encryption algorithm. The strength of the symmetric key encryption depends on the size of the key

used. For the same algorithm, encryptingusing a longer key is tougher to break than the one using a shorter key.

Public (Asymmetric) Key Encryption

In 1976, Diffie and Hellman proposed a new kind of cryptosystem, which they called **public key encryption**. Public key algorithms are based on mathematical functions rather than operations on bit patterns. They address one drawback of symmetric key encryption, namely that both sender and recipient must exchange the common key in a secure manner. In public key systems, two keys are used for encryption/decryption. The public key can be transmitted in a nonsecure way, whereas the private key is not transmitted at all. These algorithms–which use tworelated keys, a public key and a private key, to perform complementary operations (encryption and decryption)–are known as **asymmetric key encryption algorithms**. The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication. The two keys used for public key encryption are referred to as the **public key** and the **private key**. The private key is kept secret, but it is referred to as a private key rather than a secret key (the key used in conventional encryption) to avoid confusion with conventional encryption. The two keys are mathematically related, since one of the keys is used to perform encryption and the other to perform decryption. However, it is very difficult to derive the private key from the public key.

A public key encryption scheme, or infrastructure, has six ingredients:

1. **Plaintext.** This is the data or readable message that is fed into the algorithmas input.
2. **Encryption algorithm.** This algorithm performs various transformations on the plaintext.
3. **Public and private keys.** These are a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input. For example, if a message is encrypted using the public key, it can only be decrypted using the private key.
5. **Ciphertext.** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
6. **Decryption algorithm.** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

The public key of the pair is made public for others to use, whereas the private key is known only to its owner. A general-purpose public key cryptographic algorithm relies on one key for encryption and a different but relatedkey for decryption. The essential steps are as follows:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user places one of the two keys in a public register or other accessiblefile. This is the public key. The companion key is kept private.
3. If a sender wishes to send a private message to a receiver, the sender encryptsthe message using the receiver's public key.
4. When the receiver receives the message, he or she decrypts it using the receiver's private key. No other recipient can decrypt the message because only the receiver knows his or her private key.

The RSA Public Key Encryption Algorithm. One of the first public key schemeswas introduced in 1978 by Ron Rivest, Adi Shamir, and Len Adleman at MIT andis named after them as the **RSA scheme**. The RSA scheme has since then reigned supreme as the most widely accepted and implemented approach to public key encryption. The RSA encryption algorithm incorporates results from number theory, combined with the difficulty of determining the prime factors of a target. The RSA algorithm also operates with modular arithmetic–mod n.

Two keys, d and e, are used for decryption and encryption. An important property is that they can be interchanged. n is chosen as a large integer that is a product of two large distinct prime numbers, a and b, $n = a \times b$. The encryption key e is a randomly chosen number between 1 and n that is relatively prime to $(a - 1) \times (b - 1)$. The plaintext block P is encrypted as P^e where $P^e = P$

mod n. Because the exponentiation is performed mod n, factoring P^e to uncover the encrypted plaintext is difficult. However, the decrypting key d is carefully chosen so that $(P^e)^d \bmod n = P$. The decryption key d can be computed from the condition that $d \times e = 1 \bmod ((a - 1) \times (b - 1))$. Thus, the legitimate receiver who knows d simply computes $(P^e)^d \bmod n = P$ and recovers P without having to factor P^e.

Digital Signatures

A digital signature is an example of using encryption techniques to provide authentication services in electronic commerce applications. Like a handwritten signature, a **digital signature** is a means of associating a mark unique to an individual with a body of text. The mark should be unforgettable, meaning that others should be able to check that the signature comes from the originator.

A digital signature consists of a string of symbols. If a person's digital signature were always the same for each message, then one could easily counterfeit it by simply copying the string of symbols. Thus, signatures must be different for each use. This can be achieved by making each digital signature a function of the message that it is signing, together with a timestamp. To be unique to each signer and counterfeitproof, each digital signature must also depend on some secret number that is unique to the signer. Thus, in general, a counterfeitproof digital signature must depend on the message and a unique secret number of the signer. The verifier of the signature, however, should not need to know any secret number. Public key techniques are the best means of creating digital signatures with these properties.

Digital Certificates

A digital certificate is used to combine the value of a public key with the identity of the person or service that holds the corresponding private key into a digitally signed statement. Certificates are issued and signed by a certification authority (CA). The entity receiving this certificate from a CA is the subject of that certificate. Instead of requiring each participant in an application to authenticate every user, third-party authentication relies on the use of digital certificates.

The digital certificate itself contains various types of information. For example, both the certification authority and the certificate owner information are included. The following list describes all the information included in the certificate:

1. The certificate owner information, which is represented by a unique identi- fier known as the distinguished name (DN) of the owner. This includes the owner's name, as well as the owner's organization and other information about the owner.
2. The certificate also includes the public key of the owner.
3. The date of issue of the certificate is also included.
4. The validity period is specified by 'Valid From' and 'Valid To' dates, which are included in each certificate.
5. Issuer identifier information is included in the certificate.
6. Finally, the digital signature of the issuing CA for the certificate is included. All the information listed is encoded through a message-digest function, which creates the digital signature. The digital signature basically certifies that the association between the certificate owner and public key is valid.

Privacy Issues and Preservation

Preserving data privacy is a growing challenge for database security and privacy experts. In some perspectives, to preserve data privacy we should even limit performing large-scale data

mining and analysis. The most commonly used techniques to address this concern are to avoid building mammoth central warehouses as a single repository of vital information. This is one of the stumbling blocks for creating nationwide registries of patients for many important diseases. Another possible measure is to intentionally modify or perturb data.

If all data were available at a single warehouse, violating only a single repository's security could expose all data. Avoiding central warehouses and using distributed data mining algorithms minimizes the exchange of data needed to develop globally valid models. By modifying, perturbing, and anonymizing data, we can also mitigate privacy risks associated with data mining. This can be done by removing identity information from the released data and injecting noise into the data. However, by using these techniques, we should pay attention to the quality of the resulting data in the database, which may undergo too many modifications. We must be able to estimate the errors that may be introduced by these modifications.

Privacy is an important area of ongoing research in database management. It is complicated due to its multidisciplinary nature and the issues related to the subjectivity in the interpretation of privacy, trust, and so on. As an example, consider medical and legal records and transactions, which must maintain certain privacy requirements. Providing access control and privacy for mobile devices is also receiving increased attention. DBMSs need robust techniques for efficient storage of security-relevant information on small devices, as well as trust negotiation techniques. Where to keep information related to user identities, profiles, credentials, and permissions and how to use it for reliable user identification remains an important problem. Because large-sized streams of data are generated in such environments, efficient techniques for access control must be devised and integrated with processing techniques for continuous queries. Finally, the privacy of user location data, acquired from sensors and communication networks, must be ensured.

19. Explain about the Challenges to Maintaining Database Security.

Considering the vast growth in volume and speed of threats to databases and information assets, research efforts need to be devoted to a number of issues: data quality, intellectual property rights, and database survivability, to name a few.

1) Data Quality

The database community needs techniques and organizational solutions to assess and attest to the quality of data. These techniques may include simple mechanisms such as quality stamps that are posted on Web sites. We also need techniques that provide more effective integrity semantics verification and tools for the assessment of data quality, based on techniques such as record linkage. Application-level recovery techniques are also needed for automatically repairing incorrect data. The ETL (extract, transform, load) tools widely used to load data in data warehouses are presently grappling with these issues.

2) Intellectual Property Rights

With the widespread use of the Internet and intranets, legal and informational aspects of data are becoming major concerns for organizations. To address these concerns, watermarking techniques for relational data have been proposed. The main purpose of digital watermarking is to protect content from unauthorized duplication and distribution by enabling provable ownership of the content. Digital watermarking has traditionally relied upon the availability of a large noise domain within which the object can be altered while retaining its essential properties. How- ever, research is needed to assess the robustness of such techniques and to investi- gate different approaches aimed at preventing intellectual property rights violations.

3) Database Survivability

Database systems need to operate and continue their functions, even with reduced capabilities,

despite disruptive events such as information warfare attacks. A DBMS,in addition to making every effort to prevent an attack and detecting one in the event of occurrence, should be able to do the following:

- **Confinement.** Take immediate action to eliminate the attacker's access tothe system and to isolate or contain the problem to prevent further spread.
- **Damage assessment.** Determine the extent of the problem, including failed functions and corrupted data.
- **Reconfiguration.** Reconfigure to allow operation to continue in a degradedmode while recovery proceeds.
- **Repair.** Recover corrupted or lost data and repair or reinstall failed system functions to reestablish a normal level of operation.
- **Fault treatment.** To the extent possible, identify the weaknesses exploited inthe attack and take steps to prevent a recurrence.

The goal of the information warfare attacker is to damage the organization's operation and fulfillment of its mission through disruption of its information systems. The specific target of an attack may be the system itself or its data. Although attacksthat bring the system down outright are severe and dramatic, they must also be welltimed to achieve the attacker's goal, since attacks will receive immediate and concentrated attention in order to bring the system back to operational condition, diagnose how the attack took place, and install preventive measures.

Unit-5

Part-A

1. What Constitutes a DDB

For a database to be called distributed, the following minimum conditions should be satisfied:

- **Connection of database nodes over a computer network.** There are mul-tiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **network** to transmit data and commands among sites.
- **Logical interrelation of the connected databases.** It is essential that the information in the various database nodes be logically related.
- **Possible absence of homogeneity among connected nodes.** It is not neces-sary that all nodes be identical in terms of data, hardware, and software.

2. Define distributed database.

Distributed database (DDB) as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.

3. What are two main types of multiprocessor system architectures?
   - **Shared memory (tightly coupled) architecture.** Multiple processors share secondary (disk) storage and also share primary memory.
   - **Shared disk (loosely coupled) architecture.** Multiple processors share sec-ondary (disk) storage but each has their own primary memory.

4. List the various stages of distributed database query processing.

   Various stages of distributed database query processing are

   a) Query Mapping
   b) Localization.
   c) Global Query Optimization
   d) Local Query Optimization

5. Write down the differences between NoSQL and RDBMS?

Following is a list of the differences between NoSQL and RDBMS: –

- In terms of data format, NoSQL does not follow any order for its data format. Whereas, RDBMS is more organized and structured when it comes to the format of its data.
  When it comes to scalability, NoSQL is more very good and more scalable. Whereas, RDBMS is average and less scalable than NoSQL.
  For querying of data, NoSQL is limited in terms of querying because there is no join clause present in NoSQL. Whereas, querying can be used in RDBMS as it uses the structured query language.
  The difference in the storage mechanism of NoSQL and RDBMS, NoSQL uses key-value pairs, documents, column storage, etc. for storage. Whereas, RDBMS uses various tables for storing data and relationships.

6. What do you understand by NoSQL in databases?

The database management systems which are highly scalable and flexible are known as NoSQL databases. These databases allow us to store and process unstructured and semi-structured data which is not possible when we make use of the Relational database management system. NoSQL can be termed as a solution to all the conventional databases which were not able to handle the data seamlessly. It also gives an opportunity to the companies to store massive amounts of structured and unstructured data in real-time. In today's time, big firms such as- Google, Facebook, Amazon, etc. use NoSQL for providing cloud-based services for storing data in real-time.

7. List some of the features of NoSQL?

Some of the features of NoSQL are listed below: –

Using NoSQL, we can store a large amount of structured, semi-structured, and unstructured data.
It supports agile sprint, quick iteration, and frequent code pushes.
It uses object-oriented programming which is frequent and is also easy to use.
It is more efficient. It has a scale-out architecture. It is cheap instead of being expensive. It has a monolithic architecture. It can be easily accessed.

8. What are the pros and cons of a graph database under NoSQL databases?

Following are the pros and cons of a graph database which is a type of NoSQL databases: –
Pros of using graph database:
These are tailor-made for networking applications. A social network is a good example of this.
They can also be perfect for an object-oriented programming system.
Cons of using graph database:
Since the degree of interconnection between nodes is high in the graph database, so it is not suitable for network partitioning.
Also, graph databases don't scale out well in NoSQL databases.

9. List the different kinds of NoSQL data stores?

The variety of NoSQL data stores available which are widely distributed are categorized into four categories. They are: –
Key-value store– it is a simple data storage key system that uses keys to access different values.
Column family store– it is a sparse matrix system. It uses columns and rows as keys.
Graph store– it is used in case of relationships-intensive problems.
Document stores- it is used for storing hierarchical data structures directly in the database.

10. What is the CAP Theorem? How is it applicable to NoSQL systems?

The CAP theorem was proposed by Eric Brewer in early 2000. In this, three system attributes have been discussed within the distributed databases. That is-
Consistency- in this, all the nodes see the same data at the same time.
Availability- it gives us a guarantee that there will be a response for every request made to the system about whether it was successful or not.
Partition tolerance- it is the quality of the NoSQL database management system which states that the system will work even if a part of the system has failed or is not working.
A distributed database system might provide only 2 of the 3 above qualities.

11. What do you mean by eventual consistency in NoSQL stores?

Eventual consistency in NoSQL means that when all the service logics have been executed, the system is left in a consistent state. For achieving high availability, this concept is used in the distributed systems. It gives a guarantee that, if new updates are not made to a given data item, then eventually all accesses to that item will return the last updated value. In NoSQL, it is provided in terms of BASE and RDMS are also known as the ACID properties. Present NoSQL databases provide client applications with a guarantee of eventual consistency. Some NoSQL databases like- MongoDB and Cassandra are eventually consistent in some of the configurations.

12. What are the different types of NoSQL databases? Give some examples.

NoSQL database can be classified as 4 basic types:

1. Key-value store NoSQL database
2.Document store NoSQL database
3.Column store NoSQL database
4. Graph-based NoSQL database

There are many NoSQL databases. MongoDB, Cassandra, CouchDB, Hypertable, Redis, Riak, Neo4j, HBase, Couchbase, MemcacheDB, Voldemort, RevenDB, etc. are examples of NoSQL databases.

13. Is MongoDB better than other SQL databases? If yes then how?

MongoDB is better than other SQL databases because it allows a highly flexible and scalable document structure.

For example:
One data document in MongoDB can have five columns and the other one in the same collection can have ten columns.
MongoDB database is faster than SQL databases due to efficient indexing and storage techniques.

14. Why MongoDB is known as the best NoSQL database?

MongoDB is the best NoSQL database because it is:

1. Document Oriented
2. Rich Query language
3. High Performance
4. Highly Available
5. Easily Scalable

15. Explain the structure of ObjectID in MongoDB.

ObjectID is a 12-byte BSON type. These are:

1. 4 bytes value representing seconds
2. 3-byte machine identifier
3. 2-byte process id
4.3 byte counter

16. What are Indexes in MongoDB?

In MongoDB, Indexes are used to execute queries efficiently. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

17. What is a Namespace in MongoDB?

A namespace is a concatenation of the database name and the collection name. Collection, in which MongoDB stores BSON objects.

18. How to do Transaction/locking in MongoDB?

MongoDB doesn't use traditional locking or complex transaction with Rollback. MongoDB is designed to be lightweight, fast and predictable to its performance. It keeps transaction support simply to enhance performance.

34) Why 32-bit version of MongoDB is not preferred?

Because MongoDB uses memory-mapped files so when you run a 32-bit build of MongoDB, the total storage size of the server is 2 GB. But when you run a 64-bit build of MongoDB, this provides virtually unlimited storage size. So 64-bit is preferred over 32-bit.

19. What is the importance of covered queries?

Covered query makes the execution of the query faster because indexes are stored in RAM or sequentially located on disk. It makes the execution of the query faster.
Covered query makes the fields are covered in the index itself, MongoDB can match the query condition as well as return the result fields using the same index without looking inside the documents.

20. What is sharding in MongoDB?

In MongoDB, Sharding is a procedure of storing data records across multiple machines. It is a MongoDB approach to meet the demands of data growth. It creates a horizontal partition of data in a database or search engine. Each partition is referred to as a shard or database shard.

21. What is a replica set in MongoDB?

A replica can be specified as a group of mongo instances that host the same data set. In a replica set, one node is primary, and the other is secondary. All data is replicated from primary to secondary nodes.

22. What is the primary and secondary replica set in MongoDB?

In MongoDB, primary nodes are the nodes that can accept write. These are also known as master nodes. The replication in MongoDB is a single master so, only one node can accept write operations at a time. Secondary nodes are known as slave nodes. These are read-only nodes that replicate from the primary.

23. What is CRUD in MongoDB?

MongoDB supports following CRUD operations:

1. Create
2. Read
3. Update
4. Delete

24. What is DynamoDB?

- Amazon DynamoDB is a fully managed proprietary NoSQL database service that provides fast and predictable performance with seamless scalability.
- It supports key-value and document data structures
- DynamoDB allows users to create tables for your database to store and retrieve any data volume and to support any request traffic level.

25. What is NO-SQL Key-Value Stores?

- Key-value stores focus on high performance, availability, and scalability by storing data in a distributed storage system. The data model used in key-value stores is relatively simple, and in many of these systems, there is no query language but rather a set of operations

that can be used by the application programmers.

- The **key** is a unique identifier associated with a data item and is used to locate this data item rapidly.
- The **value** is the data item itself, and it can have very different formats for different key-value storage systems. In some cases, the value is just a *string of bytes* or an *array of bytes*, and the application using the key-value store has to interpret the structure of the data value. In other cases, some standard formatted data is allowed; for example, structured data rows (tuples) similar to relational data, or semistructured data using JSON or some other self-describing data format. Different key-value stores can thus store unstructured, semistructured, or structured data items. The main characteristic of key-value stores is the fact that every value (data item) must be associated with a unique key, and that retrieving the value by supplying the key must be very fast.

26. What is BigTable and HBase?

BigTable (and Hbase) is sometimes described as a *sparse multidimensional distributed persistent sorted map*, where the word *map* means a *collection of (key, value) pairs* (the key is *mapped* to the value). One of the main differences that distinguish column-based systems from key-value stores is the *nature of the key*. In column-based systems such as Hbase, the key is *multidimensional* and so has several components: typically, a combination of table name, row key, column, and timestamp. As we shall see, the column is typically composed of two components: column family and column qualifier. We discuss these concepts in more detail next as they are realized in Apache Hbase.

27. What is graph database?

In **graph databases** or **graph- oriented NOSQL** systems, the data is represented as a graph, which is a collection of vertices (nodes) and edges. Both nodes and edges can be labeled to indicate the types of entities and relationships they represent, and it is generally possible to store data associated with both individual nodes and individual edges. Many systems can be categorized as graph databases. We will focus our discussion on one particular system, Neo4j, which is used in many applications. Neo4j is an open source system, and it is implemented in Java.

28. What is difference between MongoDB and DynamoDB?

MongoDB	*DynamoDB*
MongoDB is designed and developed by MongoDB Inc.	DynamoDB is designed and developed by Amazon.com.
MongoDB uses JSON-like documents.	DynamoDB uses tables, items and attributes.
MongoDB is difficult to configure and install it. That is because it does not have provide guidance to perform. In order to make it easier to use MongoDB, they have provided MongoDB Atlas which is a cloud hosted.	Setting up and installation of DynamoDB is very easy since it is a web service provided by Amazon.com.
MongoDB supports more data types and has fewer size restrictions.	DynamoDB supports limited data types and smaller item

	sizes.
MongoDB is not so secure, because by default it keeps authentication off while installing MongoDB.	DynamoDB security is better and is usually supported by the security measure provided by AWS.

29. Define **SQL injection attack.**

In an **SQL injection attack**, the attacker injects a string input through the application, which changes or manipulates the SQL statement to the attacker's advantage. An SQL injection attack can harm the database in various ways, such as unauthorized manipulation of the data-base or retrieval of sensitive data. It can also be used to execute system-level commands that may cause the system to deny service to the application.

30. What are Types of injection attacks?

Types of injection attacks are

- SQL Manipulation
- Code Injection
- Function Call Injection

31. What are the Risks Associated with SQL Injection?

SQL injection is harmful and the risks associated with it provide motivation for attackers. Some of the risks associated with SQL injection attacks are explained below.

- **Database fingerprinting.** The attacker can determine the type of database being used in the backend so that he can use database-specific attacks that correspond to weaknesses in a particular DBMS.
- **Denial of service.** The attacker can flood the server with requests, thus denying service to valid users, or the attacker can delete some data.
- **Bypassing authentication.** This is one of the most common risks, in whichthe attacker can gain access to the database as an authorized user and per- form all the desired tasks.
- **Identifying injectable parameters.** In this type of attack, the attacker gathers important information about the type and structure of the back-end database of a Web application. This attack is made possible by the fact that the default error page returned by application servers is often overly descriptive.
- **Executing remote commands.** This provides attackers with a tool to exe- cute arbitrary commands on the database. For example, a remote user can execute stored database procedures and functions from a remote SQL inter-active interface.

Performing privilege escalation. This type of attack takes advantage of logical flaws within the database to upgrade the access level.

32. What is flow control?

Flow control regulates the distribution or flow of information among accessible objects. A flow between object X and object Y occurs when a program reads values from X and writes values into Y. **Flow controls**

check that information contained in some objects does not flow explicitly or implicitly into less protected objects. Thus, a user cannot get indirectly in Y what he or she cannot get directly in X.

33. What is covert channel?

C**overt channel** allows information to pass from a higher classification level to a lower classification level through improper means. Covert channels can be classified into two broad categories: timing channels and storage. The distinguishing feature between the two is that in a **timing channel** the information is conveyed by the timing of events or processes, whereas **storage channels**do not require any temporal synchronization, in that information is conveyed by accessing system information or what is otherwise inaccessible to the user.

34. Define secret key algorithms.

A symmetric key is one key that is used for both encryption and decryption. By using a symmetric key, fast encryption and decryption is possible for routineuse with sensitive data in the database. A message encrypted with a secret key can be decrypted only with the same secret key. Algorithms used for symmetrickey encryption are called **secret key algorithms**.

35. Define **asymmetric key encryption algorithms.**

Public key algorithms are based on mathematical functions rather than operations on bit patterns. They address one drawback of symmetric key encryption, namely that both sender and recipient must exchange the common key in a secure manner. In public key systems, two keys are used for encryption/decryption. The public key can be transmitted in a nonsecure way, whereas the private key is not transmitted at all. These algorithms—which use tworelated keys, a public key and a private key, to perform complementary operations (encryption and decryption)—are known as **asymmetric key encryption algorithms**.

36. Define digital certificate.

A digital certificate is used to combine the value of a public key with the identity ofthe person or service that holds the corresponding private key into a digitally signedstatement. Certificates are issued and signed by a certification authority (CA). Theentity receiving this certificate from a CA is the subject of that certificate. Instead ofrequiring each participant in an application to authenticate every user, third-party authentication relies on the use of digital certificates.

37. Define plaintext, cipher text, private key , public key, encryption algorithm, decryption algorithm.

- **Plaintext.** This is the data or readable message that is fed into the algorithmas input.
- **Encryption algorithm.** This algorithm performs various transformations on the plaintext.
- **Public and private keys.** These are a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input. For example, ifa message is encrypted using the public key, it can only be decrypted using the private key.
- **Ciphertext.** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
- **Decryption algorithm.** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

38 . What is digital signature?

Digital signature is a means of associating a mark unique to an individual with a body of text. The mark should be unforgettable, meaning that others should be able to check that the signature comes from the originator.

A digital signature consists of a string of symbols. If a person's digital signature were always the same for each message, then one could easily counterfeit it by simply copying the string of symbols. Thus, signatures must be different for each use.

Part-B

1. Explain about Distributed Database Concepts, characteristics and the advantages.
2. Discuss about Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design.
3. Explain about Transaction Management in Distributed Databases.
4. Explain about Query Processing and Optimization in Distributed Databases.
5. Explain about Distributed Database Architectures.
6. Explain about Emergence of NOSQL Systems, Characteristics of NOSQL Systems and types of NOSQL Systems.
7. Discuss about the CAP Theorem.
8. Explain about Document-Based NOSQL Systems and MongoDB.
9. Elaborate about NOSQL Key-Value Stores.
10. Explain about Column-Based or Wide Column NOSQL Systems.
11. Discuss about NOSQL Graph Databases and Neo4j.
12. Explain about Database Security Issues.
13. Discuss about Discretionary Access Control Based on Granting and Revoking Privileges.
14. Elaborate about Mandatory Access Control and Role-Based Access Control for Multilevel Security.
15. Explain in detail about SQL Injection.
16. Explain about Statistical Database Security.
17. Elaborate about Flow Control.
18. Explain about Encryption and Public Key Infrastructures.
19. Explain about the Challenges to MaintainingDatabase Security.