

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 programs. 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.

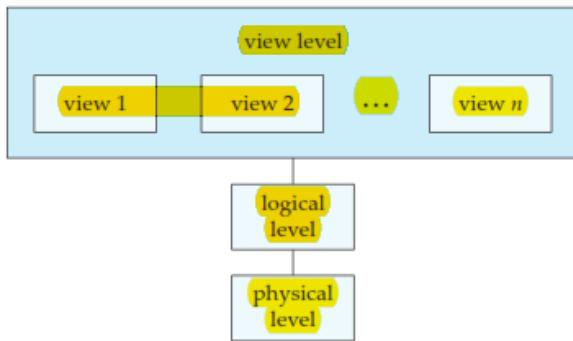


Figure .The three levels of data abstraction.

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.

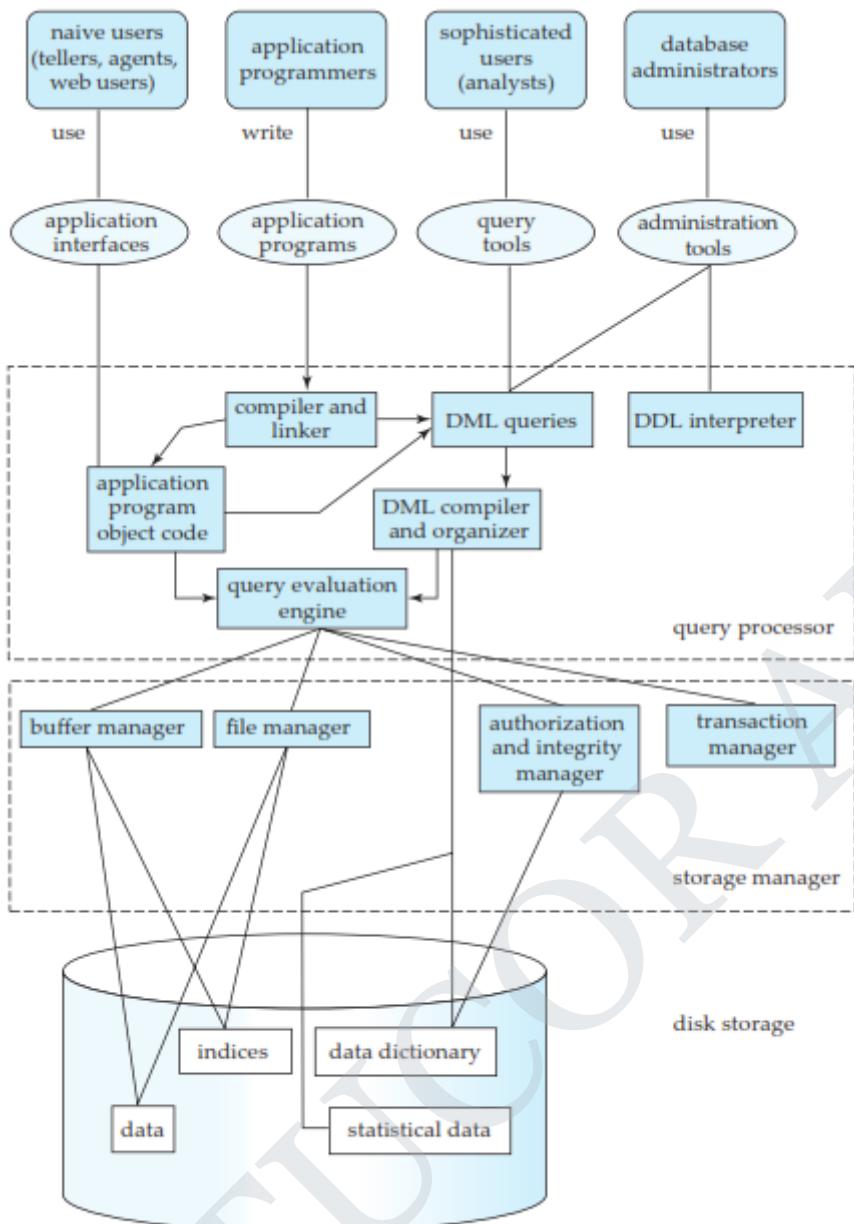


Figure 1.5 System structure.

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 manager 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:

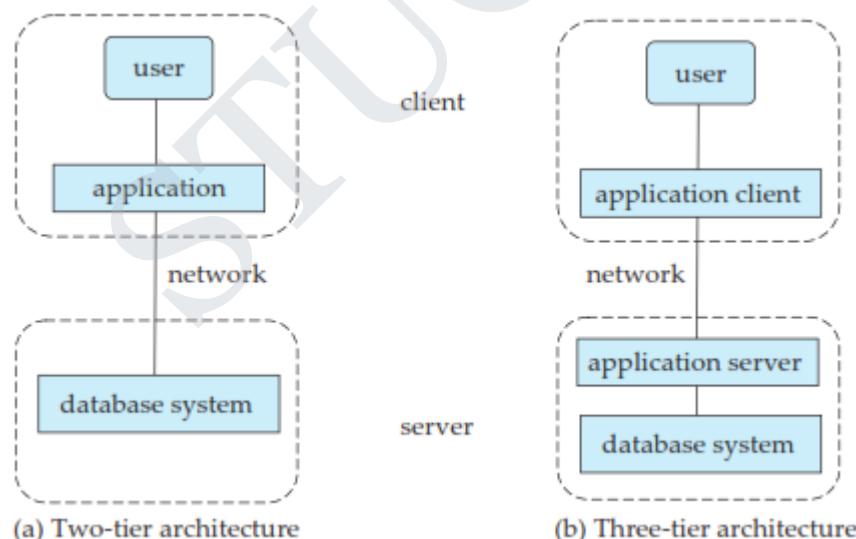


Figure .Two-tier and three-tier architectures.

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 enable an application programmer to construct forms and reports with minimal programming effort.
- 3) **Sophisticated users** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.
- 4) **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided

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

5) Database Administrator

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

- **Schema definition.** The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- **Storage structure and access-method definition.**
- **Schema and physical-organization modification.** The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- **Granting of authorization for data access.** By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- **Routine maintenance.** Examples of the database administrator's routine maintenance activities are:
 - ✓ Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
 - ✓ Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
 - ✓ Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

6. Explain about relational model.

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

Structure of Relational Databases:

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

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

- A row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In

mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between n values is represented mathematically by an n -*tuple* of values, i.e., a tuple with n values, which corresponds to a row in a table.

<i>ID</i>	<i>Name</i>	<i>dept_name</i>	<i>Salary</i>
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.

<i>course_id</i>	<i>Title</i>	<i>dept_name</i>	<i>Credits</i>
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

<i>Course_id</i>	<i>prereq_id</i>
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.

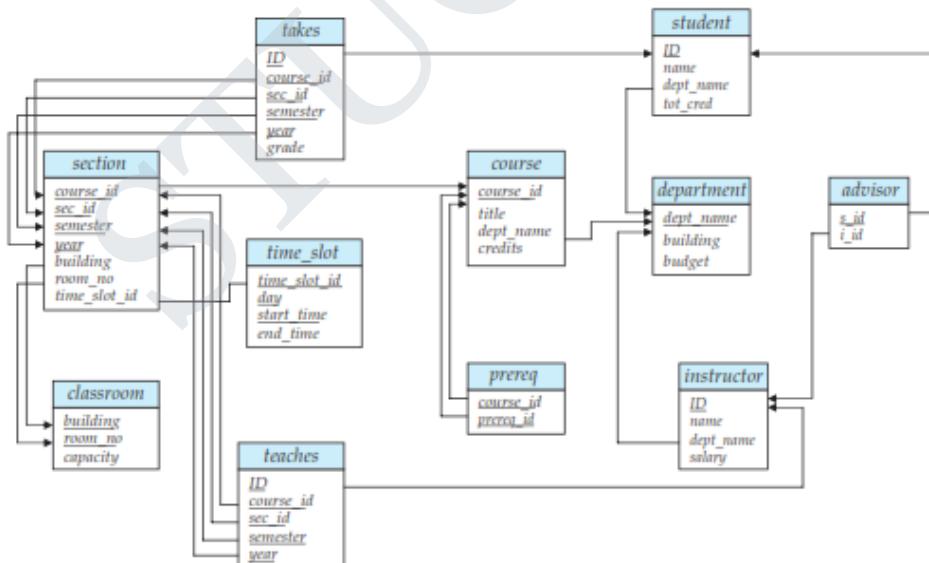


Figure. Schema diagram for the university database.

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 another 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 $=, \neq, \geq, <, >, \leq$.

For example: LOAN Relation

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

Input:

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

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

Where

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

Example: CUSTOMER RELATION

NAME	STREET	CITY
Jones	Main	Harrison
Smith	North	Rye

Input:

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

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 \cup .
- 1. Notation: $R \cup S$

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:

$$1. \prod \text{CUSTOMER_NAME} (\text{BORROW}) \cup \prod \text{CUSTOMER_NAME} (\text{DEPOSITOR})$$

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 \cap .
1. Notation: $R \cap S$

Example: Using the above DEPOSITOR table and BORROW table

Input:

$$1. \prod_{\text{CUSTOMER_NAME}} (\text{BORROW}) \cap \prod_{\text{CUSTOMER_NAME}} (\text{DEPOSITOR})$$

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:

$$1. \prod_{\text{CUSTOMER_NAME}} (\text{BORROW}) - \prod_{\text{CUSTOMER_NAME}} (\text{DEPOSITOR})$$

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: $E \times D$

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:

$$1. \text{EMPLOYEE} \times \text{DEPARTMENT}$$

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. $\rho(\text{STUDENT1}, \text{STUDENT})$

Join Operations:

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

Example:

EMPLOYEE

EMP_CODE	EMP_NAME
101	Stephan
102	Jack
103	Harry

SALARY

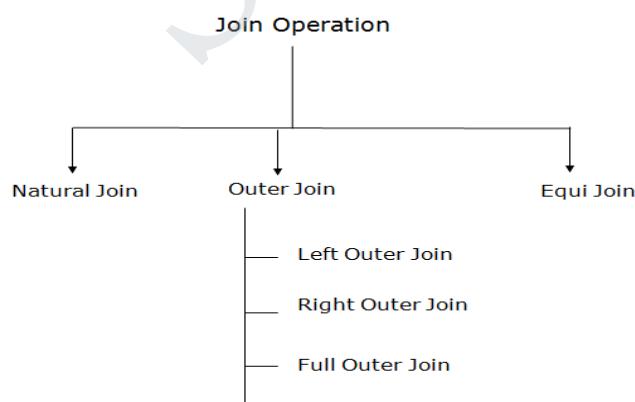
EMP_CODE	SALARY
101	50000
102	30000
103	25000

1. Operation: (EMPLOYEE \bowtie SALARY)

Result:

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

8. Types of Join operations:



1) Natural Join:

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

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

Input:

$$1. \prod_{\text{EMP_NAME}, \text{SALARY}} (\text{EMPLOYEE} \bowtie \text{SALARY})$$

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:

$$1. (\text{EMPLOYEE} \bowtie \text{FACT_WORKERS})$$

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 \bowtie_l .

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

$$1. \text{EMPLOYEE} \bowtie_l \text{FACT_WORKERS}$$

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 \bowtie .

Example: Using the above EMPLOYEE table and FACT_WORKERS Relation

Input:

1. EMPLOYEE \bowtie 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 \bowtie .

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

1. EMPLOYEE \bowtie FACT_WORKERS

Output:

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

3) Equi join:

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

Example:**CUSTOMER RELATION**

CLASS_ID	NAME
1	John
2	Harry
3	Jackson

PRODUCT

PRODUCT_ID	CITY
1	Delhi
2	Mumbai
3	Noida

Input:

CUSTOMER ↔ PRODUCT

Output:

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

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

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

The main categories are

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

i) **DDL (Data Definition Language)**

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

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

CREATE TABLE:

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

ALTER TABLE:

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

DROP TABLE

```
DROP TABLE employee;
```

CREATE INDEX

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

ALTER INDEX

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

DROP INDEX

```
DROP INDEX index_name;
```

CREATE VIEW

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

DROP VIEW

```
DROP VIEW emp_view;
```

ii) DML (Data Manipulation Language):

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

There are three basic DML commands:

INSERT:

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

UPDATE:

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

DELETE:

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

iii) DQL (Data Query Language)

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

SELECT:

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

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

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

iv) Data Control Language(DCL)

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

GRANT:

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

REVOKE:

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

CREATE SYNONYM:

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

v) Data Administration Commands

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

```
START AUDIT
STOP AUDIT
```

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

vi) Transactional Control Commands

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

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

9. Explain about nested sub queries.

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
 - **Example Query**
- Find courses offered in Fall 2009 and in Spring 2010
 - **select distinct course_id from section where semester = 'Fall' and year= 2009 and course_id in (select course_id from section where semester = 'Spring' and year= 2010);**
- Find courses offered in Fall 2009 but not in Spring 2010
 - **select distinct course_id from section where semester = 'Fall' and year= 2009 and course_id not in (select course_id from section where semester = 'Spring' and year= 2010);**

Example Query:

- Find the total number of (distinct) students who 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
from 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.
- $\text{Exists } r \Leftrightarrow r \neq \emptyset$
- $\text{not exists } r \Leftrightarrow r = \emptyset$
 - 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
 - `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 table1` would 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**

**10. Explain about join operations in SQL.**

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

**Join operations – Example**

- Relation *course*

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

- Relation *prereq*

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

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

**Outer Join**

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from 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       | <i>null</i> |

**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       | Join Conditions                  |
|------------------|----------------------------------|
| inner join       | natural                          |
| left outer join  | on <predicate>                   |
| right outer join | using ( $A_1, A_2, \dots, A_n$ ) |
| full outer join  |                                  |

**Full Outer Join**

□ □ *course natural full outer join prereq*

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

**Joined Relations – Examples**

- *course inner join prereq on course.course\_id = prereq.course\_id*

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

- What is the difference between the above, and a natural join?

- *course left outer join prereq on course.course\_id = prereq.course\_id*

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

**Joined Relations – Examples**

- *course natural right outer join prereq*

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

- *course full outer join prereq using (course\_id)*

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

### 11. Discuss about view in SQL.

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

SQL> CREATE TABLE EMPLOYEE (

|               |               |
|---------------|---------------|
| EMPLOYEE_NAME | VARCHAR2(10), |
| EMPLOYEE_NO   | NUMBER(8),    |
| DEPT_NAME     | VARCHAR2(10), |
| DEPT_NO       | NUMBER (5),   |
| DATE_OF_JOIN  | DATE          |

);

Table created.

### CREATE VIEW

#### SNTAX 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      |

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

1 ROW CREATED.

SQL>DROP VIEW EMPVIEW;  
view dropped

SQL> CREATE OR REPLACE VIEW EMP\_TOTSAL AS SELECT  
EMPNO "EID", ENAME "NAME", SALARY "SAL" FROM EMPL;  
**JOIN VIEW:**

#### EXAMPLE:

SQL> CREATE OR REPLACE VIEW DEPT\_EMP\_VIEW AS SELECT A.EMPNO,  
A.ENAME,  
A.DEPTNO,  
B.DNAME,  
B.LOC  
FROM EMPL A, DEPMT B  
WHERE  
A.DEPTNO=B.DEPTNO;

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

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

### **not null**

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

➤ **unique** (*A*<sub>1</sub>, *A*<sub>2</sub>, ..., *A*<sub>m</sub>)

- ✓ The unique specification states that the attributes *A*<sub>1</sub>, *A*<sub>2</sub>, ... *A*<sub>m</sub> 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.

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

Forms of authorization on parts of the database:

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

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

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

### Authorization Specification in SQL

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

### Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view  
Example: grant users  $U_1$ ,  $U_2$ , and  $U_3$  select authorization on the *instructor* relation:  
 $\text{grant select on instructor to } U_1, U_2, U_3$

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

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

#### 14. Explain about functions and procedures in SQL.

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

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

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

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

##### Program 1:

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

Procedure created.

```
SQL> execute p1;
welcome
```

##### Program 2:

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

Procedure created.

```
declare
a number;
b number;
c number;
begin
a:=23;
b:=4;
```

```

findMin(a,b,c);
dbms_output.put_line('Minimum value is:'||c);
end;
/
Output:
Minimum value is:4

```

**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

## 15. Explain about triggers in SQL.

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

### Triggering Events and Actions in SQL

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

### Table creation:

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

Example:

CREATE OR REPLACE TRIGGER display\_salary

```

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

```

## 16. Explain about embedded SQL.

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

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

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

```

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

```

```

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

.....statement to work with student name and id.....
EXEC SQL close c ;

```

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

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

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

EXEC SQL fetch *c* into :*si*, :*sn* END\_EXEC

Repeated calls to fetch get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL close *c* ;

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

### Updates Through Embedded SQL

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

### EXEC SQL

**declare *c* cursor for**

**select \*from instructor where dept\_name = 'Music' for update**

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

**update instructor**

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

## **17. Discuss about dynamic SQL.**

- The embedded SQL queries were written as part of the host program source code. Hence, any time we want to write a different query, we must modify the program code, and go through all the steps involved (compiling, debugging, testing, and so on).
- In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) *dynamically at runtime*. For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs.
- Another example is when a user-friendly interface generates SQL queries dynamically for the user based on point-and-click operations on a graphical schema (for example, a QBE-like interface).
- **Dynamic SQL** is one technique for writing this type of database program, by giving a simple example to illustrate how dynamic SQL can work.
- Program segment E3 in Figure reads a string that is input by the user (that string should be an SQL update command) into the string program variable sqlupdatestring in line 3. It then prepares this as an SQL command in line 4 by associating it with the SQL variable sqlcommand. Line 5 then executes the command.
- Notice that in this case no syntax check or other types of checks on the command are possible *at compile time*, since the SQL command is not available until runtime. This contrasts with our previous examples of embedded SQL, where the query could be checked at compile time because its text was in the program source code.
  
- Although including a dynamic update command is relatively straightforward in dynamic SQL, a dynamic query is much more complicated. This is because usually we do not know the types or the number of attributes to be retrieved by the SQL query when we are writing the program.
- A complex data structure is sometimes needed to allow for different numbers and types of attributes in the query result if no prior information is known about the dynamic query.
- In E3, the reason for separating PREPARE and EXECUTE is that if the command is to be executed multiple times in a program, it can be prepared only once. Preparing the command generally involves syntax and other types of checks by the system, as well as generating the code for executing it. It is possible to combine the PREPARE and EXECUTE commands (lines 4 and 5 in E3) into a single statement by writing

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

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

```
EXEC SQL BEGIN DECLARE SECTION ;
```

```
varchar sqlupdatestring [256] ;
```

```
EXEC SQL END DECLARE SECTION ;
```

```
...
```

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

```
EXEC SQL EXECUTE sqlcommand ;
```

**CS8492 / DATABASE MANAGEMENT SYSTEMS****UNIT-1****PART-A****1. What is the purpose of Database Management System? Nov/ Dec 2014**

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

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

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

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

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

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

The disadvantages of file processing systems are

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

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

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

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

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

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

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

## **8. Define instance and schema?**

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

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

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

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

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

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

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

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

The storage manager components include

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

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

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

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

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

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

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

A relationship is an association among several entities.

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

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

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

**15. Define null values**

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

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

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

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

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

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

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

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

It is a computer language and a subset of SQL, used to control access to data in a database.

Examples of DCL commands include:

GRANT: used to allow specified users to perform specified tasks.

REVOKE: used to cancel previously granted permission

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

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

- i. Duplicate elimination is an expensive operation one way to implement it, is to sort the tuples first and then eliminate duplicates.
- ii. The users may want to see duplicate tuples in the result of a query

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

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

- a. data definition language
- b. data manipulation language

- c. data control language
- d. transaction control language

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

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

**22. Define Data Manipulation Language.**

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

Ex. Insert, delete, modify

**23. List the SQL statements used for Transaction Control.**

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

**24. Define database objects.**

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

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

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

**26. What is a trigger in SQL?**

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

**27. What are primary keys?**

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

**28. Explain the basic structure of an SQL expression.**

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

- α. Select- used to list the attributes desired in the result of a query
- β. From- lists the relations to be scanned
- γ. 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
- iii) 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.

- find the name of employees who have borrowed a book published by McGraw-Hill.
- 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

- Which courses does a specific professor teach and what courses are taught by two specific professors?
- Who teaches a specific courses and where is his/her office?
- For specific student number in which courses is the student registered and what is his/her name?
- 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, title, authors, publisher)  
 Loan(empno, isbn, date)

Write the following queries in relational algebra and SQL.

- Find the name of employees who have borrowed a book published by McGraw-Hill.
- Find the name of employees who have borrowed all books published by McGraw-Hill.
- Find the name of the employee who has loan and age greater than 25

Answers:

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

- 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.
- Write a SELECT statement to display the name of all the candidates who have got less than 40 marks in exactly 2 subjects .
- 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.
- Write a SELECT statement to display the names of all the candidates who have got highest total marks in MSc., (Maths).

Answers:

- select degcode from candidate where degcode not in select degcode from degree;

- ii) select name from candidate,marks where candidate.seatid=marks.seatid and marks.mark <40 ;
  - iii) Select name,subject, count (seatno) from degree, marks group by degcode;
  - iv) Select seatno from marks m1,marks m2 where m1.mark>m2.mark;
5. Justify the need of embedded SQL .consider the relation student(studentno, name, mark and grade).Write embedded SQL statements in C language to retrieve all the student's records whose marks more than 90.

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

**UNIT II DATABASE DESIGN**

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

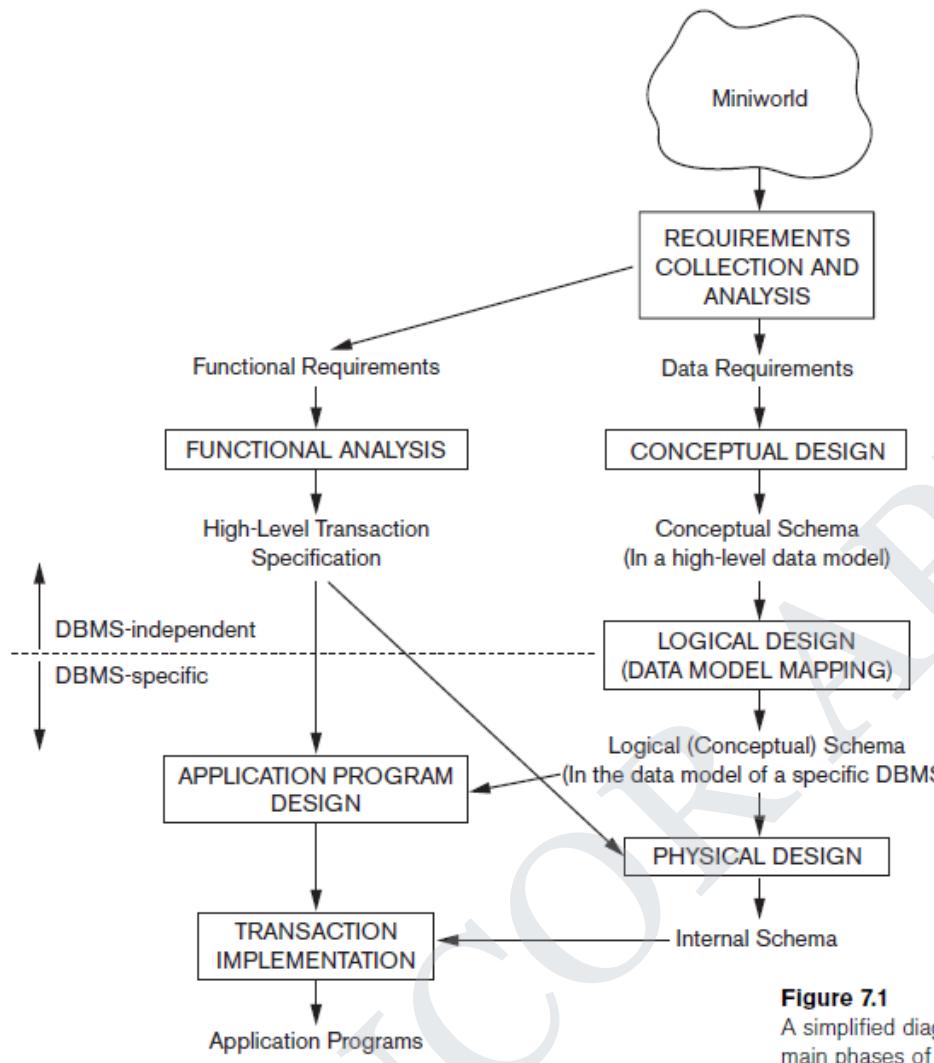
---

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

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

**Using High-Level Conceptual Data Models for Database Design:****1) Requirements collection and analysis :**

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

**Figure 7.1**

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

## 2) Conceptual design:

- Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided 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 easier to create a good conceptual data-base design.
- During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

**3) Logical design:**

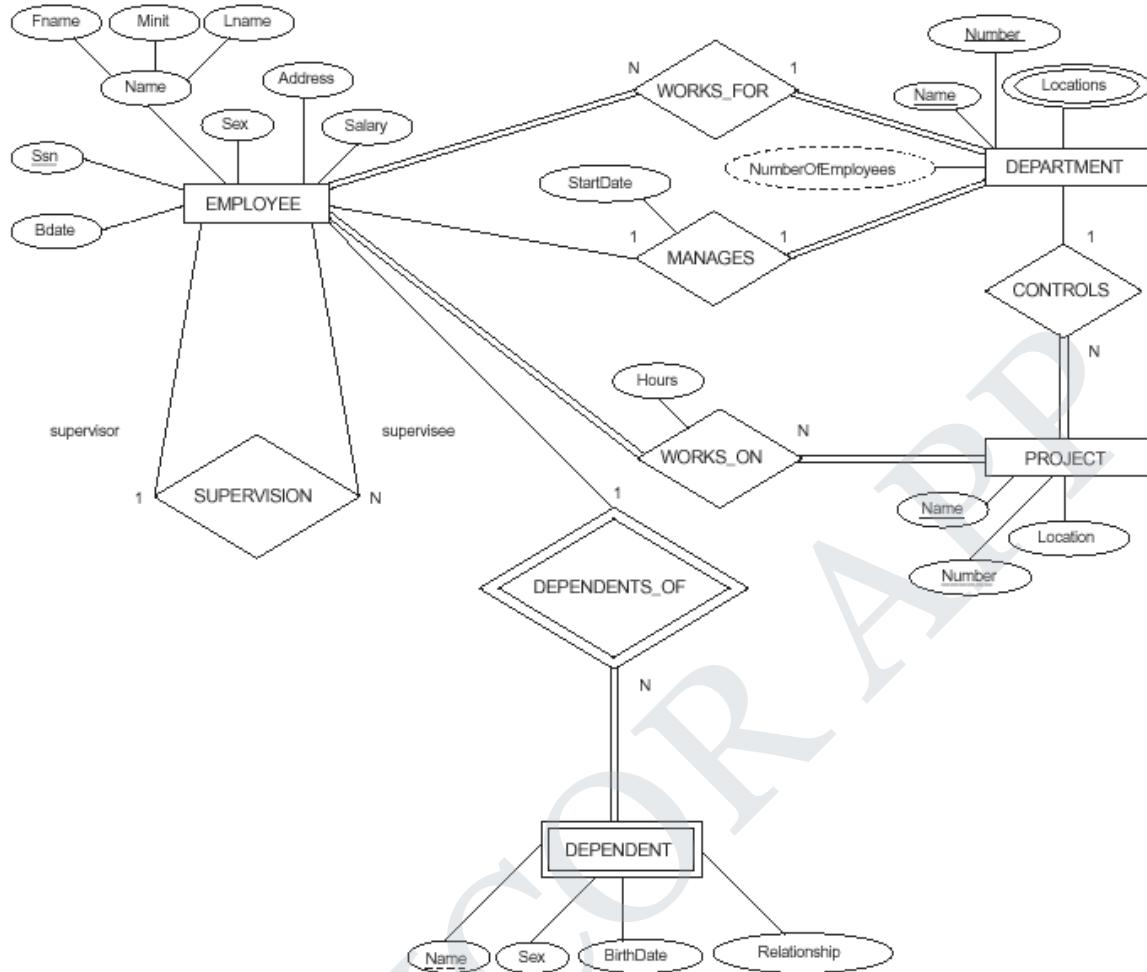
- The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model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

**4) Physical design:**

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

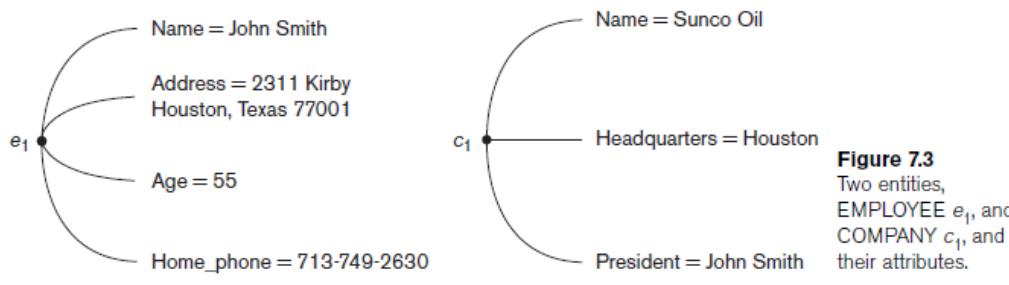
**A Sample Database Application**

- Consider a sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design.
- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- We store each employee's name, Social Security number, address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).
- We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.
- Figure shows how the schema for this database application can be displayed by means of the graphical notation known as **ER diagrams**. This figure will be explained gradually as the ER model concepts are presented. We describe the step-by-step process of deriving this schema from the stated requirements—and explain the ER diagrammatic notation—as we introduce the ER model concepts.



### Entity Types, Entity Sets, Attributes, and Keys

- The ER model describes data as entities, relationships, and attributes.
- 1) Entities and Attributes:**
- The basic object that the ER model represents is an **entity**, which is a thing in the real world with an independent existence.
  - An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course).
  - Each entity has **attributes**—the particular properties that describe it. For example, an **EMPLOYEE** entity may be described by the employee's name, age, address, salary, and job.
  - A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.



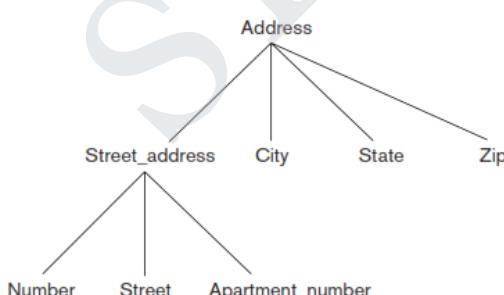
**Figure 7.3**  
Two entities,  
EMPLOYEE  $e_1$ , and  
COMPANY  $c_1$ , and  
their attributes.

- Figure shows two entities and the values of their attributes. The EMPLOYEE entity  $e_1$  has four attributes: Name, Address, Age, and Home\_phone; their values are ‘John Smith,’ ‘2311 Kirby, Houston, Texas 77001’, ‘55’, and ‘713-749-2630’, respectively. 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:

- simple versus composite
  - single-valued versus multivalued
  - 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) Single-Valued versus Multivalued Attributes:**

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

### **iii) Stored versus Derived Attributes:**

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

### **NULL Values:**

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

### **Complex Attributes:**

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

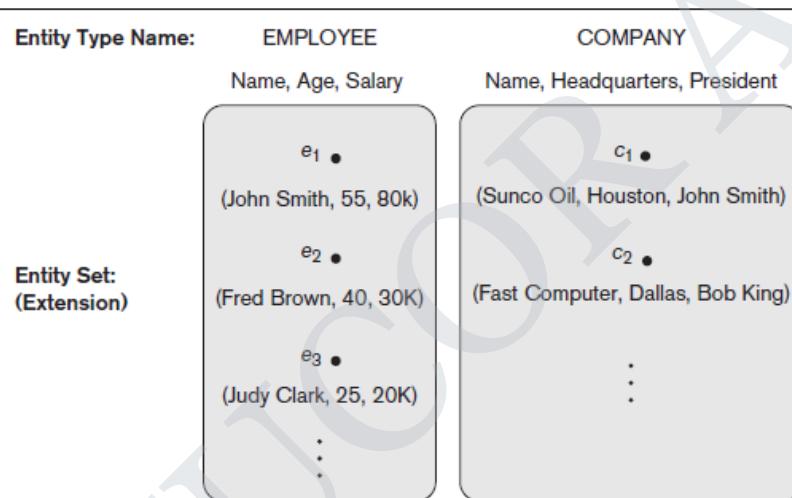
**Figure: complex attribute : Address\_phone**

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

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

### Entity Types and Entity Sets.

- A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its own value(s) for each attribute.
- An **entity type** defines a collection (or set) of entities that have the same attributes. Each entity type in the database is described by its name and attributes.

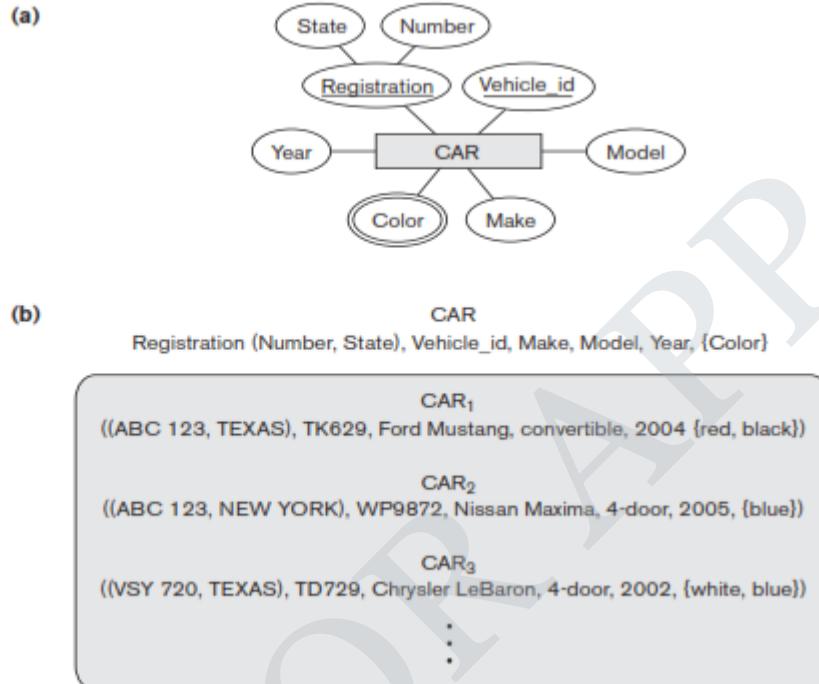


**Figure 7.6**  
Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

- Figure shows two entity types: EMPLOYEE and COMPANY, and a list of some of the attributes for each. A few individual entities of each type are also illustrated, along with the values of their attributes.
- The collection of all entities of a particular entity type in the data-base at any point in time is called an **entity set**; the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a type of entity as well as the current set of all employee entities in the database.
- An entity type is represented in ER diagrams as a rectangular box enclosing the entity type name.
- Attribute names are enclosed in ovals and are attached to their entity type by straight lines.
- Composite attributes are attached to their component attributes by straight lines.

- Multivalued attributes are displayed in double ovals. Figure (a) shows a CAR entity type in this notation.
- An entity type describes the **schema** or **intension** for a set of entities that share the same structure. The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.

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



### Key Attributes of an Entity Type:

- An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely.
- For example, the Name attribute is a key of the COMPANY entity type, because no two companies are allowed to have the same name.
- For the PERSON entity type, a typical key attribute is Ssn (Social Security number). Sometimes several attributes together form a key, meaning that the combination of the attribute values must be distinct for each entity.
- If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a composite attribute and designate it as a key attribute of the entity type. Notice that such a composite key must be minimal; that is, all component attributes must be included in the composite attribute to have the uniqueness property. Superfluous attributes must not be included in a key. In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure (a).

- Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for every entity set of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. It is not the property of a particular entity set; rather, it is a constraint on any entity set of the entity type at any point in time. This key constraint (and other constraints we discuss later) is derived from the constraints of the miniworld that the database represents.
- Some entity types have more than one key attribute. For example, each of the Vehicle\_id and Registration attributes of the entity type CAR (Figure 7.7) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, neither of which is a key on its own. An entity type may also have no key, in which case it is called a weak entity type..
- In our diagrammatic notation, if two attributes are underlined separately, then each is a key on its own. Unlike the relational model, there is no concept of primary key in the ER model; the primary key will be chosen during mapping to a relational schema.

### Value Sets (Domains) of Attributes.

- Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity.
- In Figure, if the range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70. Similarly, we can specify the value set for the Name attribute to be the set of strings of alphabetic characters separated by blank characters, and so on.
- Value sets are not displayed in ER diagrams, and are typically specified using the basic **data types** available in most programming languages, such as integer, string, Boolean, float, enumerated type, subrange, and so on. Additional data types to represent common database types such as date, time, and other concepts are also employed.
- Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function** from E to the power set<sup>6</sup> P(V ) of V:

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

- We refer to the value of attribute A for entity e as A(e). The previous definition covers both single-valued and multivalued attributes, as well as NULLs. A NULL value is represented by the empty set. For single-valued attributes, A(e) is restricted to being a singleton set for each entity e in E, whereas there is no restriction on multivalued attributes.<sup>7</sup> For a composite attribute A, the value set V is the power set of the Cartesian product of P(V<sub>1</sub>), P(V<sub>2</sub>), ..., P(V<sub>n</sub>), where V<sub>1</sub>, V<sub>2</sub>, ..., V<sub>n</sub> are the value sets of the simple component attributes that form A:

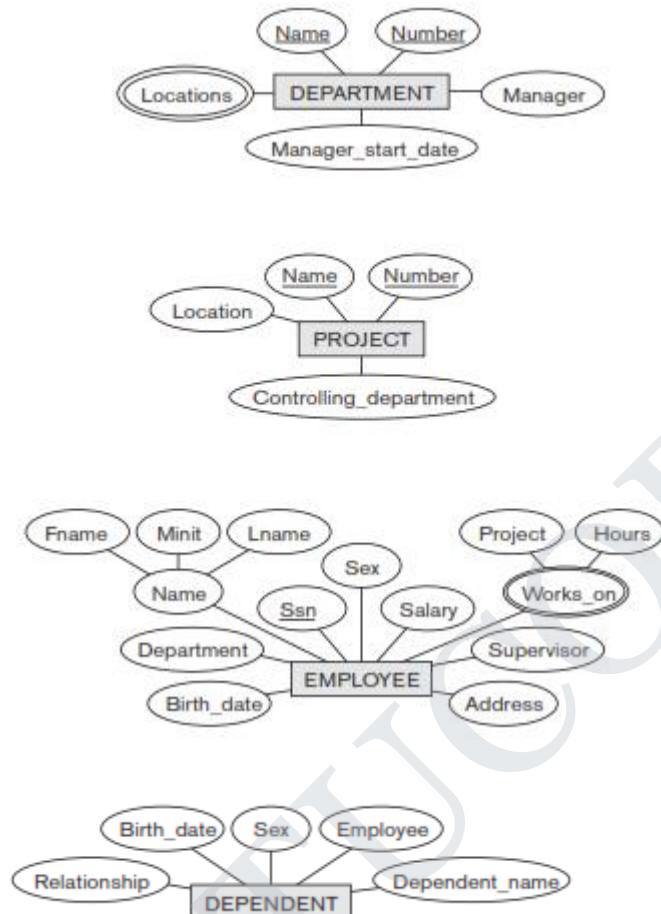
$$V = P(P(V_1) \times P(V_2) \times \dots \times P(V_n))$$

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

### Initial Conceptual Design of the COMPANY Database

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

2. An entity type PROJECT with attributes Name, Number, Location, and Controlling\_department. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth\_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—First\_name, Middle\_initial, Last\_name—or of Address.
4. An entity type DEPENDENT with attributes Employee, Dependent\_name, Sex, Birth\_date, and Relationship (to the employee).



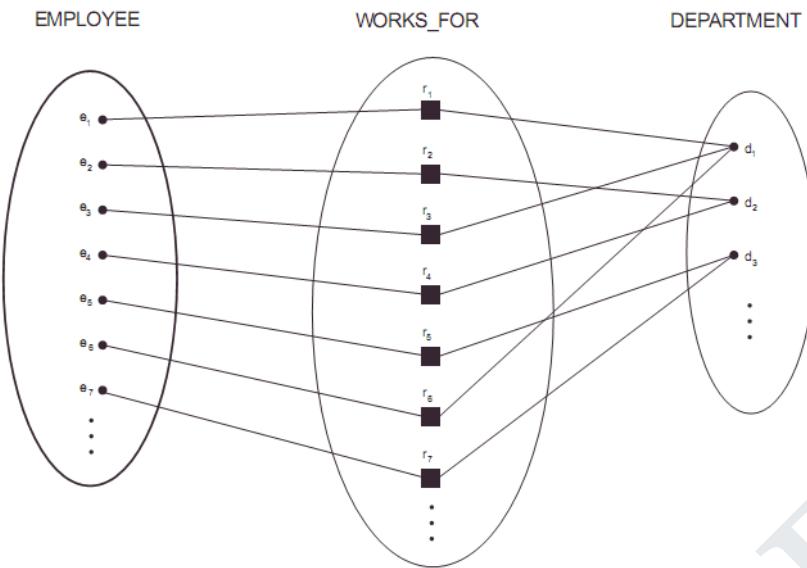
**Figure.** Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

#### **Relationship types, Roles, Structural Constraints:**

**Relationship type:** R among n Entity types E<sub>1</sub>, ..., E<sub>n</sub> defines a set of associations among entities from these types. Each association will be denoted as:

(e<sub>1</sub>, ..., e<sub>n</sub>) where e<sub>i</sub> belongs to E<sub>i</sub>, 1 <= i <= n.

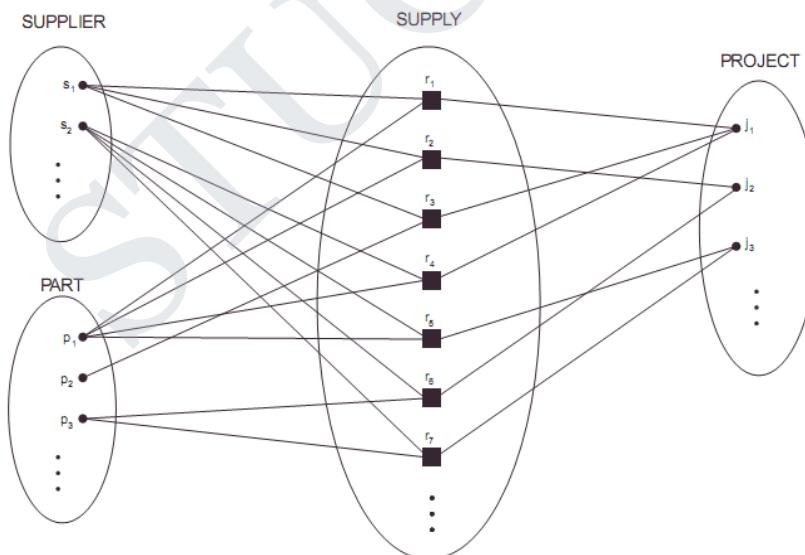
ex. WORKS\_FOR relationship



**Figure.** Some instances of the WORKS\_FOR relationship between EMPLOYEE and DEPARTMENT.

#### Degree of a relationship:

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



**Figure.** Some relationship instances of a ternary relationship SUPPLY.

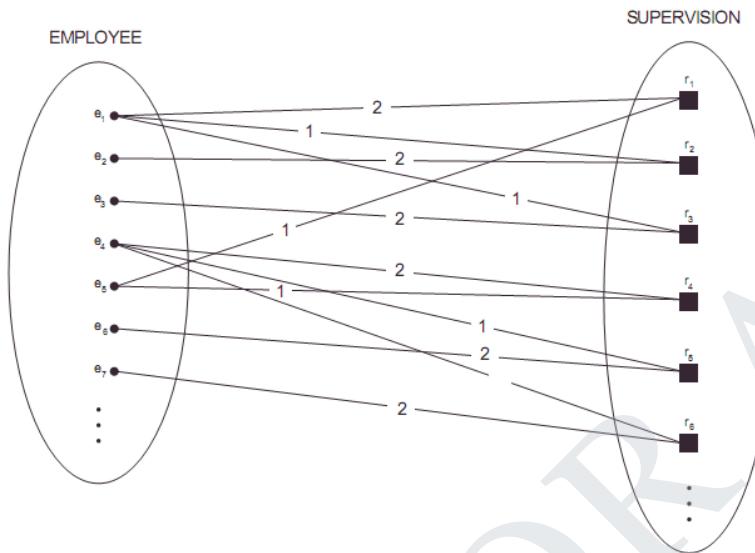
**Relationships as Attributes:** It is sometimes convenient to think of a relationship type in terms of attributes,

### Role names

Each entity participating in a relationship has a **ROLE**.

E.g. Employee plays the role of worker and Department plays the role of employer in the WORKS\_FOR relationship type

Role names are more important in recursive relationships.

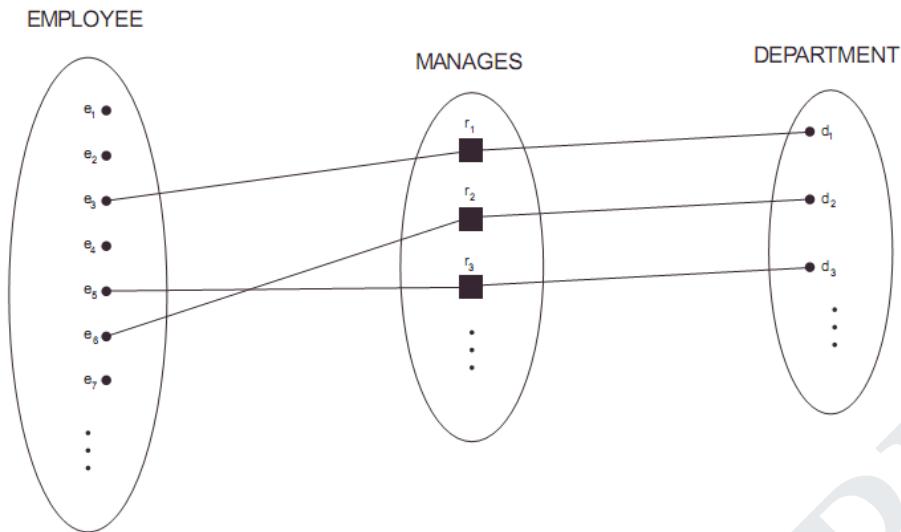


**Figure.** The recursive relationship SUPERVISION, where the EMPLOYEE entity type plays the two roles of supervisor (1) and supervisee (2).

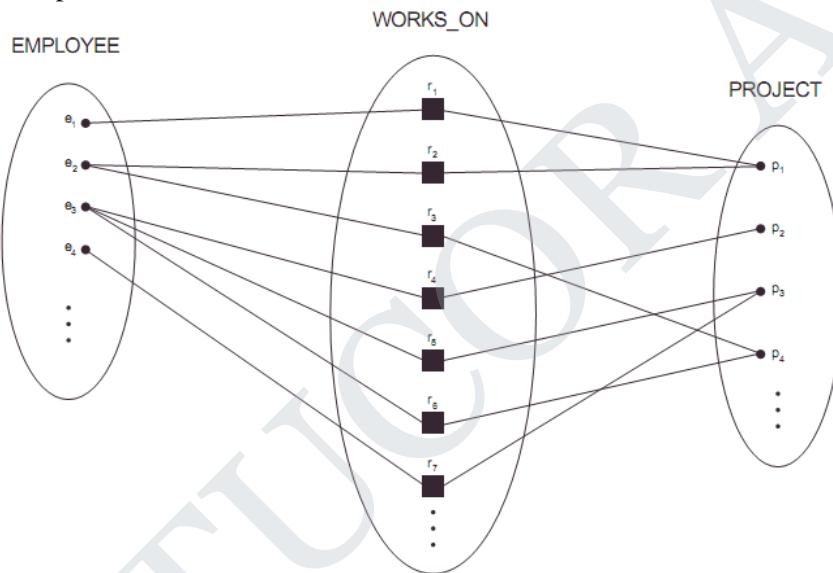
### Structural Constraints on Relationships

#### Two types:

- 1) Cardinality Ratio Constraint (1-1, 1-N, M-N)
- 2) Participation Constraint
  - \* Total participation (existence dependency)
  - \* Partial participation



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



**Figure.** The M:N relationship WORKS\_ON between EMPLOYEE and PROJECT.

In ER Diagrams:

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

#### Attributes of relationships:

e.x. Hours attribute for WORKS\_ON relationship

If relationship is 1-N or 1-1, these attributes can be migrated to the entity sets involved in the relationship.

1-N: migrate to N side

1-1: migrate to either side

**Weak Entity Types**

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

**Example:**

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

DEPENDENT\_OF

**Weak Entity Type is: DEPENDENT**

**Identifying Relationship is: DEPENDENTS\_OF**

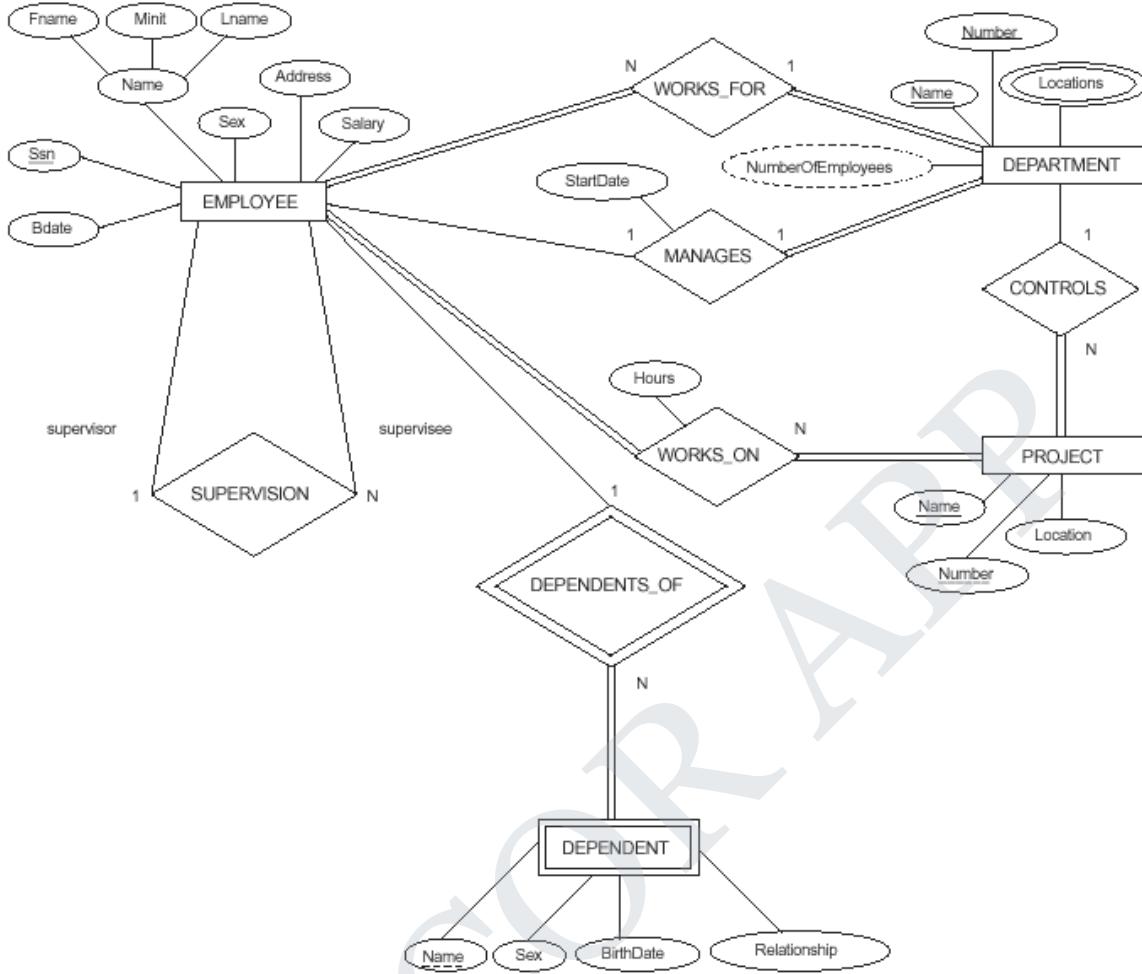
**Constraints on Relationships**

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

**Notation for ER Diagrams:**

| <u>Symbol</u> | <u>Meaning</u>                                                     |
|---------------|--------------------------------------------------------------------|
|               | ENTITY TYPE                                                        |
|               | WEAK ENTITY TYPE                                                   |
|               | RELATIONSHIP TYPE                                                  |
|               | IDENTIFYING RELATIONSHIP TYPE                                      |
|               | ATTRIBUTE                                                          |
|               | KEY ATTRIBUTE                                                      |
|               | MULTIVALUED ATTRIBUTE                                              |
|               | COMPOSITE ATTRIBUTE                                                |
|               | DERIVED ATTRIBUTE                                                  |
|               | TOTAL PARTICIPATION OF $E_2$ IN $R$                                |
|               | CARDINALITY RATIO $1:N$ FOR $E_1:E_2$ IN $R$                       |
|               | STRUCTURAL CONSTRAINT (min, max)<br>ON PARTICIPATION OF $E$ IN $R$ |

**Figure 3.14** Summary of ER diagram notation.



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

## **2.2. EXTENDED-ER (EER) MODEL :**

### **Extended-ER (EER) Model Concepts**

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

### **Subclasses and Superclasses:**

- An entity type may have additional meaningful subgroupings of its entities
- Example: EMPLOYEE may be further grouped into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED\_EMPLOYEE, HOURLY\_EMPLOYEE,...

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

#### **Attribute Inheritance in Superclass / Subclass Relationships**

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

#### **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

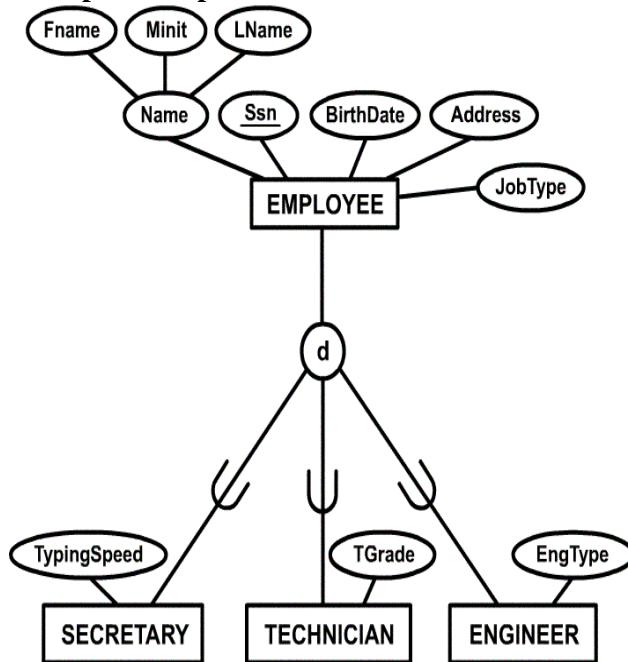
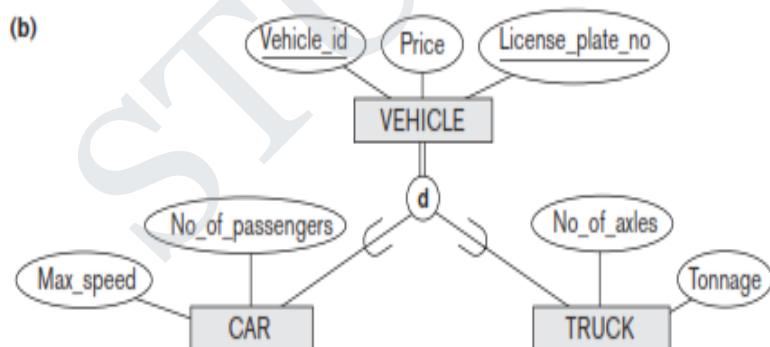
**Example of a Specialization**

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

**3)Generalization and Specialization**

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

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

### **Constraints on Specialization and Generalization**

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

### **Example of disjoint partial Specialization**

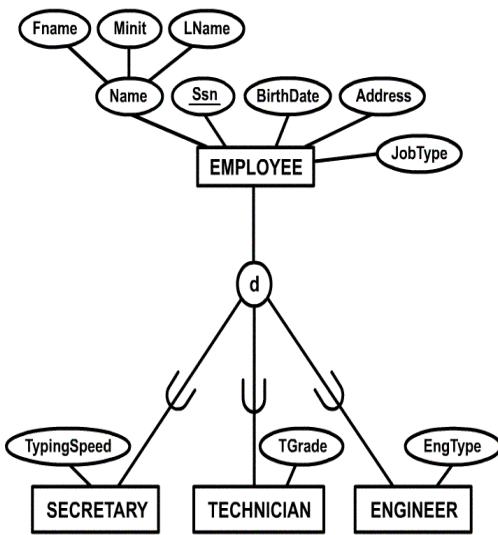


Figure. Disjoint partial Specialization

**Specialization / Generalization Hierarchies, Lattices and Shared Subclasses**

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

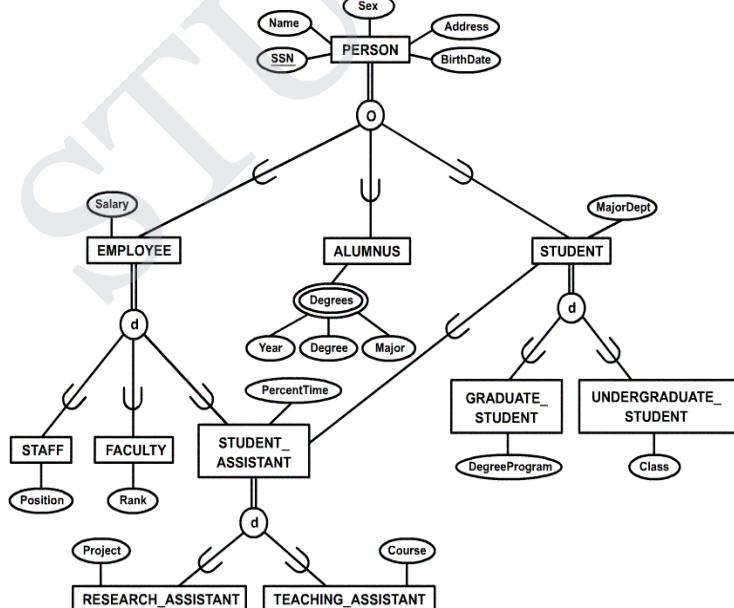
**Specialization / Generalization Lattice Example (UNIVERSITY)**

Figure. Specialization / Generalization

### Modeling of UNION types using categories:

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

### Example of categories (UNION TYPES)

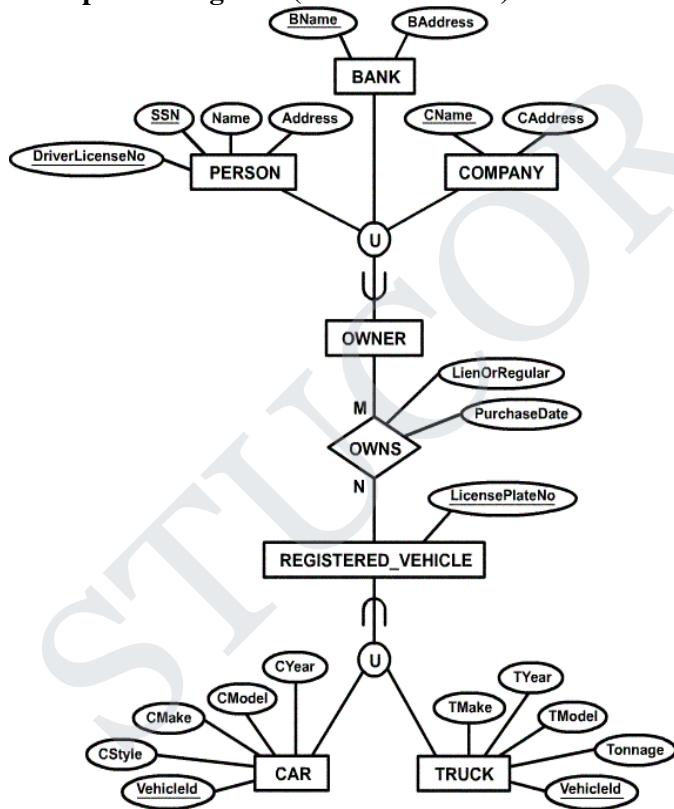
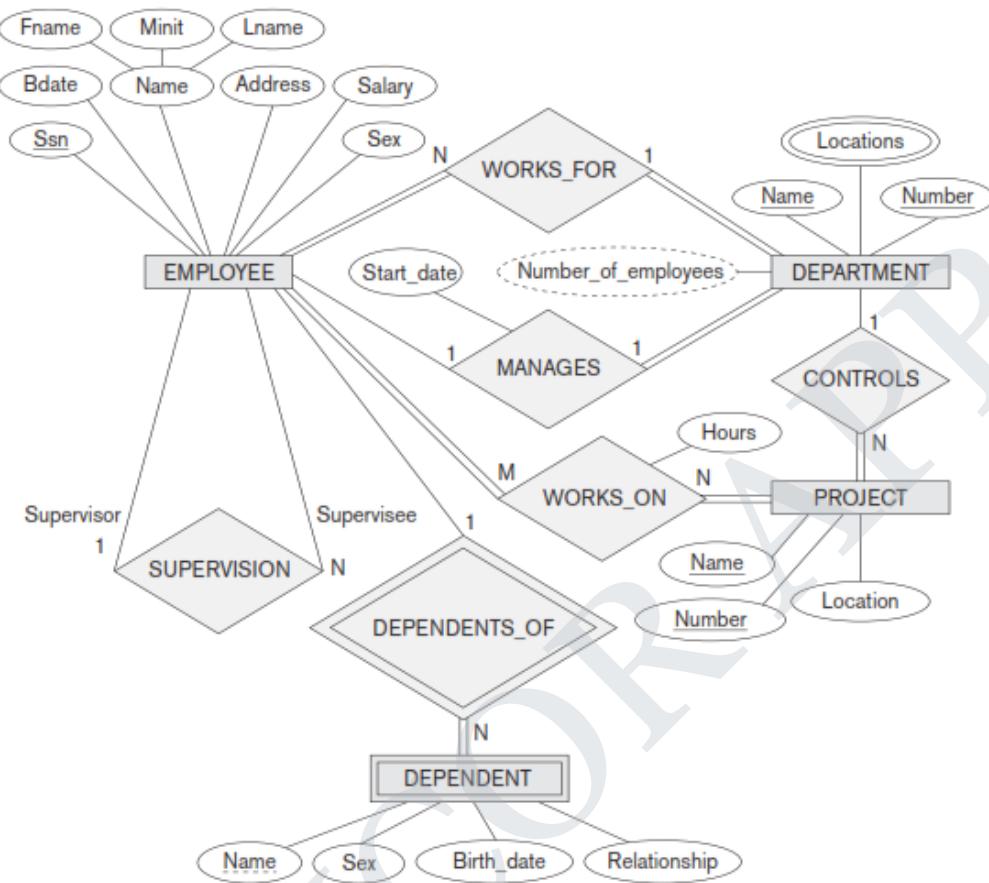
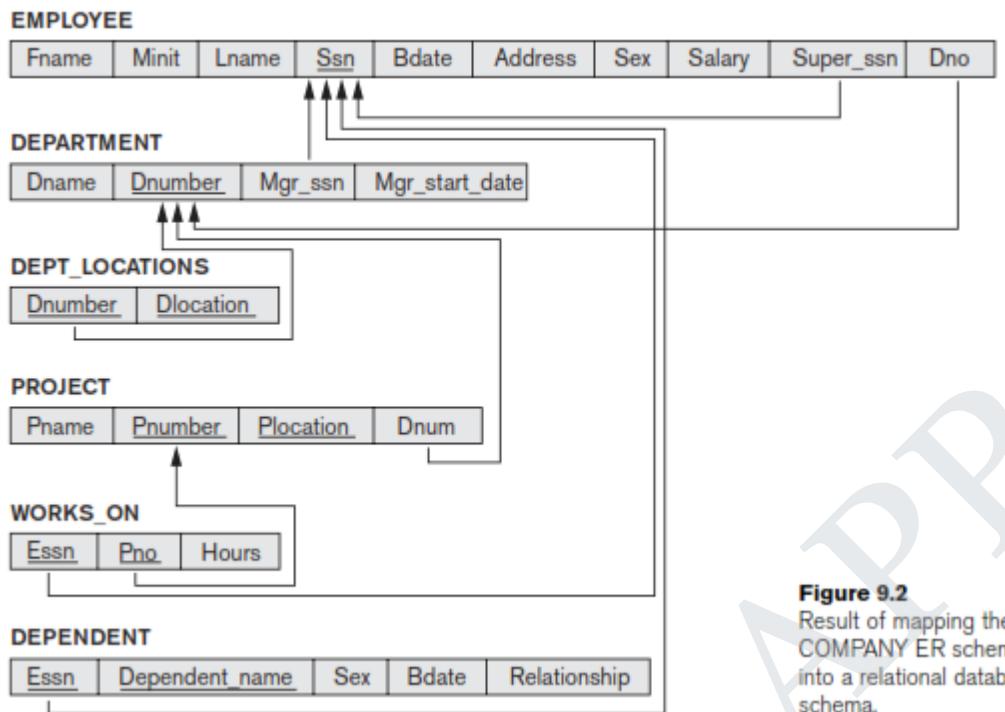


Figure. categories (UNION TYPES)

### 2.3. ER-TO-RELATIONAL MAPPING

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





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

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

#### **Step 1: Mapping of Regular Entity Types:**

- For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as the primary key for R. If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R.
- If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R. Knowledge about keys is also kept for index-ing purposes and other types of analyses.
- In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT in Figure . The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include the attributes Super\_ssn and Dno of EMPLOYEE, Mgr\_ssn and Mgr\_start\_date of DEPARTMENT, and Dnum of PROJECT. In our example, we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively. Knowledge that Dname of DEPARTMENT and Pname of PROJECT are secondary keys is kept for possible use later in the design.
- The relations that are created from the mapping of entity types are sometimes called **entity relations** because each tuple represents an entity instance.

#### **Step 2: Mapping of Weak Entity Types:**

- For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes (or simple components of composite attributes) of W as

attributes of R. In addition, include as foreign key attributes of R, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of W. The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

- If there is a weak entity type  $E_2$  whose owner is also a weak entity type  $E_1$ , then  $E_1$  should be mapped before  $E_2$  to determine its primary key first.
- In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it Essn, although this is not necessary.

**Figure .** Illustration of some mapping steps.

- Entity relations after step 1.
- Additional weak entity relation after step 2.
- Relationship relation after step 5.
- Relation representing multivalued attribute after step 6.

The diagram illustrates the mapping steps from entity relations to a relationship relation:

(a) Entity relations after step 1:

| <b>EMPLOYEE</b> |       |       |     |       |         |     |        |  |
|-----------------|-------|-------|-----|-------|---------|-----|--------|--|
| Fname           | Minit | Lname | Ssn | Bdate | Address | Sex | Salary |  |

| <b>DEPARTMENT</b> |         |
|-------------------|---------|
| Dname             | Dnumber |

| <b>PROJECT</b> |         |           |
|----------------|---------|-----------|
| Pname          | Pnumber | Plocation |

(b) Relationship relation after step 5:

| Essn | Dependent_name | Sex | Bdate | Relationship |
|------|----------------|-----|-------|--------------|
|------|----------------|-----|-------|--------------|

(c) Relationship relation after step 5:

| Essn | Pno | Hours |
|------|-----|-------|
|------|-----|-------|

(d) Relationship relation after step 5:

| Dnumber | Dlocation |
|---------|-----------|
|---------|-----------|

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

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

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

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

- (1) Foreign key approach
- (2) Merged relationship approach
- (3) Cross-reference or relationship relation approach.

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

#### **(1) Foreign key approach:**

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

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

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

#### **(2) Merged relation approach:**

An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. 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 each tuple 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 simple 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 create 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 reference 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 location 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                     | <i>Entity</i> relation                                 |
| 1:1 or 1:N relationship type    | Foreign key (or <i>relationship</i> relation)          |
| M:N relationship type           | <i>Relationship</i> relation and two foreign keys      |
| <i>n</i> -ary relationship type | <i>Relationship</i> relation and <i>n</i> 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                             |

### 2.4. FUNCTIONAL DEPENDENCIES:

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

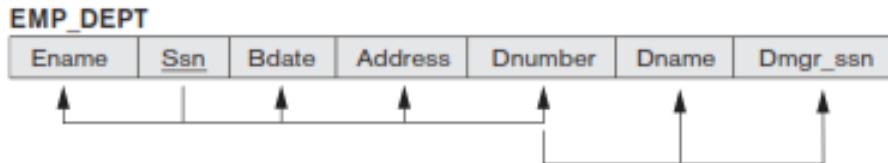
- This means that the values of the Y component of a tuple in  $r$  depend on, or are determined by, the values of the X component; alternatively, the values of the X component of a tuple uniquely (or **functionally**) determine the values of the Y component. We also say that there is a functional dependency 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  $X \rightarrow Y$  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  $X \rightarrow R$ .

If  $X \rightarrow Y$  in R, this does not say whether or not  $Y \rightarrow X$  in R.

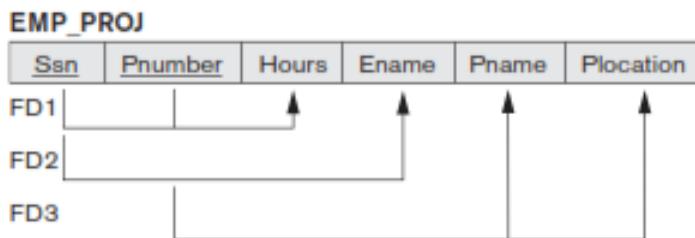
- A functional dependency is a property of the **semantics** or **meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of R—that is, how they relate to one another—to specify the functional dependencies that should hold on all relation states (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}  $\rightarrow$  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  $\rightarrow$  Area\_code used to exist as a relationship between postal codes and telephone number 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:
  - $Ssn \rightarrow Ename$
  - $Pnumber \rightarrow \{Pname, Plocation\}$
  - $\{Ssn, Pnumber\} \rightarrow Hours$
- 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.

(a)



(b)



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

## 2.5. NON LOSS DECOMPOSITION (OR) LOSSLESS DECOMPOSITION :

- The decomposition 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:

&lt;EmpInfo&gt;

| 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:

&lt;EmpDetails&gt;

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

&lt;DeptDetails&gt;

| 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:

&lt;EmpInfo&gt;

| 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:

## &lt;EmpDetails&gt;

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

## &lt;DeptDetails&gt;

| Dept_ID | Dept_Name  |
|---------|------------|
| Dpt1    | Operations |
| Dpt2    | HR         |
| Dpt3    | Finance    |

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

**2.6. NORMAL FORMS:**

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

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

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

**Database design may be performed using two approaches:**

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

- 2) In contrast, a **top-down design methodology** (also called design by analysis) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met. The theory described in this chapter is applicable to both the top-down and bottom-up design approaches, but is more appropriate when used with the top-down approach.
- Relational database design ultimately produces a set of relations. The implicit goals of the design activity are information preservation and minimum redundancy. Information is very hard to quantify—hence we consider information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships.
  - Thus, the relational design must preserve all of these concepts, which are originally captured in the conceptual design after the conceptual to logical design mapping. Minimizing redundancy implies minimizing redundant storage of the same information and reducing the need for multiple updates to maintain consistency across multiple copies of the same information in response to real-world events that require making an update.
  - Four informal guidelines that may be used as measures to determine the quality of relation schema design:
    - ✓ Making sure that the semantics of the attributes is clear in the schema
    - ✓ Reducing the redundant information in tuples
    - ✓ Reducing the NULL values in tuples
    - ✓ Disallowing the possibility of generating spurious tuples

### **Normalization of Relations:**

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

**Example:** Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp\_id for storing employee's id, emp\_name for storing employee's name, emp\_address for storing employee's address and emp\_dept for storing the department details in which the employee works. At some point of time the table looks like this:

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

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

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

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

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

- To overcome these anomalies we need to normalize the data.
- It can be considered as a “filtering” or “purification” process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database design-ers with the following:
  - a. A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
  - b. A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

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

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

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

- The nonadditive join property is extremely critical and **must be achieved at any cost**, whereas the dependency preservation property, although desirable, is sometimes sacrificed.
- **Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.
- A **superkey** of a relation schema  $R = \{A_1, A_2, \dots, A_n\}$  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.

## 2.7. FIRST NORMAL FORM (1NF)

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

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

a)

DEPARTMENT

| Dname | Dnumber | Dmgr_ssn | Dlocations |
|-------|---------|----------|------------|
|       |         |          |            |

(b)

DEPARTMENT

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

( c ) DEPARTMENT

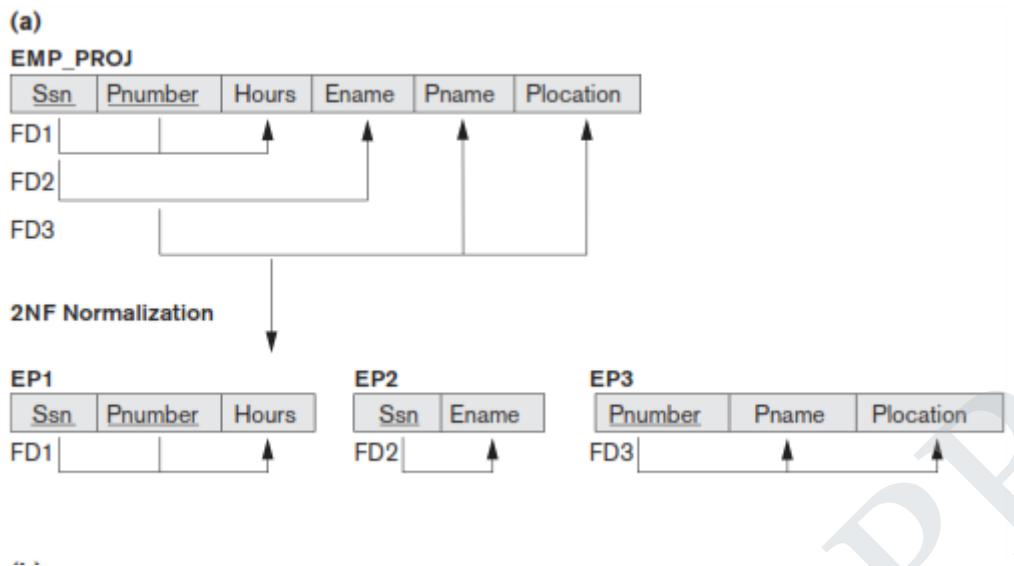
| Dname          | Dnumber | Dmgr_ssn  | Dlocation |
|----------------|---------|-----------|-----------|
| Research       | 5       | 333445555 | Bangalore |
| Research       | 5       | 333445555 | Chennai   |
| Research       | 5       | 333445555 | Delhi     |
| Administration | 4       | 987654321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

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

## 2.8. SECOND NORMAL FORM:

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

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

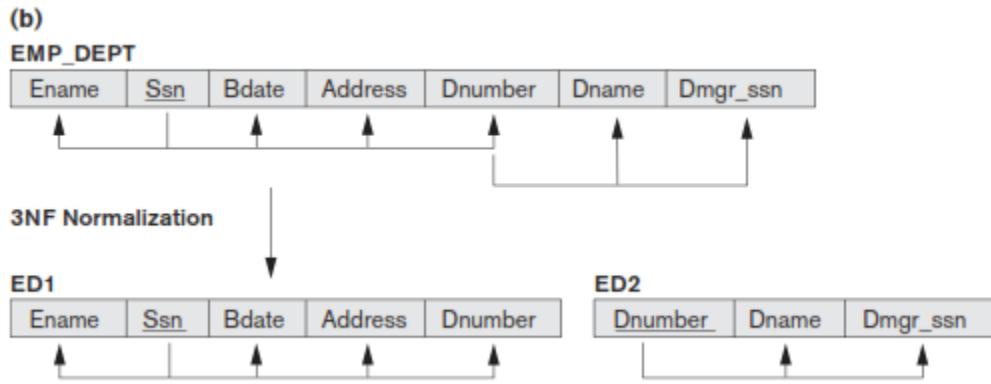


- The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP\_PROJ relation in Figure is in 1NF but is not in 2NF.
- The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP\_PROJ, thus violating the 2NF test.
- If a relation schema is not in 2NF, it can be second normalized or 2NF normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure lead to the decomposition of EMP\_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

## 2.9. THIRD NORMAL FORM:

- Third normal form (3NF)** is based on the concept of transitive dependency. A functional dependency  $X \rightarrow Y$  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  $X \rightarrow Z$  and  $Z \rightarrow Y$  hold. The dependency  $Ssn \rightarrow Dmgr\_ssn$  is transitive through Dnumber in EMP\_DEPT in Figure 15.3(a), because both the dependencies  $Ssn \rightarrow Dnumber$  and  $Dnumber \rightarrow Dmgr\_ssn$  hold and Dnumber is neither a key itself nor a subset of the key of EMP\_DEPT. Intuitively, we can see that the dependency of  $Dmgr\_ssn$  on Dnumber is undesirable in [ ] since Dnumber is not a key of EMP\_DEPT.

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



- The relation schema **EMP\_DEPT** in Figure 15.3(a) is in 2NF, since no partial dependencies on a key exist. However, **EMP\_DEPT** is not in 3NF because of the transitive dependency of **Dmgr\_ssn** (and also **Dname**) on **Ssn** via **Dnumber**. We can normalize **EMP\_DEPT** by decomposing it into the two 3NF relation schemas **ED1** and **ED2** shown in Figure. Intuitively, we see that **ED1** and **ED2** represent independent entity facts about employees and departments. A NATURAL JOIN operation on **ED1** and **ED2** will recover the original relation **EMP\_DEPT** without generating spurious tuples.
- Intuitively, we can see that any functional dependency in which the left-hand side is part (a proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute, is a problematic FD.
- 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF.

## 2.10. BOYCE-CODD NORMAL FORM (BCNF):

- Boyce-Codd normal form (BCNF)** was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF.
- A table is in BCNF if every functional dependency  $X \rightarrow Y$ ,  $X$  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.  $\text{EMP\_ID} \rightarrow \text{EMP\_COUNTRY}$
2.  $\text{EMP\_DEPT} \rightarrow \{\text{DEPT\_TYPE}, \text{EMP\_DEPT\_NO}\}$

**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.  $\text{EMP\_ID} \rightarrow \text{EMP\_COUNTRY}$
2.  $\text{EMP\_DEPT} \rightarrow \{\text{DEPT\_TYPE}, \text{EMP\_DEPT\_NO}\}$

**Candidate keys:****For the first table:** EMP\_ID**For the second table:** EMP\_DEPT**For the third table:** {EMP\_ID, EMP\_DEPT}

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

**2.11. MULTIVALUED DEPENDENCY AND FOURTH NORMAL FORM**

**Definition.** A multivalued dependency  $X \twoheadrightarrow Y$  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  $(R - (X \cup Y))$

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

$$t_3[Y] = t_1[Y] \text{ and } t_4[Y] = t_2[Y].$$

$$t_3[Z] = t_2[Z] \text{ and } t_4[Z] = t_1[Z].$$

- Whenever  $X \twoheadrightarrow Y$  holds, we say that X **multidetermines** Y. Because of the symmetry in the definition, whenever  $X \twoheadrightarrow Y$  holds in R, so does  $X \twoheadrightarrow Z$ . Hence,  $X \twoheadrightarrow Y$  implies  $X \twoheadrightarrow Z$ , and therefore it is sometimes written as  $X \twoheadrightarrow Y|Z$ .
- An MVD  $X \twoheadrightarrow Y$  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  $\twoheadrightarrow$  Pname. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. A trivial MVD will hold in any relation state r of R; it is called trivial because it does not specify any significant or meaningful constraint on R.
- If we have a nontrivial MVD in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure(a), the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname). This redundancy is clearly undesirable. However, the EMP schema is in BCNF because no functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP.
- Notice that relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together. Furthermore, it is rare that such all-key relations with a combinatorial occurrence of repeated values would be designed in practice. However, recognition of MVDs as a potential problematic dependency is essential in relational design.

---

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

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

(a) EMP

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

(b) EMP\_PROJECTS

| Ename | Pname |
|-------|-------|
| Smith | X     |
| Smith | Y     |

EMP\_DEPENDENTS

| Ename | Dname |
|-------|-------|
| Smith | John  |
| Smith | Anna  |

Figure . (a)The EMP relation with two MVDs: Ename  $\rightarrow\!\!\rightarrow$  Pname and Ename  $\rightarrow\!\!\rightarrow$  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  $\rightarrow\!\!\rightarrow$  Pname and Ename  $\rightarrow\!\!\rightarrow$  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  $\rightarrow\!\!\rightarrow$  Pname in EMP\_PROJECTS and Ename  $\rightarrow\!\!\rightarrow$  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.

## 2.12. JOIN DEPENDENCIES AND FIFTH NORMAL FORM

**Definition.** A join dependency (JD) can be said to exist if the join of R<sub>1</sub> and R<sub>2</sub> over C is equal to relation R. Where, R<sub>1</sub> and R<sub>2</sub> are the decompositions R<sub>1</sub>(A, B, C), and R<sub>2</sub>(C,D) of a given relations R (A, B, C, D). Alternatively, R<sub>1</sub> and R<sub>2</sub> 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 | CLASS      | LECTURER |
| Mathematics | Alex     | SEMESTER 1 | Alex     |
| Mathematics | Rose     | SEMESTER 1 | Rose     |
| Physics     | Rose     | SEMESTER 1 | Rose     |
| Physics     | Joseph   | SEMESTER 2 | Joseph   |
| Chemistry   | Adam     | SEMESTER 1 | Adam     |

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

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

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

**UNIT III    TRANSACTIONS**

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

**3.1. TRANSACTION CONCEPTS:**

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

**3.2. ACID Properties:**

Properties of the transactions:

- 1) **Atomicity.** Either all operations of the transaction are reflected properly in the database, or none are.
  - 2) **Consistency.** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the data-base.
  - 3) **Isolation.** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions  $T_i$  and  $T_j$  , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started or  $T_j$  started execution after  $T_i$  finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
  - 4) **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

**A Simple Transaction Model:**

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

- $\text{read}(X)$ , 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.
- $\text{write}(X)$ , 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:

```

 $T_i : \text{read}(A);$
 $A := A - 50;$
 $\text{write}(A);$
 $\text{read}(B);$
 $B := B + 50;$
 $\text{write}(B).$
```

Let us now consider each of the ACID properties.

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

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

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

Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction  $T_i$  is executed to completion, there exists a point at which the value of account  $A$  is \$950 and

the value of account  $B$  is \$2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account  $A$  is \$950, and the value of account  $B$  is \$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

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

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

funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either:

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

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

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

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

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

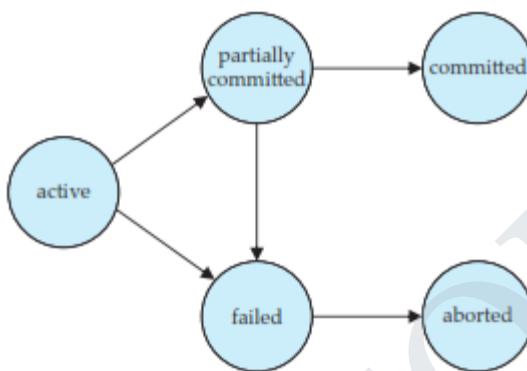
A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as we shall see in Section 14.5. Other solutions have therefore been developed;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

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

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

- We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model.
- A transaction must be in one of the following states:
  - 1) **Active**, the initial state; the transaction stays in this state while it is executing.
  - 2) **Partially committed**, after the final statement has been executed.
  - 3) **Failed**, after the discovery that normal execution can no longer proceed.
  - 4) **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
  - 5) **Committed**, after successful completion. We say that a transaction has committed only if it has entered the committed state.



**Figure.** State diagram of a transaction.

- Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if it has either committed or aborted.
- A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.
- The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.
- A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.
- We must be cautious when dealing with **observable external writes**, such as writes to a user's screen, or sending email. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system.
- Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in a special relation in the database, and to perform the actual writes only after the transaction enters the committed state.
- If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.
- Handling external writes can be more complicated in some situations. For example, suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically).
- It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the user's account, needs to be executed when the system is restarted.
- As another example, consider a user making a booking over the Web. It is possible that the database system or the application server crashes just after the booking transaction commits. It is also possible that the network connection to the user is lost just after the booking transaction commits.
- In either case, even though the transaction has committed, the external write has not taken place. To handle such situations, the application must be designed such that when the user connects to the Web application again, she will be able to see whether her transaction had succeeded or not.
- For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity.

### **3.3. SCHEDULES:**

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

**Two good reasons for allowing concurrency:**

- **Improved throughput and resource utilization.** A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.
- **Reduced waiting time.** There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Con-current execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.
- The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.
- When several transactions run concurrently, the isolation property may be violated, resulting in database consistency being destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure the isolation property and thus database consistency.
- The database system must control the interaction among the concurrent trans-actions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control schemes**.

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

$T_1$ :  
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  $T_2$  appearing in the right column. The final values of accounts  $A$  and  $B$ , after the execution in Figure takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in accounts  $A$  and  $B$  —that is, the sum  $A + B$ —is preserved after the execution of both transactions.

| $T_1$                                                                         | $T_2$                                                                                             |
|-------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <pre> read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) commit </pre> | <pre> read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) commit </pre> |

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

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

| $T_1$                                                                         | $T_2$                                                                                             |
|-------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <pre> read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) commit </pre> | <pre> read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) commit </pre> |

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

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

- A schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction. For example, in transaction  $T_1$ , the instruction write( $A$ ) must appear before the instruction read( $B$ ), in any valid schedule.
- Note that we include in our schedules the **commit** operation to indicate that the transaction has entered the committed state. In the following discussion, we shall refer to the first execution sequence ( $T_1$  followed by  $T_2$ ) as schedule 1, and to the second execution sequence ( $T_2$  followed by  $T_1$ ) as schedule 2.
- These schedules are **serial**:
- Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.
- When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.
- Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.<sup>1</sup>
- Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order  $T_1$  followed by  $T_2$ . The sum  $A + B$  is indeed preserved.
- Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure. After the execution of this schedule, we arrive at a state where the final values of accounts  $A$  and  $B$  are \$950 and \$2100, respectively. This final state is an *inconsistent state*, since we have gained \$50 in the process of the concurrent execution. Indeed, the sum  $A + B$  is not preserved by the execution of the two transactions.
- If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.

| $T_1$                                                                                         | $T_2$                                                                                               |
|-----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <b>read(<math>A</math>)</b><br>$A := A - 50$<br><b>write(<math>A</math>)</b>                  | <b>read(<math>A</math>)</b><br>$temp := A * 0.1$<br>$A := A - temp$<br><b>write(<math>A</math>)</b> |
| <b>read(<math>B</math>)</b><br>$B := B + 50$<br><b>write(<math>B</math>)</b><br><b>commit</b> | <b>read(<math>B</math>)</b><br>$B := B + temp$<br><b>write(<math>B</math>)</b><br><b>commit</b>     |

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

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

| $T_1$                                                                                                                         | $T_2$                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>read(<math>A</math>)</b><br>$A := A - 50$                                                                                  | <b>read(<math>A</math>)</b><br>$temp := A * 0.1$<br>$A := A - temp$<br><b>write(<math>A</math>)</b><br><b>read(<math>B</math>)</b> |
| <b>write(<math>A</math>)</b><br><b>read(<math>B</math>)</b><br>$B := B + 50$<br><b>write(<math>B</math>)</b><br><b>commit</b> | $B := B + temp$<br><b>write(<math>B</math>)</b><br><b>commit</b>                                                                   |

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

### **3.4. SERIALIZABILITY:**

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

Different forms of schedule serializability are

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

#### **1. CONFLICT SERIALIZABILITY:**

- Let us consider a schedule  $S$  in which there are two consecutive instructions, and  $I, J$ , 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.  $I = \text{read}(Q)$ ,  $J = \text{read}(Q)$ . The order of  $I$  and  $J$  does not matter, since the same value of  $Q$  is read by  $T_i$  and  $T_j$ , regardless of the order.
  - b.  $I = \text{read}(Q)$ ,  $J = \text{write}(Q)$ . If  $I$  comes before  $J$ , then  $T_i$  does not read the value of  $Q$  that is written by  $T_j$  in instruction  $J$ . If  $J$  comes before  $I$ , then  $T_i$  reads the value of  $Q$  that is written by  $T_j$ . Thus, the order of  $I$  and  $J$  matters.
  - c.  $I = \text{write}(Q)$ ,  $J = \text{read}(Q)$ . The order of  $I$  and  $J$  matters for reasons similar to those of the previous case.
  - d.  $I = \text{write}(Q)$ ,  $J = \text{write}(Q)$ . Since both instructions are write operations, the order of these instructions does not affect either  $T_i$  or  $T_j$ . However, the value obtained by the next read( $Q$ ) instruction of  $S$  is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write( $Q$ ) instruction after  $I$  and  $J$  in  $S$ , then the order of  $I$  and  $J$  matters.

and  $J$  directly affects the final value of  $Q$  in the database state that results from schedule  $S$ .

| $T_1$                       | $T_2$                       |
|-----------------------------|-----------------------------|
| read( $A$ )<br>write( $A$ ) | read( $A$ )<br>write( $A$ ) |
| read( $B$ )<br>write( $B$ ) | read( $B$ )<br>write( $B$ ) |

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

| $T_1$                       | $T_2$                       |
|-----------------------------|-----------------------------|
| read( $A$ )<br>write( $A$ ) | read( $A$ )                 |
| read( $B$ )                 | write( $A$ )                |
| write( $B$ )                | read( $B$ )<br>write( $B$ ) |

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

- Thus, only in the case where both  $I$  and  $J$  are read instructions does the relative order of their execution not matter.
- We say that  $I$  and  $J$  **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.
- To illustrate the concept of conflicting instructions, we consider schedule 3 in Figure 14.6. The write( $A$ ) instruction of  $T_1$  conflicts with the read( $A$ ) instruction of  $T_2$ . However, the write( $A$ ) instruction of  $T_2$  does not conflict with the read( $B$ ) instruction of  $T_1$ , because the two instructions access different data items.

| $T_1$                                                      | $T_2$                                                      |
|------------------------------------------------------------|------------------------------------------------------------|
| read( $A$ )<br>write( $A$ )<br>read( $B$ )<br>write( $B$ ) | read( $A$ )<br>write( $A$ )<br>read( $B$ )<br>write( $B$ ) |

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

| $T_3$        | $T_4$        |
|--------------|--------------|
| read( $Q$ )  |              |
| write( $Q$ ) | write( $Q$ ) |

**Figure.** Schedule 7.

- Let  $I$  and  $J$  be consecutive instructions of a schedule  $S$ . If  $I$  and  $J$  are instructions of different transactions and  $I$  and  $J$  do not conflict, then we can swap the order of  $I$  and  $J$  to produce a new schedule  $S'$ .  $S'$  is equivalent to  $S$ , since all instructions appear in the same order in both schedules except for  $I$  and  $J$ , whose order does not matter.
- Since the write( $A$ ) instruction of  $T_2$  in schedule 3 does not conflict with the read( $B$ ) instruction of  $T_1$ , we can swap these instructions to generate an equivalent schedule, schedule 5. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

We continue to swap non-conflicting instructions:

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

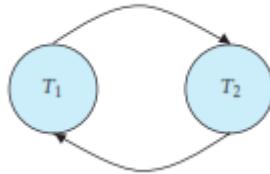
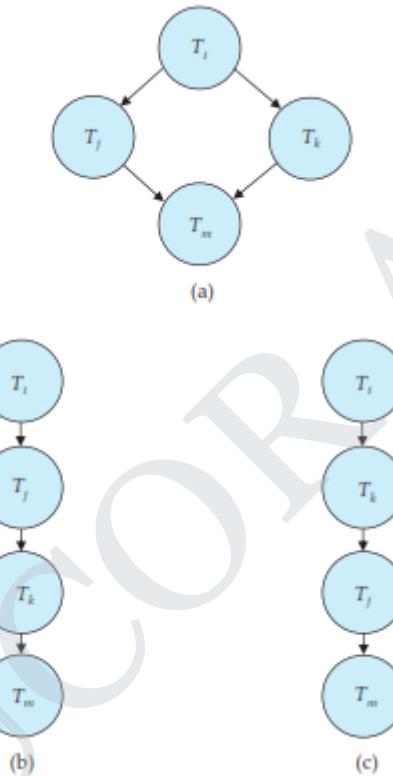
The final result of these swaps, schedule 6, is a serial schedule. Note that schedule 6 is exactly the same as schedule 1, but it shows only the read and write instructions. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial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

### Definition:

- If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are conflict equivalent. The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule  $S$  is conflict serializable if it is conflict equivalent to a serial schedule.
- Not all serial schedules are conflict equivalent to each other. For example, schedules 1 and 2 are not conflict equivalent.
- Schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.
- Finally, consider schedule 7; it consists of only the significant operations (that is, the read and write) of transactions  $T_3$  and  $T_4$ . This schedule is not conflict serializable, since it is not equivalent to either the serial schedule  $\langle T_3, T_4 \rangle$  or the serial schedule  $\langle T_4, T_3 \rangle$ .

**PRECEDENCE GRAPH:****Figure.** Precedence graph for (a) schedule 1 and (b) schedule 2.

- Precedence graph is a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule  $S$ . We construct a directed graph, called a **precedence graph**, from  $S$ . This graph consists of a pair  $G = (V, E)$ , where  $V$  is a set of vertices and  $E$  is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges  $T_i \rightarrow T_j$  for which one of three conditions holds:
  - ✓  $T_i$  executes write( $Q$ ) before  $T_j$  executes read( $Q$ ).
  - ✓  $T_i$  executes read( $Q$ ) before  $T_j$  executes write( $Q$ ).
  - ✓  $T_i$  executes write( $Q$ ) before  $T_j$  executes write( $Q$ ).
- If an edge  $T_i \rightarrow T_j$  exists in the precedence graph, then, in any serial schedule  $S$  equivalent to  $S$ ,  $T_i$  must appear before  $T_j$ .
- For example, the precedence graph for schedule 1 in Figure(a) contains the single edge  $T_1 \rightarrow T_2$ , since all the instructions of  $T_1$  are executed before the first instruction of  $T_2$  is executed. Similarly, Figure(b) shows the precedence graph for schedule 2 with the single edge  $T_2 \rightarrow T_1$ , since all the instructions of  $T_2$  are executed before the first instruction of  $T_1$  is executed.
- The precedence graph for schedule 4 appears in Figure. It contains the edge  $T_1 \rightarrow T_2$ , because  $T_1$  executes read( $A$ ) before  $T_2$  executes write( $A$ ). It also contains the edge  $T_2 \rightarrow T_1$ , because  $T_2$  executes read( $B$ ) before  $T_1$  executes write( $B$ ).
- **If the precedence graph for  $S$  has a cycle, then schedule  $S$  is not conflict serializable. If the graph contains no cycles, then the schedule  $S$  is conflict serializable.**
- A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**. There are several possible linear orders that can be obtained through a topological sort.
- For example, the graph of Figure(a) has the two acceptable linear orderings shown in Figures(b) and (c).

**Figure.** Precedence graph for schedule 4.**Figure .** Illustration of topological sorting.

- Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms.
- Cycle-detection algorithms, such as those based on depth-first search, require on the order of  $n^2$  operations, where  $n$  is the number of vertices in the graph (that is, the number of transactions).
- Returning to our previous examples, note that the precedence graphs for schedules 1 and 2, indeed do not contain cycles. The precedence graph for schedule 4, on the other hand, contains a cycle, indicating that this schedule is not conflict serializable.

- It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent. For example, consider transaction  $T_5$ , which transfers \$10 from account  $B$  to account  $A$ .
- Let schedule 8 be as defined in Figure. We claim that schedule 8 is not conflict equivalent to the serial schedule  $\langle T_1, T_5 \rangle$ , since, in schedule 8, the write( $B$ ) instruction of  $T_5$  conflicts with the read( $B$ ) instruction of  $T_1$ . This creates an edge  $T_5 \rightarrow T_1$  in the precedence graph.
- Similarly, we see that the write( $A$ ) instruction of  $T_1$  conflicts with the read instruction of  $T_5$  creating an edge  $T_1 \rightarrow T_5$ . This shows that the precedence graph has a cycle and that schedule 8 is not serializable. However, the final values of accounts  $A$  and after the execution of either schedule 8 or the serial schedule  $\langle T_1, T_5 \rangle$  are the same —\$960 and \$2040, respectively.

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

**Figure.** Schedule 8.

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

### Recoverable Schedules:

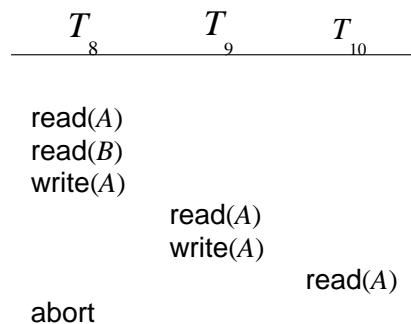
| $T_6$        | $T_7$       |
|--------------|-------------|
| read( $A$ )  |             |
| write( $A$ ) |             |
|              | read( $A$ ) |
|              | commit      |
| read( $B$ )  |             |

**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:  $\text{read}(A)$ . 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  $\text{read}(A)$  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 transaction  $T_9$ . Transaction  $T_9$  writes a value of  $A$  that is read by transaction  $T_{10}$ . Suppose that, at this point,  $T_8$  fails.  $T_8$  must be rolled back. Since  $T_9$  is dependent on  $T_8$ ,  $T_9$  must be rolled back. Since  $T_{10}$  is dependent on  $T_9$ ,  $T_{10}$  must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.**



**Figure.** Schedule 10.

- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally,

a **cascadeless schedule** is one where, for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ . It is easy to verify that every cascadeless schedule is also recoverable.

## 2. VIEW SERIALIZABILITY:

- There is another form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the **read** and **write** operations of transactions.
- Consider two schedules  $S$  and  $S'$ , where the same set of transactions 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  $S'$ .
- Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.
- The concept of view equivalence leads to the concept of view serializability. We say that a schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule.
- As an illustration, suppose that we augment schedule 4 with transaction  $T_{29}$ , and obtain the following view serializable (schedule 5):
 

|               |               |               |
|---------------|---------------|---------------|
| $T_{27}$      | $T_{28}$      | $T_{29}$      |
| read ( $Q$ )  |               |               |
|               | write ( $Q$ ) |               |
| write ( $Q$ ) |               | write ( $Q$ ) |

|               |               |               |
|---------------|---------------|---------------|
| $T_{27}$      | $T_{28}$      | $T_{29}$      |
| read ( $Q$ )  |               |               |
|               | write ( $Q$ ) |               |
| write ( $Q$ ) |               | write ( $Q$ ) |

- Indeed, schedule 5 is view equivalent to the serial schedule  $\langle T_{27}, T_{28}, T_{29} \rangle$ , 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  $\text{write}(Q)$  operations without having performed a  $\text{read}(Q)$  operation. Writes of this sort are called **blind writes**. Blind writes appear in any view-serializable schedule that is not conflict serializable.

### **3.5. NEED FOR CONCURRENCY CONTROL:**

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

### **3.6. LOCK-BASED PROTOCOLS**

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

#### **Locks:**

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

Two modes are:

- 1) **Shared.** If a transaction  $T_i$  has obtained a **shared-mode lock** (denoted by S) on item  $Q$ , then  $T_i$  can read, but cannot write,  $Q$ .
- 2) **Exclusive.** If a transaction  $T_i$  has obtained an **exclusive-mode lock** (denoted by X) on item  $Q$ , then  $T_i$  can both read and write  $Q$ .

|   | S     | X     |
|---|-------|-------|
| S | true  | false |
| X | false | false |

**Figure.** Lock-compatibility matrix comp.

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

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

- Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.
- A transaction requests a shared lock on data item  $Q$  by executing the  $\text{lock-S}(Q)$  instruction. Similarly, a transaction requests an exclusive lock through the  $\text{lock-X}(Q)$  instruction. A transaction can unlock a data item  $Q$  by the  $\text{unlock}(Q)$  instruction.
- To access a data item, transaction  $T_i$  must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency-control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus,  $T_i$  is made to **wait** until all incompatible locks held by other transactions have been released.
- Transaction  $T_i$  may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

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

```

 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.** Transaction  $T_1$ .

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

```

 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.

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

```

 $T_3:$ lock-X(B);
read(B);
 $B := B - 50$;
write(B);
lock-X(A);
read(A);
 $A := A + 50$;
write(A);
unlock(B);
unlock(A).

```

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

```

 $T_4:$ lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

```

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

- You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of \$250 being displayed, is no longer possible with  $T_3$  and  $T_4$ . Other schedules are possible.  $T_4$  will not print out an inconsistent result in any of them;
- Unfortunately, locking can lead to an undesirable situation.
- Consider the partial schedule of Figure for  $T_3$  and  $T_4$ . Since  $T_3$  is holding an exclusive-mode lock on  $B$  and  $T_4$  is requesting a shared-mode lock on  $B$ ,  $T_4$  is waiting for  $T_3$  to unlock  $B$ . Similarly, since  $T_4$  is holding a shared-mode lock on  $A$  and  $T_3$  is requesting an exclusive-mode lock on  $A$ ,  $T_3$  is waiting for  $T_4$  to unlock  $A$ . Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock.
- When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that

transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. There are ways to avoid deadlock in some situations.
- However, in general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

| T3                                                | T4                                |
|---------------------------------------------------|-----------------------------------|
| lock-X(B)<br>read(B)<br>$B := B - 50$<br>write(B) | lock-S(A)<br>read(A)<br>lock-S(B) |

**Figure.** Schedule 2

- We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules.

### **GRANTING OF LOCKS:**

- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.
- However, care must be taken to avoid the following scenario. Suppose a transaction  $T_2$  has a shared-mode lock on a data item, and another transaction  $T_1$  requests an exclusive-mode lock on the data item.
- Clearly,  $T_1$  has to wait for  $T_2$  to release the shared-mode lock. Meanwhile, a transaction  $T_3$  may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to  $T_2$ , so  $T_3$  may be granted the shared-mode lock.
- At this point  $T_2$  may release the lock, but still  $T_1$  has to wait for  $T_3$  to finish. But again, there may be a new transaction  $T_4$  that requests a shared-mode lock on the same data item, and is granted the lock before  $T_3$  releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the

data item, and each transaction releases the lock a short while after it is granted, but  $T_1$  never gets the exclusive-mode lock on the data item. The transaction  $T_1$  may never make progress, and is said to be **starved**.

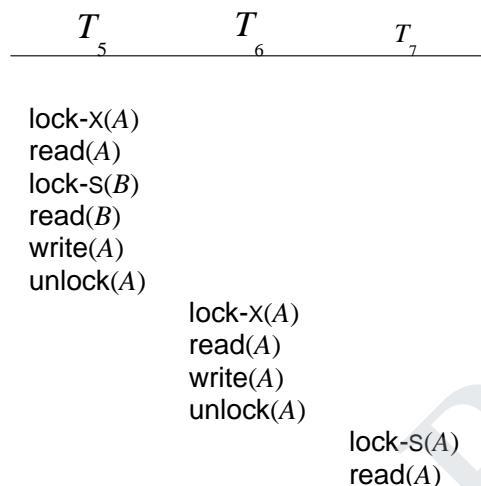
- We can avoid starvation of transactions by granting locks in the following manner: When a transaction  $T_i$  requests a lock on a data item  $Q$  in a particular mode  $M$ , the concurrency-control manager grants the lock provided that:
  - There is no other transaction holding a lock on  $Q$  in a mode that conflicts with  $M$ .
  - There is no other transaction that is waiting for a lock on  $Q$  and that made its lock request before  $T_i$ .
  - Thus, a lock request will never get blocked by a lock request that is made later.

### **3.7. THE TWO-PHASE LOCKING PROTOCOL:**

- One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

- 1) **Growing phase.** A transaction may obtain locks, but may not release any lock.
- 2) **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

- Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
- The two-phase locking protocol ensures conflict serializability.
- Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction.
- Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions. We leave the proof as an exercise for you to do.
- Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions  $T_3$  and  $T_4$  are two phase, but, in schedule 2, they are deadlocked.
- Cascading rollback may occur under two-phase locking.
- Consider the partial schedule of Figure 15.8. Each transaction observes the two-phase locking protocol, but the failure of  $T_5$  after the read(A) step of  $T_7$  leads to cascading rollback of  $T_6$  and  $T_7$ .

**Figure .Partial schedule under two-phase locking.****STRICT TWO-PHASE LOCKING PROTOCOL:**

- Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

**RIGOROUS TWO-PHASE LOCKING PROTOCOL:**

- Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits.
- Consider the following two transactions, for which we have shown only some of the significant read and write operations:

$T_8$ : read( $a_1$ );  
 read( $a_2$ );  
 ...  
 read( $a_n$ );  
 write( $a_1$ ).

$T_9$ : read( $a_1$ );  
 read( $a_2$ );  
 display( $a_1 + a_2$ ).

- If we employ the two-phase locking protocol, then  $T_8$  must lock  $a_1$  in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that  $T_8$  needs an exclusive lock on  $a_1$  only at the end of its execution, when it writes  $a_1$ . Thus, if  $T_8$  could initially lock  $a_1$  in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since  $T_8$  and  $T_9$  could access  $a_1$  and  $a_2$  simultaneously.
- This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed.
- We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by  **downgrade**.
- Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

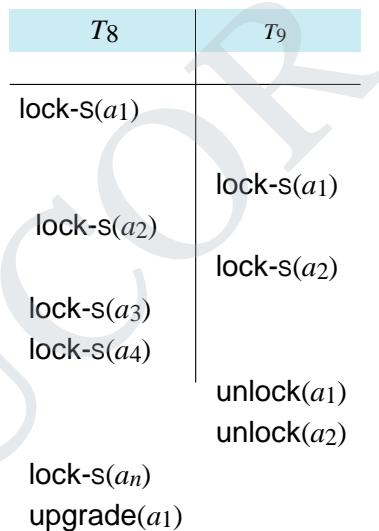


Figure. Incomplete schedule with a lock conversion.

- Returning to our example, transactions  $T_8$  and  $T_9$  can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure, where only some of the locking instructions are shown.
- Note that a transaction attempting to upgrade a lock on an item  $Q$  may be forced to wait. This enforced wait occurs if  $Q$  is currently locked by *another* transaction in shared mode.
- Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

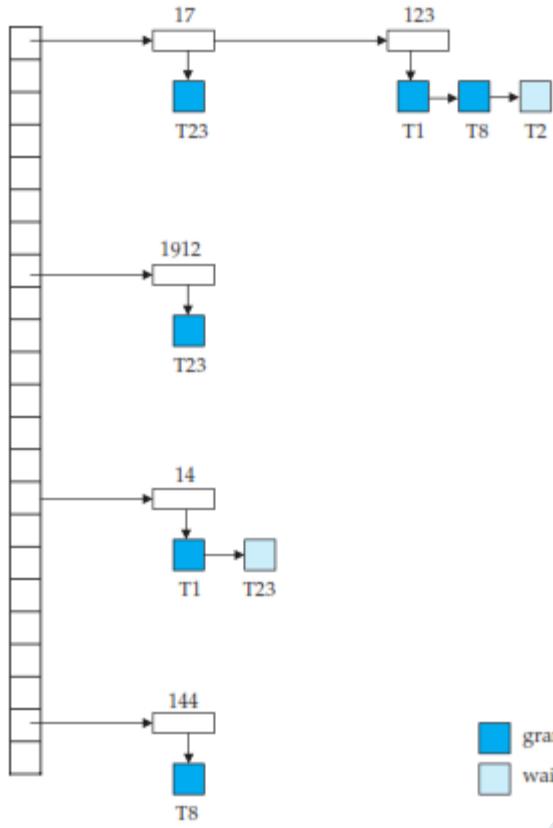
- For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. We shall see examples when we consider other locking protocols later in this chapter.
- Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.
- A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:
  - When a transaction  $T_i$  issues a  $\text{read}(Q)$  operation, the system issues a  $\text{lock-S}(Q)$  instruction followed by the  $\text{read}(Q)$  instruction.
  - When  $T_i$  issues a  $\text{write}(Q)$  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  $\text{write}(Q)$  instruction. Otherwise, the system issues a  $\text{lock-X}(Q)$  instruction, followed by the  $\text{write}(Q)$  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 **lock table**.

- Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.
  - Figure shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.
- 
- Although the figure does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction.
  - The lock manager processes requests this way:
    - When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.
    - It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.
    - When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.
    - If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction , it releases all locks held by the aborted transaction.

**Figure.** Lock table.

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

#### GRAPH-BASED PROTOCOLS: (or) Tree Protocol

- The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.
- To acquire such prior knowledge, we impose a partial ordering  $\rightarrow$  on the set  $\mathbf{D} = \{d_1, d_2, \dots, d_h\}$  of all data items. If  $d_i \rightarrow d_j$ , then any transaction accessing both  $d_i$  and  $d_j$  must access  $d_i$  before accessing  $d_j$ . This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.
- The partial ordering implies that the set  $\mathbf{D}$  may now be viewed as a directed acyclic graph, called a **database graph**. In this section, for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We shall present a simple protocol, called the *tree protocol*, which is

restricted to employ only *exclusive* locks. References to other, more complex, graph-based locking protocols are in the bibliographical notes.

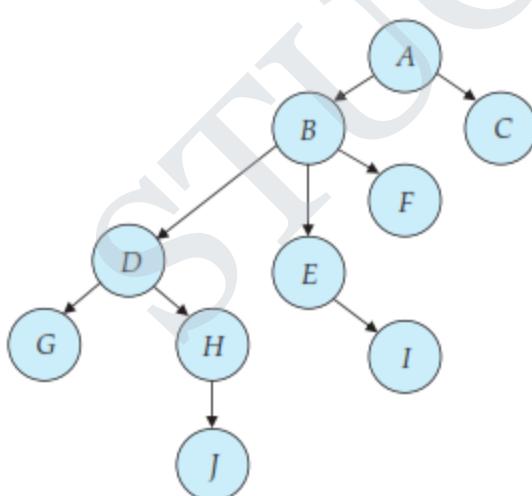
- In the **tree protocol**, the only lock instruction allowed is lock-X. Each trans-action  $T_i$  can lock a data item at most once, and must observe the following rules:
  - The first lock by  $T_i$  may be on any data item.
  - Subsequently, a data item  $Q$  can be locked by  $T_i$  only if the parent of  $Q$  is currently locked by  $T_i$ .
  - Data items may be unlocked at any time.
  - A data item that has been locked and unlocked by  $T_i$  cannot subsequently be relocked by  $T_i$ .
- All schedules that are legal under the tree protocol are conflict serializable.
- To illustrate this protocol, consider the database graph of Figure. The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

$T_{10}$ : lock-X( $B$ ); lock-X( $E$ ); lock-X( $D$ ); unlock( $B$ ); unlock( $E$ ); lock-X( $G$ ); unlock( $D$ ); unlock( $G$ ).

$T_{11}$ : lock-X( $D$ ); lock-X( $H$ ); unlock( $D$ ); unlock( $H$ ).

$T_{12}$ : lock-X( $B$ ); lock-X( $E$ ); unlock( $E$ ); unlock( $B$ ).

$T_{13}$ : lock-X( $D$ ); lock-X( $H$ ); unlock( $D$ ); unlock( $H$ ).



**Figure.** Tree-structured database graph.

- One possible schedule in which these four transactions participated appears in Figure. Note that, during its execution, transaction  $T_{10}$  holds locks on two *disjoint* subtrees.
- Observe that the schedule of Figure is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.
- The tree protocol does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction.
- Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write, we record which transaction performed the last write to the data item.
- Whenever a transaction  $T_i$  performs a read of an uncommitted data item, we record a **commit dependency** of  $T_i$  on the transaction that performed the last write to the data item. Transaction  $T_i$  is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts,  $T_i$  must also be aborted.

| $T_{10}$                                                         | $T_{11}$                                        | $T_{12}$                       | $T_{13}$                                                         |
|------------------------------------------------------------------|-------------------------------------------------|--------------------------------|------------------------------------------------------------------|
| lock-X( $B$ )                                                    |                                                 |                                |                                                                  |
|                                                                  | lock-X( $D$ )<br>lock-X( $H$ )<br>unlock( $D$ ) |                                |                                                                  |
| lock-X( $E$ )<br>lock-X( $D$ )<br>unlock( $B$ )<br>unlock( $E$ ) |                                                 |                                |                                                                  |
|                                                                  |                                                 | lock-X( $B$ )<br>lock-X( $E$ ) |                                                                  |
| lock-X( $G$ )<br>unlock( $D$ )                                   | unlock( $H$ )                                   |                                | lock-X( $D$ )<br>lock-X( $H$ )<br>unlock( $D$ )<br>unlock( $H$ ) |
|                                                                  |                                                 | unlock( $E$ )<br>unlock( $B$ ) |                                                                  |
| unlock( $G$ )                                                    |                                                 |                                |                                                                  |

Figure .Serializable schedule under the tree protocol.

- The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

- However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items  $A$  and  $J$  in the database graph of Figure must lock not only  $A$  and  $J$ , but also data items  $B$ ,  $D$ , and  $H$ .
- This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.
- For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that are not possible under the tree protocol, and vice versa.

### **3.8. DEADLOCK:**

- A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions  $\{T_0, T_1, \dots, T_n\}$  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  $\dots$ , and  $T_{n-1}$  is waiting for a data item that  $T_n$  holds, and  $T_n$  is waiting for a data item that  $T_0$  holds. None of the transactions can make progress in such a situation.
- The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.
- There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state.
- Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme. Both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.
- Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

## **DEADLOCK PREVENTION:**

There are five approaches to deadlock prevention.

- 1) The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked.

### **Two main disadvantages to this protocol:**

- a) It is often hard to predict, before the transaction begins, what data items need to be locked;
- b) Data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

- 2) Another approach for preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed is to ensure that locks are requested in the right order.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction  $T_j$  requests a lock that transaction  $T_i$  holds, the lock granted to  $T_i$  may be **preempted** by rolling back of  $T_i$ , and granting of the lock to  $T_j$ . To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted.

Two different deadlock-prevention schemes using timestamps have been proposed:

### **3) Wait –die scheme:**

- Consider a transaction  $T_i$  requests to lock a resource (data item), which is already held with a conflicting lock by another transaction  $T_j$ .
- ( TS- Time Stamp)
- If  $TS(T_i) < TS(T_j)$  – that is  $T_i$ , which is requesting a conflicting lock, is older than  $T_j$  – then  $T_i$  is allowed to wait until the data-item is available. Else  $T_i$  dies.
- $T_i$  is restarted later with a random delay but with the same timestamp.
- The **wait–die** scheme is a non-preemptive technique.

### **4) Wound–wait scheme:**

- It is a preemptive technique.
- Consider a transaction  $T_i$  requests to lock a resource (data item), which is already held with a conflicting lock by another transaction  $T_j$ .
- ( TS- Time Stamp)

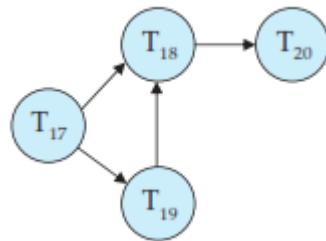
- If  $TS(T_i) < TS(T_j)$ , then  $T_i$  forces  $T_j$  to be rolled back – that is  $T_i$  wounds  $T_j$ .  $T_j$  is restarted later with a random delay but with the same timestamp.
- Else  $T_i$  is forced to wait until the resource is available.
- The major problem with both of these schemes is that unnecessary rollbacks may occur.

### 5) Time out based approach:

- In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.
- If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.
- The timeout scheme is particularly easy to implement, and works well if transactions are short and if long waits are likely to be due to deadlocks. However, in general it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources. Starvation is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

### DEADLOCK DETECTION:

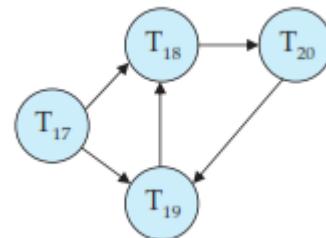
- If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:
  - Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
  - Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
  - Recover from the deadlock when the detection algorithm determines that a deadlock exists.
- Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair  $G = (V, E)$ , where  $V$  is a set of vertices and  $E$  is a set of edges.
- The set of vertices consists of all the transactions in the system. Each element in the set  $E$  of edges is an ordered pair  $T_i \rightarrow T_j$ . If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from transaction  $T_i$  to  $T_j$ , implying that transaction  $T_i$  is waiting for transaction  $T_j$  to release a data item that it needs.

**Figure.** Wait-for graph with no cycle.

- When transaction  $T_i$  requests a data item currently being held by transaction  $T_j$ , then the edge  $T_i \rightarrow T_j$  is inserted in the wait-for graph. This edge is removed only when transaction  $T_j$  is no longer holding a data item needed by transaction  $T_i$ .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.
- To illustrate these concepts, consider the wait-for graph in Figure, which depicts the following situation:
  - Transaction  $T_{17}$  is waiting for transactions  $T_{18}$  and  $T_{19}$ .
  - Transaction  $T_{19}$  is waiting for transaction  $T_{18}$ .
  - Transaction  $T_{18}$  is waiting for transaction  $T_{20}$ .
- Since the graph has no cycle, the system is not in a deadlock state.
- Suppose now that transaction  $T_{20}$  is requesting an item held by  $T_{19}$ . The edge  $T_{20} \rightarrow T_{19}$  is added to the wait-for graph, resulting in the new system state in Figure 15.14. This time, the graph contains the cycle:

$$T_{18} \rightarrow T_{20} \rightarrow T_{19} \rightarrow T_{18}$$

implying that transactions  $T_{18}$ ,  $T_{19}$ , and  $T_{20}$  are all deadlocked.

**Figure.** Wait-for graph with a cycle.

- If deadlocks occur frequently, then the detection algorithm should be invoked more frequently. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

**RECOVERY FROM DEADLOCK:**

- When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock.
- Three actions need to be taken:
  - 1) **Selection of a victim.** Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including:
    - a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
    - b. How many data items the transaction has used.
    - c. How many more data items the transaction needs for it to complete.
    - d. How many transactions will be involved in the rollback.
  - 2) **Rollback.** Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
- The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded.
- The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the bibliographical notes for relevant references.
- 3) **Starvation.** In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

**UNIT IV      IMPLEMENTATION TECHNIQUES**

RAID – File Organization – Organization of Records in Files – Indexing and Hashing – Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for SELECT and JOIN operations – Query optimization using Heuristics and Cost Estimation.

---

**4.1. RAID**

- The data-storage requirements of some applications (in particular Web, database, and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.
- Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.
- A variety of disk-organization techniques, collectively called **redundant arrays of independent disks (RAID)**, have been proposed to achieve improved performance and reliability.
- In the past, system designers viewed storage systems composed of several small, cheap disks as a cost-effective alternative to using large, expensive disks; the cost per megabyte of the smaller disks was less than that of larger disks. In fact, the I in RAID, which now stands for *independent*, originally stood for *inexpensive*. Today, however, all disks are physically small, and larger-capacity disks actually have a lower cost per megabyte.
- RAID systems are used for their higher reliability and higher performance rate, rather than for economic reasons.
- Another key justification for RAID use is easier management and operations.

**Improvement of Reliability via Redundancy:**

- If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data. Such a high frequency of data loss is unacceptable.
- The solution to the problem of reliability is to introduce **redundancy**; that is, we store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost, so the effective mean time to failure is increased, provided that we

at lead to loss of data or to non availability of data.

- The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or, sometimes, *shadowing*). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other.
- Data will be lost only if the second disk fails before the first failed disk is repaired.
- The mean time to failure (where failure is the loss of data) of a mirrored disk depends on the mean time to failure of the individual disks, as well as on the **mean time to repair**, which is the time it takes (on an average) to replace a failed disk and to restore the data on it.
- Power failures, and natural disasters such as earthquakes, fires, and floods, may result in damage to both disks at the same time. As disks age, the probability of failure increases, increasing the chance that a second disk will fail while the first is being repaired. In spite of all these considerations, however, mirrored-disk systems offer much higher reliability than do single-disk systems.
- Mirrored-disk systems with mean time to data loss of about 500,000 to 1,000,000 hours, or 55 to 110 years, are available today.

### **Improvement in Performance via Parallelism:**

- Now let us consider the benefit of parallel access to multiple disks. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.
- With multiple disks, we can improve the transfer rate as well (or instead) by **striping data** across multiple disks.

### **Bit-level striping :**

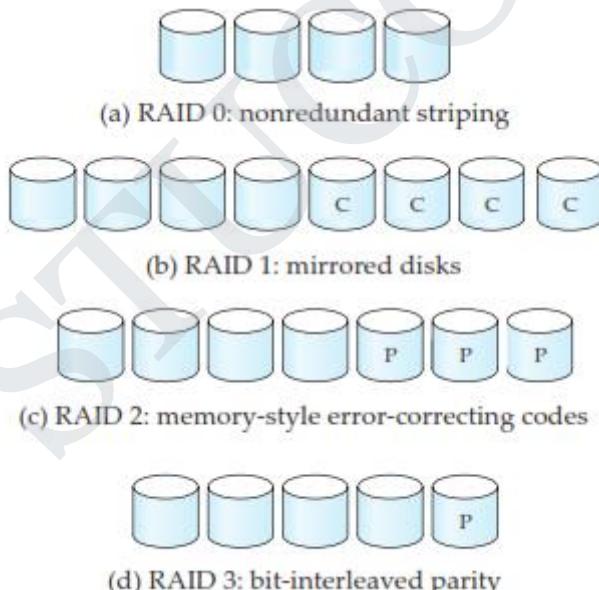
- Data striping consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.
- For example, if we have an array of eight disks, we write bit  $i$  of each byte to disk  $i$ . The array of eight disks can be treated as a single disk with sectors that are eight times the normal size, and, more important, that has eight times the transfer rate. In such an organization, every disk participates in every access (read or write), so the number of accesses that can be processed per second is about the same as on a single disk, but each access can read eight times as many data in the same time as on a single disk. Bit-level striping can be generalized to a number of disks that either is a multiple of 8 or a factor of 8. For example, if we use an array of four disks, bits  $i$  and  $4 + i$  of each byte go to disk  $i$ .

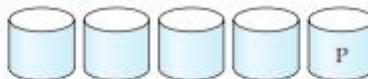
**Block-level striping :**

- **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  $(i \bmod n) + 1$ ; it uses the  $i/n$  th physical block of the disk to store logical block  $i$ .
- For example, with 8 disks, logical block 0 is stored in physical block 0 of disk 1, while logical block 11 is stored in physical block 1 of disk 4. When reading a large file, block-level striping fetches  $n$  blocks at a time in parallel from the  $n$  disks, giving a high data-transfer rate for large reads. When a single block is read, the data-transfer rate is the same as on one disk, but the remaining  $n - 1$  disks are free to perform other actions.
- Block-level striping is the most commonly used form of data striping. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible.

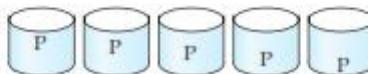
There are two main goals of parallelism in a disk system:

- 1) Load-balance multiple small accesses (block accesses), so that the throughput of such accesses increases.
- 2) Parallelize large accesses so that the response time of large accesses is reduced.

**RAID Levels:**



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy

**Figure. RAID levels.**

- Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability. Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with “parity” bits. These schemes have different cost – performance trade-offs. The schemes are classified into **RAID levels**.
- In the figure, P indicates error-correcting bits, and C indicates a second copy of the data.
- For all levels, the figure depicts four disks’ worth of data, and the extra disks depicted are used to store redundant information for failure recovery.

**RAID level 0:**

- **RAID level 0** refers 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** refers to disk mirroring with block striping. Figure(b) shows a mirrored organization that holds four disks’ worth of data.
- Note that some vendors use the term **RAID level 1+0** or **RAID level 10** to refer to mirroring with striping, and use the term RAID level 1 to refer to mirroring without striping. Mirroring without striping can also be used with arrays of disks, to give the appearance of a single large, reliable disk: if each disk has  $M$  blocks, logical blocks 0 to  $M - 1$  are stored on disk 0,  $M$  to  $2M - 1$  on disk 1(the second disk), and so on, and each disk is mirrored.

**RAID level 2:**

- **RAID level 2** known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for error detection and correction. Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte that are set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte gets damaged (either a

1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity. Thus, all 1-bit errors will be detected by the memory system. Error-correcting schemes store 2 or more extra bits, and can reconstruct the data if a single bit gets damaged.

- The idea of error-correcting codes can be used directly in disk arrays by striping bytes across disks. For example, the first bit of each byte could be stored in disk 0, the second bit in disk 1, and so on until the eighth bit is stored in disk 7, and the error-correction bits are stored in further disks.
- Figure (c) shows the level 2 scheme. The disks labeled  $P$  store the error-correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks, and can be used to reconstruct the damaged data. Figure 10.3c shows an array of size 4; note RAID level 2 requires only three disks' overhead for four disks of data, unlike RAID level 1, which required four disks' overhead.

#### RAID level 3:

- **RAID level 3** bit-interleaved parity organization, improves on level 2 by exploiting the fact that disk controllers, unlike memory systems, can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows: If one of the sectors gets damaged, the system knows exactly which sector it is, and, for each bit in the sector, the system can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.
- RAID level 3 is as good as level 2, but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice. Figure (d) shows the level 3 scheme.
- RAID level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas level 1 needs one mirror disk for every disk, and thus level 3 reduces the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with  $N$ -way striping of data, the transfer rate for reading or writing a single block is  $N$  times faster than a RAID level 1 organization using  $N$ -way striping. On the other hand, RAID level 3 supports a lower number of I/O operations per second, since every disk has to participate in every I/O request.

#### RAID level 4:

- **RAID level 4** block-interleaved parity organization, uses block-level striping, like RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from  $N$  other disks. This scheme is shown pictorially in Figure (e). If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.
- A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes

- 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  $P_k$ , for logical blocks  $4k, 4k + 1, 4k + 2, 4k + 3$  is stored in disk  $k \bmod 5$ ; 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)

- The time to rebuild the data of a failed disk can be significant, and it varies with the RAID level that is used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk. The **rebuild performance** of a RAID system may be an important factor if continuous availability of data is required, as it is in high-performance database systems. Furthermore, since rebuild time can form a significant part of the repair time, rebuild performance also influences the mean time to data loss.
- RAID level 0 is used in high-performance applications where data safety is not critical. Since RAID levels 2 and 4 are subsumed by RAID levels 3 and 5, the choice of RAID levels is restricted to the remaining levels. Bit striping (level 3) is inferior to block striping (level 5), since block striping gives as good data-transfer rates for large transfers, while using fewer disks for small transfers.
- For small transfers, the disk access time dominates anyway, so the benefit of parallel reads diminishes. In fact, level 3 may perform worse than level 5 for a small transfer, since the transfer completes only when corresponding sectors on all disks have been fetched; the average latency for the disk array thus becomes very close to the worst-case latency for a single disk, negating the benefits of higher transfer rates. Level 6 is not supported currently by many RAID implementations, but it offers better reliability than level 5 and can be used in applications where data safety is very important.
- The choice between RAID level 1 and level 5 is harder to make. RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance. RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes. For applications where data are read frequently, and written rarely, level 5 is the preferred choice.
- Disk-storage capacities have been growing at a rate of over 50 percent per year for many years, and the cost per byte has been falling at the same rate. As a result, for many existing database applications with moderate storage requirements, the monetary cost of the extra disk storage needed for mirroring has become relatively small (the extra monetary cost, however, remains a significant issue for storage-intensive applications such as video data storage). Access speeds have improved at a much slower rate (around a factor of 3 over 10 years), while the number of I/O operations required per second has increased tremendously, particularly for Web application servers.
- RAID level 5, which increases the number of I/O operations needed to write a single logical block, pays a significant time penalty in terms of write performance. RAID level 1 is therefore the RAID level of choice for many applications with moderate storage requirements and high I/O requirements.
- RAID system designers have to make several other decisions as well. For example, how many disks should there be in an array? How many bits should be protected by each parity bit? If there are more disks in an array, data-transfer rates are higher, but the system will be more expensive. If there are more bits protected by a parity bit, the space overhead due to parity bits is lower, but there is an increased chance that a second disk will fail before the first failed disk is repaired, and that will result

**Hardware Issues:**

- Another issue in the choice of RAID implementations is at the level of hardware. RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called **software RAID**. However, there are significant benefits to be had by building special-purpose hardware to support RAID, which we outline below; systems with special hardware support are called **hardware RAID** systems.
- Hardware RAID implementations can use nonvolatile RAM to record writes before they are performed. In case of power failure, when the system comes back up, it retrieves information about any incomplete writes from nonvolatile RAM and then completes the writes. Without such hardware support, extra work needs to be done to detect blocks that may have been partially written before power failure.
- Even if all writes are completed properly, there is a small chance of a sector in a disk becoming unreadable at some point, even though it was successfully written earlier. Reasons for loss of data on individual sectors could range from manufacturing defects, to data corruption on a track when an adjacent track is written repeatedly. Such loss of data that were successfully written earlier is sometimes referred to as a *latent failure*, or as *bit rot*. When such a failure happens, if it is detected early the data can be recovered from the remaining disks in the RAID organization. However, if such a failure remains undetected, a single disk failure could lead to data loss if a sector in one of the other disks has a latent failure.
- To minimize the chance of such data loss, good RAID controllers perform **scrubbing**; that is, during periods when disks are idle, every sector of every disk is read, and if any sector is found to be unreadable, the data are recovered from the remaining disks in the RAID organization, and the sector is written back. (If the physical sector is damaged, the disk controller would remap the logical sector address to a different physical sector on disk.)
- Some hardware RAID implementations permit **hot swapping**; that is, faulty disks can be removed and replaced by new ones without turning power off. Hot swapping reduces the mean time to repair, since replacement of a disk does not have to wait until a time when the system can be shut down. In fact many critical systems today run on a  $24 \times 7$  schedule; that is, they run 24 hours a day, 7 days a week, providing no time for shutting down and replacing a failed disk. Further, many RAID implementations assign a spare disk for each array (or for a set of disk arrays). If a disk fails, the spare disk is immediately used as a replacement. As a result, the mean time to repair is reduced greatly, minimizing the chance of any data loss. The failed disk can be replaced at leisure.
- The power supply, or the disk controller, or even the system interconnection in a RAID system could become a single point of failure that could stop functioning of the RAID system. To avoid this possibility, good RAID implementations have multiple redundant power supplies (with battery backups so they continue to function even if power fails). Such RAID systems have multiple disk interfaces, and multiple interconnections to connect the RAID system to the computer system (or to a network of computer systems). Thus, failure of any single component will not bring down the RAID system.

### **Other RAID Applications:**

- The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, the RAID structures are able to recover data even if one of the tapes in an array of tapes is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units.

### **4.2. FILE ORGANIZATION:**

- A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks.
- A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files.
- Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer. Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created. Larger block sizes can be useful in some database applications.
- A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used. We shall assume that *no record is larger than a block*. This assumption is realistic for most data-processing applications, such as our university example. There are certainly several kinds of large data items, such as images, that can be significantly larger than a block. We briefly discuss how to handle such large data items later, in Section 10.5.2, by storing large data items separately, and storing a pointer to the data item in the record.
- In addition, we shall require that *each record is entirely contained in a single block*; that is, no record is contained partly in one block, and partly in another. This restriction simplifies and speeds up access to data items.
- In a relational database, tuples of distinct relations are generally of different sizes. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records; however, files of fixed-length records are easier to implement than are files of variable-length records. Many of the techniques used for the former can be applied to the variable-length case. Thus, we begin by considering a file of fixed-length records, and consider storage of variable-length records later.

### **1) Fixed-Length Records:**

- Consider a file of *instructor* records for university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record
 ID varchar (5);
 name varchar(20);
 dept name varchar (20);
 salary numeric (8,2);
end
```

- Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes. Suppose that instead of allocating a variable amount of bytes for the attributes *ID*, *name*, and *dept name*, we allocate the maximum number of bytes that each attribute can hold. Then, the *instructor* record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on.

### **Two problems with this simple approach:**

1) Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

2) It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 3  | 22222 | Einstein   | Physics    | 95000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 45565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | CaliPeri   | 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 | CaliPeri   | 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 | CaliPeri   | 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.

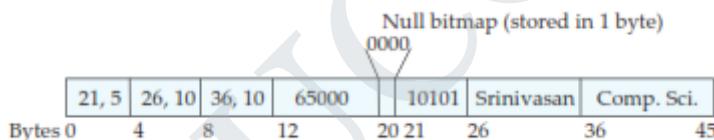
## 2) Variable-Length Records:

Variable-length records arise in database systems in several ways:

- Storage of multiple record types in a file.

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.

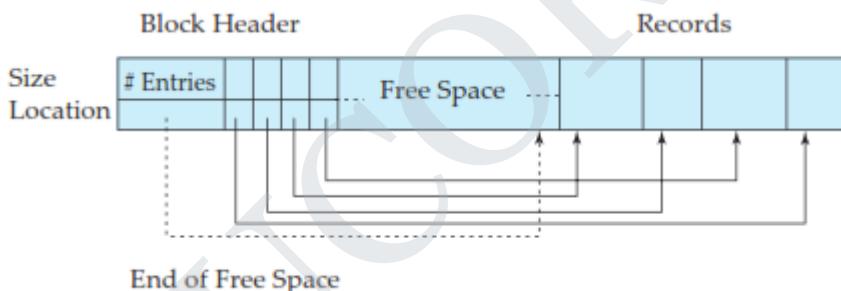


**Figure.** Representation of variable-length record.

- 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<sup>+</sup>-tree file organizations. B<sup>+</sup>-tree file organizations permit us to read an entire object, or specified byte ranges in the object, as well as to insert and delete parts of the object.

#### **4.3. ORGANIZATION OF RECORDS IN FILES:**

- A relation is a set of records.

Possible ways of organizing records in files are:

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

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

**3) Hashing file organization.** A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed. Generally, a separate file is used to store the records of each relation.

4) In a **multitable clustering file organization**, records of several different relations are stored in the same file; further, related records of the different relations are stored on the same block, so that one I/O operation fetches related records from all the relations. For example, records of the two relations can be considered to be related if they would match in a join of the two relations.

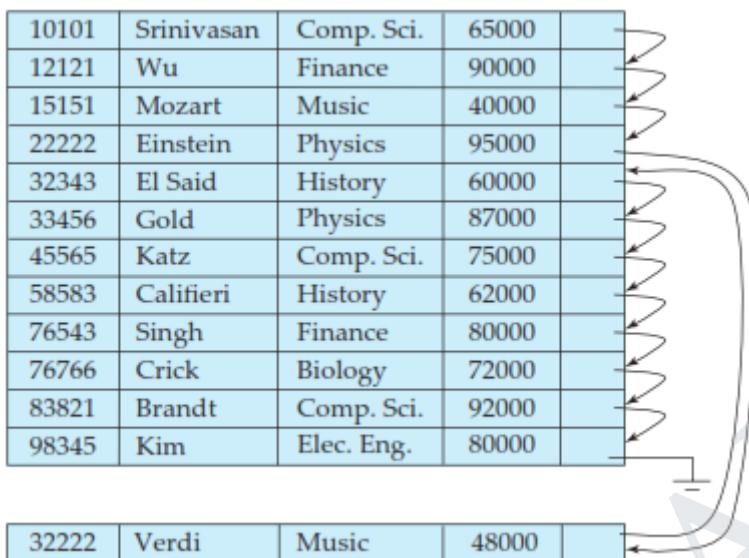
##### **1) Sequential File Organization:**

- A **sequential file** is designed for efficient processing of records in sorted order based on some search key. A **search key** is any attribute or set of attributes; it need not be the primary key, or even a super key.
- To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

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

**Figure.** Sequential file for *instructor* records.

- Figure shows a sequential file of *instructor* records taken from our university example. In that example, the records are stored in search-key order, using *ID* as the search key.
- The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms.
- It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:
  - i) Locate the record in the file that comes before the record to be inserted in search-key order.
  - ii) If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.

**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.

| <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|------------------|-----------------|---------------|
| Comp. Sci.       | Taylor          | 100000        |
| Physics          | Watson          | 70000         |

**Figure.** The *department* relation.

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 10101     | Srinivasan  | Comp. Sci.       | 65000         |
| 33456     | Gold        | Physics          | 87000         |
| 45565     | Katz        | Comp. Sci.       | 75000         |
| 83821     | Brandt      | Comp. Sci.       | 92000         |

**Figure.** The *instructor* relation.

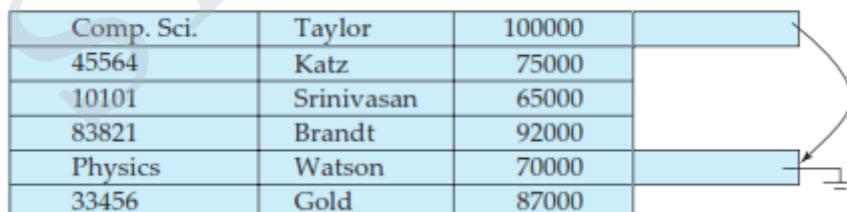
- As a concrete example, consider the *department* and *instructor* relations. In Figure, we show a file structure designed for efficient execution of queries involving the natural join of *department* and *instructor*. The *instructor* tuples for each *ID* are

stored near the *department* tuple for the corresponding *dept name*. This structure mixes together tuples of two relations, but allows for efficient processing of the join.

- When a tuple of the *department* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding *instructor* tuples are stored on the disk near the *department* tuple, the block containing the *department* tuple contains tuples of the *instructor* relation needed to process the query. If a department has so many instructors that the *instructor* records do not fit in one block, the remaining records appear on nearby blocks.
- A **multitable clustering file organization** is a file organization, such as that illustrated in Figure, that stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently.
- In the representation shown in Figure, the *dept name* attribute is omitted from *instructor* records since it can be inferred from the associated *department* record; the attribute may be retained in some implementations, to simplify access to the attributes. We assume that each record contains the identifier of the relation to which it belongs, although this is not shown in Figure.

|            |            |        |
|------------|------------|--------|
| Comp. Sci. | Taylor     | 100000 |
| 45564      | Katz       | 75000  |
| 10101      | Srinivasan | 65000  |
| 83821      | Brandt     | 92000  |
| Physics    | Watson     | 70000  |
| 33456      | Gold       | 87000  |

**Figure.** Multitable clustering file structure.



**Figure.** Multitable clustering file structure with pointer chains.

**select \* from department;**

- Our use of clustering of multiple tables into a single file has enhanced processing of a particular join (that of *department* and *instructor*), but it results in slowing processing of other types of queries. For example, requires more block accesses than it did in the scheme under which we stored each relation in a separate file, since each block now contains significantly fewer *department* records. To locate efficiently all tuples of the *department* relation in the structure of Figure, we can chain together all the records of that relation using pointers.
- When multitable clustering is to be used depends on the types of queries that the database designer believes to be most frequent. Careful use of multitable clustering can produce significant performance gains in query processing.

#### **4.4. INDEXING AND HASHING:**

- Many queries reference only a small proportion of the records in a file. For example, a query like “Find all instructors in the Physics department” or “Find the total number of credits earned by the student with *ID* 22201” references only a fraction of the student records.
- It is inefficient for the system to read every tuple in the *instructor* relation to check if the *dept name* value is “Physics”. Likewise, it is inefficient to read the entire *student* relation just to find the one tuple for the *ID* “32556.” Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

#### **Basic Concepts:**

- An index for a file in a database system works in much the same way as the index in the textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking. The words in the index are in sorted order, making it easy to find the word we want. Moreover, the index is much smaller than the book, further reducing the effort needed.
- Database-system indices play the same role as book indices in libraries. For example, to retrieve a *student* record given an *ID*, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate *student* record.
- Keeping a sorted list of students’ *ID* would not work well on very large databases with thousands of students, since the index would itself be very big; further, even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used.

There are two basic kinds of indices:

**1) Ordered indices:** Based on a sorted ordering of the values.

**2) Hash indices:** Based on a uniform distribution of values across a range of buckets.

The bucket to which a value is assigned is determined by a function, called a *hash function*.

- We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications.

Each technique must be evaluated on the basis of these factors:

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

**2) Access time:** The time it takes to find a particular data item, or set of items, using the technique in question.

**3) Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.

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

**5) Space overhead:** The additional space occupied by an index structure. Provided 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**.

#### **4.5. 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.

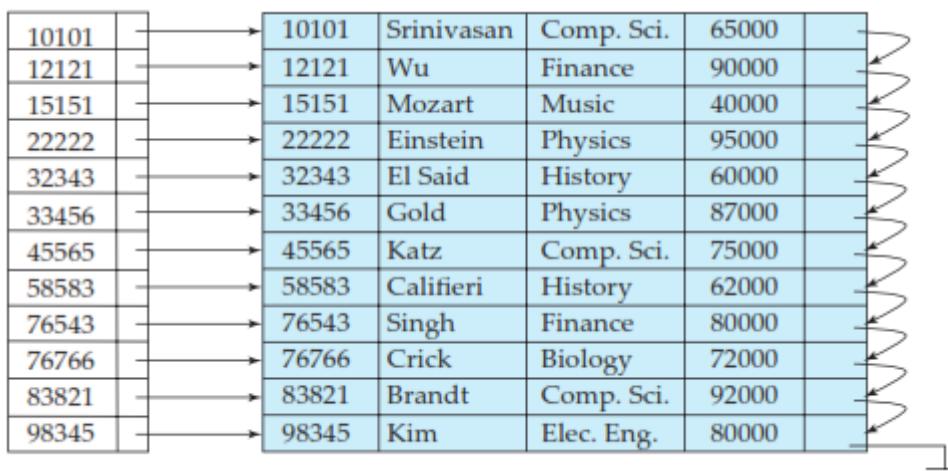
|       |            |            |       |  |
|-------|------------|------------|-------|--|
| 10101 | Srinivasan | Comp. Sci. | 65000 |  |
| 12121 | Wu         | Finance    | 90000 |  |
| 15151 | Mozart     | Music      | 40000 |  |
| 22222 | Einstein   | Physics    | 95000 |  |
| 32343 | El Said    | 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 |  |

The diagram illustrates a sequential file of *instructor* records. The records are organized by their *ID* in ascending order. Each record consists of five fields: *ID*, Name, Department, and Salary. Arrows point from the *ID* field of each record to a vertical stack of pointers on the right side of the table. This indicates that the records are physically stored in sorted order of the search key (*ID*), which is used as the search key for the index.

**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.

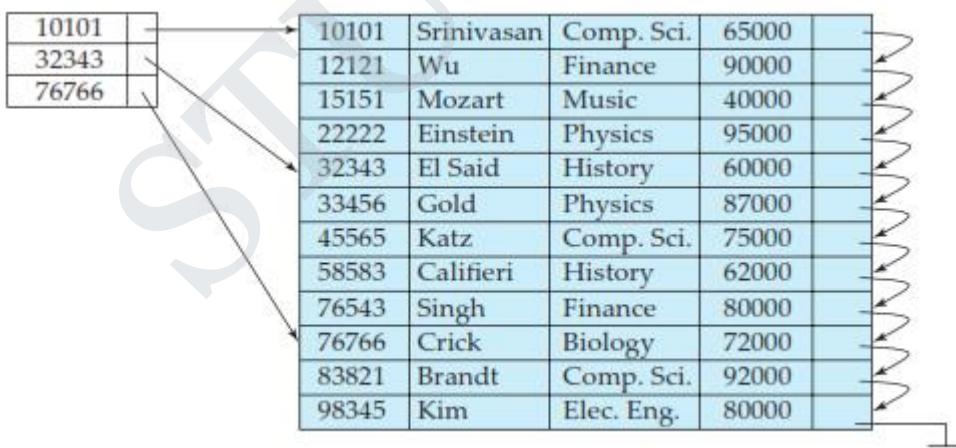
**Figure.**Dense index.

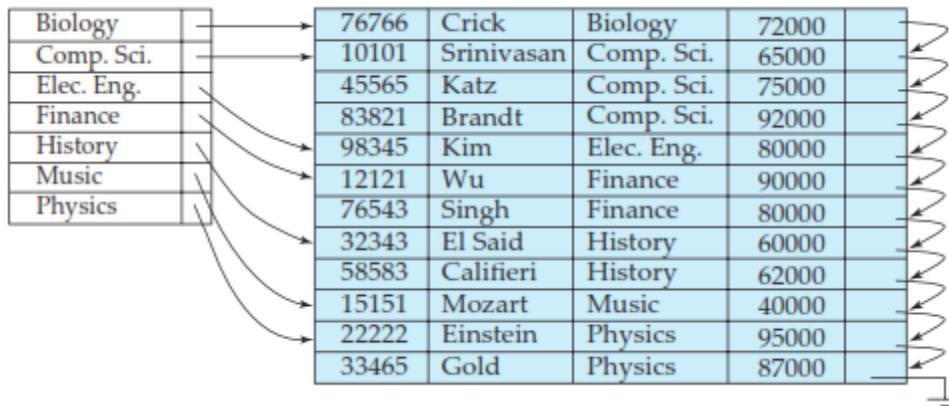
There are two types of ordered indices that we can use:

**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.

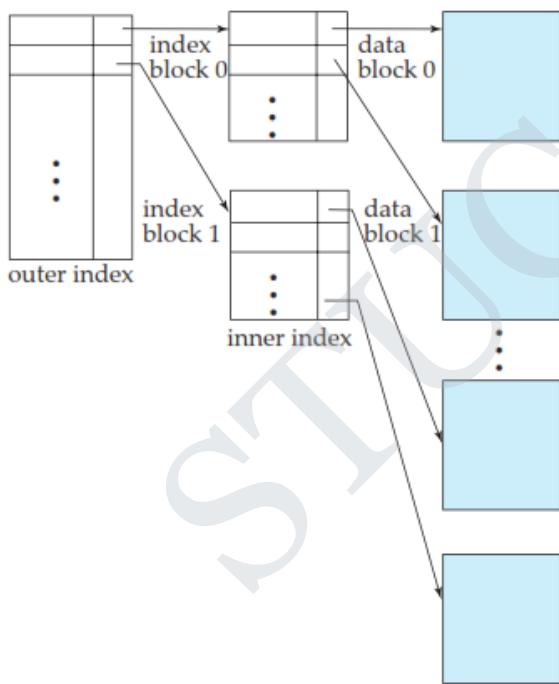


**Figure.** Dense index with search key *dept name*.

- 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.
- Consider a (printed) dictionary. The header of each page lists the first word alphabetically on that page. The words at the top of each page of the book index together form a sparse index on the contents of the dictionary pages.
- As another example, suppose that the search-key value is not not a primary key. Figure shows a dense clustering index for the *instructor* file with the search key being *dept name*. Observe that in this case the *instructor* file is sorted on the search key *dept name*, instead of *ID*, otherwise the index on *dept name* would be a nonclustering index. Suppose that we are looking up records for the History department. Using the dense index of Figure, we follow the pointer directly to the first History record. We process this record, and follow the pointer in that record to locate the next record in search-key (*dept name*) order. We continue processing records until we encounter a record for a department other than History.
- It is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.
- There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block. The reason this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block, we minimize block accesses while keeping the size of the index (and thus our space overhead) as small as possible.
- For the preceding technique to be fully general, we must consider the case where records for one search-key value occupy several blocks. It is easy to modify our scheme to handle this situation.

**MULTILEVEL INDICES:**

- Suppose we build a dense index on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4 kilobyte block. Thus, our index occupies 10,000 blocks. If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space. Such large indices are stored as sequential files on disk.
- If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.
- Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index would occupy  $b$  blocks, binary search requires as many as  $\log_2(b)$  blocks to be read. ( $x$  denotes the least integer that is greater than or equal to  $x$ ; that is, we round upward.) For a 10,000-block index, binary search requires 14 block reads.
- On a disk system where a block read takes on average 10 milliseconds, the index search will take 140 milliseconds. This may not seem much, but we would be able to carry out only seven index searches a second, whereas a more efficient search mechanism would let us carry out far more searches per second, as we shall see shortly. Note that, if overflow blocks have been used, binary search is not possible. In that case, a sequential search is typically used, and that requires  $b$  block reads, which will take even longer. Thus, the process of searching a large index may be costly.

**Figure.**Two-level sparse index.

- To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse outer index on the original index, which we now call the inner index, as shown in Figure 11.5. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the

largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

- In our example, an inner index with 10,000 blocks would require 10,000 entries in the outer index, which would occupy just 100 blocks. If we assume that the outer index is already in main memory, we would read only one index block for a search using a multilevel index, rather than the 14 blocks we read with binary search. As a result, we can perform 14 times as many index searches per second.
- If our file is extremely large, even the outer index may grow too large to fit in main memory. With a 100,000,000 tuple relation, the inner index would occupy 1,000,000 blocks, and the outer index occupies 10,000 blocks, or 40 megabytes. Since there are many demands on main memory, it may not be possible to reserve that much main memory just for this particular outer index. In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel** indices. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search.

### 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. Such a record update can be modeled as a deletion of the old record, followed by an insertion of the new value of the record, which results in an index deletion followed by an index insertion. As a result we only need to consider insertion and deletion on an index, and do not need to consider updates explicitly.

**1) Insertion.** First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:

#### 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 record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
- Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.
- Insertion and deletion algorithms for multilevel indices are a simple extension of the scheme just described. On deletion or insertion, the system updates the lowest-level index as described. As far as the second level is concerned, the lowest-level index is merely a file containing records—thus, if there is any change in the lowest-level index, the system updates the second-level index as described. The same technique applies to further levels of the index, if there are any.

**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, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.
- A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from clustering indices. If the search key of a clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.
- In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.
- We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each

points to a bucket that contains pointers to the file. Figure 11.6 shows the structure of a secondary index that uses an extra level of indirection on the *instructor* file, on the search key *salary*.

- A sequential scan in clustering index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered by both the search key of the clustering index and the search key of a secondary index.

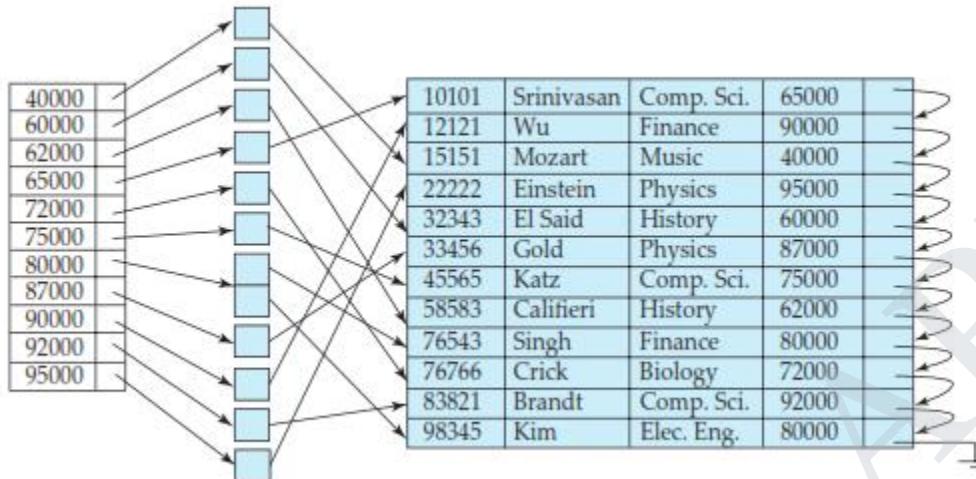


Figure . Secondary index on *instructor* file, on noncandidate key *salary*.

- Because secondary-key order and physical-key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.
- The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, *every* index must be updated.
- Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

### Indices on Multiple Keys:

- Although the examples we have seen so far have had a single attribute in a search key, in general a search key can have more than one attribute. A search key containing more than one attribute is referred to as a **composite search key**. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form  $(a_1, \dots, a_n)$ , where the indexed attributes are  $A_1, \dots, A_n$ . The ordering of search-key values is the *lexicographic ordering*. For example, for the case of two attribute search keys,  $(a_1, a_2) < (b_1, b_2)$  if either  $a_1 < b_1$  or  $a_1 = b_1$  and  $a_2 < b_2$ . Lexicographic ordering is basically the same as alphabetic ordering of words.
- As an example, consider an index on the *takes* relation, on the composite search key (*course id*, *semester*, *year*). Such an index would be useful to find all students who have registered for a particular course in a particular semester/year. An ordered index on a composite key can also be used to answer several other kinds of queries efficiently.

#### **4.6. B<sup>+</sup>-TREE INDEX FILES:**

- The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.
- The **B<sup>+</sup> -tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B<sup>+</sup>-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between  $n/2$  and  $n$  children, where  $n$  is fixed for a particular tree.
- We shall see that the B<sup>+</sup>-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<sup>+</sup>-tree structure.

#### **Structure of a B<sup>+</sup>-Tree:**

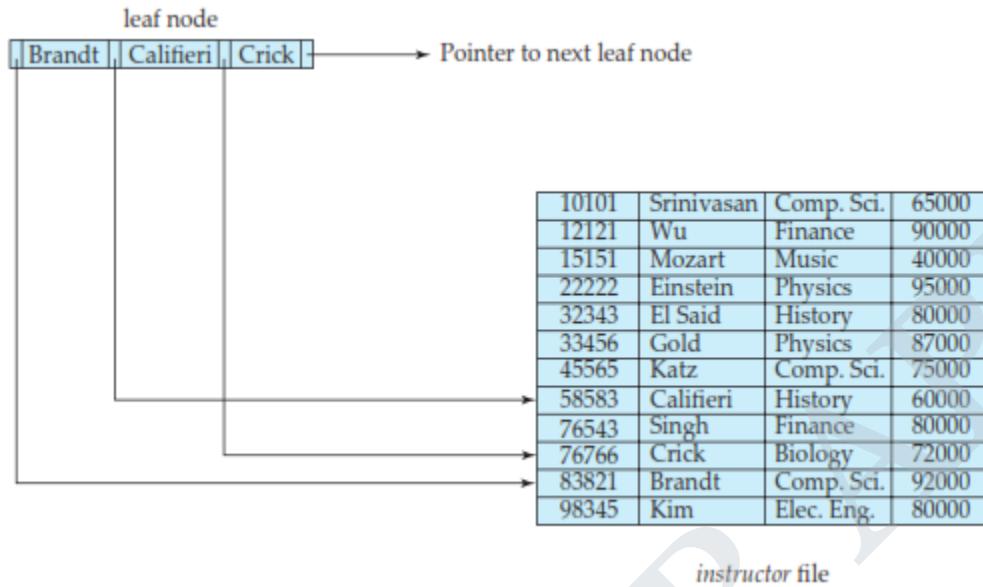
- A B<sup>+</sup>-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure shows a typical node of a B<sup>+</sup>-tree. It contains up to  $n - 1$  search-key values  $K_1, K_2, \dots, K_{n-1}$ , and  $n$  pointers  $P_1, P_2, \dots, P_n$ . 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  $i = 1, 2, \dots, n-1$ , 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<sup>+</sup>-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<sup>+</sup>-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<sup>+</sup>-tree index is used as a dense index (as is usually the case) every search-key value must appear in some leaf node.

|       |       |       |   |           |           |       |
|-------|-------|-------|---|-----------|-----------|-------|
| $P_1$ | $K_1$ | $P_2$ | É | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
|-------|-------|-------|---|-----------|-----------|-------|

**Figure.** Typical node of a B<sup>+</sup> -tree.

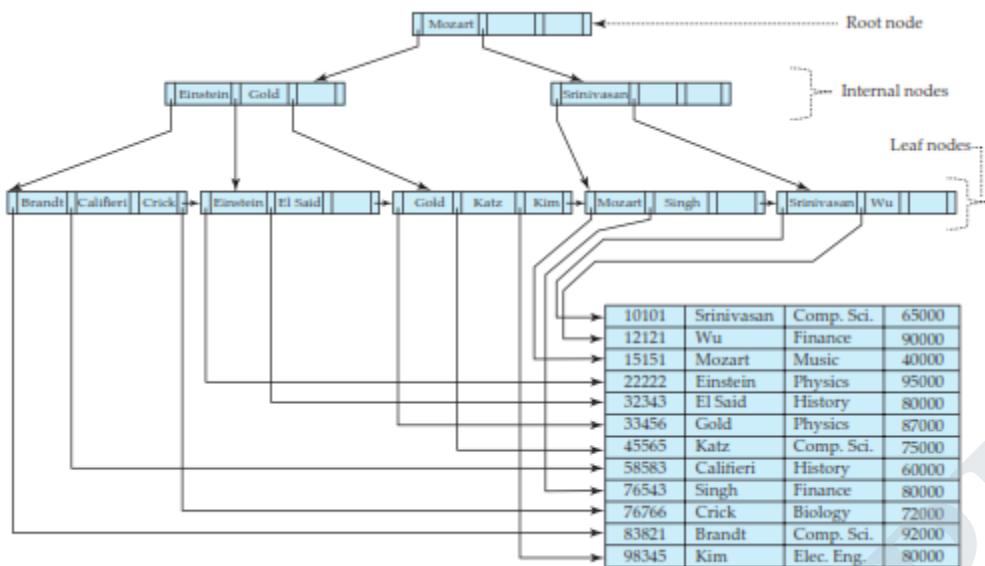
- Now we can explain the use of the pointer  $P_n$ . Since there is a linear order on the leaves based on the search-key values that they contain, we use  $P_n$  to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.
- The **nonleaf nodes** of the B<sup>+</sup>-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  $i = 2, 3, \dots, m - 1$ , 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  $K_1$ .



**Figure.** A leaf node for *instructor* B<sup>+</sup>-tree index ( $n = 4$ ).

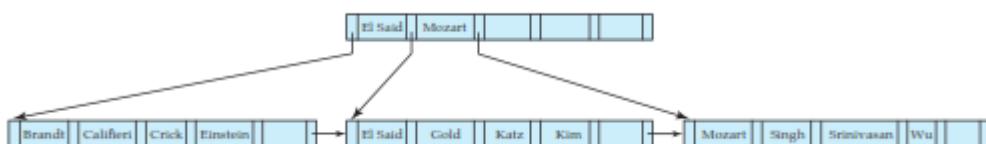
- Unlike other nonleaf nodes, the root node can hold fewer than  $n/2$  pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B<sup>+</sup>-tree, for any  $n$ , that satisfies the preceding requirements.
- Figure shows a complete B<sup>+</sup>-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:

- Let us consider how we process queries on a  $B^+$ -tree. Suppose that we wish to find records with a search-key value of  $V$ . Figure 11.11 presents pseudocode for a function `find()` to carry out this task.
- Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest  $i$  such that search-key value  $K_i$  is greater than or equal to  $V$ . Suppose such a value is found; then, if  $K_i$  is equal to  $V$ , the current node is set to the node pointed to by  $P_{i+1}$ , otherwise  $K_i > V$ , and the current node is set to the node pointed to by  $P_i$ . If no such value  $K_i$  is found, then clearly  $V > K_{m-1}$ , where  $P_m$  is the last nonnull pointer in the node. In this case the current node is set to that pointed to by  $P_m$ . The above procedure is repeated, traversing down the tree until a leaf node is reached.

**Figure.**  $B^+$  -tree for *instructor* file with  $n = 6$ .

```

function find(value V)
/* Returns leaf node C and index i such that C.Pi points to first record

 with search key value V */ Set C = root node
while (C is not a leaf node) begin
Let i = smallest number such that V ≤ C.Ki if there is no such number i then begin

Let Pm = last non-null pointer in the node Set C = C.Pm

end
else if (V = C.Ki)
then Set C = C.Pi+1
else C = C.Pi /* V < C.Ki */
end
/* C is a leaf node */
Let i be the least value such that Ki = V
if there is such a value i
then return (C, i)
else return null ; /* No record with key value V exists*/

```

```

procedure printAll(value V)
/* prints all records with search key value V */
Set done = false;
Set (L, i) = find(V);
if ((L, i) is null) return
repeat
repeat
Print record pointed to by L.Pi

Set i = i + 1
until (i > number of keys in L or L.Ki > V)
if (i > number of keys in L)
then L = L.Pn
else Set done = true;
until (done or L is null)

```

**Figure.** Querying a  $B^+$ -tree.

- At the leaf node, if there is a search-key value equal to *V*, let *K<sub>i</sub>* be the first such value; pointer *P<sub>i</sub>* directs us to a record with search-key value *K<sub>i</sub>*. The function
- then returns the leaf node *L* and the index *i*. If no search-key with value *V* is found in the leaf node, no record with key value *V* exists in the relation, and function find returns null, to indicate failure.
- If there is at most one record with a search key value *V* (for example, if the index is on a primary key) the procedure that calls the find function simply uses the pointer *L.P<sub>i</sub>* to retrieve the record and is done. However, in case there may be more than one matching record, the remaining records also need to be fetched.

- Procedure printAll shown in Figure shows how to fetch all records with a specified search key  $V$ . The procedure first steps through the remaining keys in the node  $L$ , to find other records with search-key value  $V$ . If node  $L$  contains at least one search-key value greater than  $V$ , then there are no more records matching  $V$ . Otherwise, the next leaf, pointed to by  $P_n$  may contain further entries for  $V$ . The node pointed to by  $P_n$  must then be searched to find further records with search-key value  $V$ . If the highest search-key value in the node pointed to by  $P_n$  is also  $V$ , further leaves may have to be traversed, in order to find all matching records. The **repeat** loop in printAll carries out the task of traversing leaf nodes until all matching records have been found.
- A real implementation would provide a version of find supporting an iterator interface similar to that provided by the JDBC ResultSet. Such an iterator interface would provide a method next(), which can be called repeatedly to fetch successive records with the specified search-key. The next() method would step through the entries at the leaf level, in a manner similar to printAll, but each call takes only one step, and records where it left off, so that successive calls next step through successive records. We omit details for simplicity, and leave the pseudocode for the iterator interface as an exercise for the interested reader.
- $B^+$ -trees can also be used to find all records with search key values in a specified range ( $L, U$ ). For example, with a  $B^+$ -tree on attribute *salary* of *instructor*, we can find all *instructor* records with salary in a specified range such as (50000, 100000) (in other words, all salaries between 50000 and 100000). Such queries are called **range queries**. To execute such queries, we can create a procedure printRange( $L, U$ ), whose body is the same as printAll except for these differences: printRange calls find( $L$ ), instead of find( $V$ ), and then steps through records as in procedure printAll, but with the stopping condition being that  $L.K_i > U$ , instead of  $L.K_i > V$ .
- In processing a query, we traverse a path in the tree from the root to some leaf node. If there are  $N$  records in the file, the path is no longer than  $\log_{n/2}(N)$ .
- In practice, only a few nodes need to be accessed. Typically, a node is made to be the same size as a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of 8 bytes,  $n$  is around 200. Even with a more conservative estimate of 32 bytes for the search-key size,  $n$  is around 100. With  $n = 100$ , if we have 1 million search-key values in the file, a lookup requires only  $\log_{50}(1,000,000) = 4$  nodes to be accessed. Thus, at most four blocks need to be read from disk for the lookup. The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.
- An important difference between  $B^+$ -tree structures and in-memory tree structures, such as binary trees, is the size of a node, and as a result, the height of the tree. In a binary tree, each node is small, and has at most two pointers. In a  $B^+$ -tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus,  $B^+$ -trees tend to be fat and short, unlike thin and tall binary trees. In a balanced binary tree, the path for a lookup can be of length  $\log_2(N)$ , where  $N$  is the number of records in the file being indexed. With  $N = 1,000,000$  as in the previous example, a balanced binary tree requires around 20 node accesses. If each node were on a different disk block, 20 block reads would be required to process a lookup, in contrast to the four block reads for the  $B^+$ -tree. The difference is significant, since each block read could require a disk arm seek, and a block read together with the disk arm seek takes about 10 milliseconds on a typical disk.

### Updates on $B^+$ -Trees:

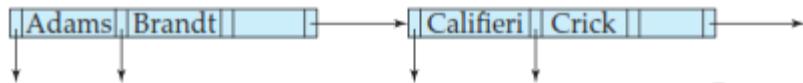
- When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly. Recall that updates to a record can be modeled as a deletion of the old record followed by insertion of the updated record. Hence we only consider the case of insertion and deletion.

- Insertion and deletion are more complicated than lookup, since it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce** nodes (that is, combine nodes) if a node becomes too small (fewer than  $n/2$  pointers). Furthermore, when a node is split or a pair of nodes is combined, we must ensure that balance is preserved. To introduce the idea behind insertion and deletion in a  $B^+$ -tree, we shall assume temporarily that nodes never become too large or too small. Under this assumption, insertion and deletion are performed as defined next.

**Insertion.** Using the same technique as for lookup from the `find()` function (Figure 11.11), we first find the leaf node in which the search-key value would appear. We then insert an entry (that is, a search-key value and record pointer pair) in the leaf node, positioning it such that the search keys are still in order.

**Deletion.** Using the same technique as for lookup, we find the leaf node containing the entry to be deleted, by performing a lookup on the search-key value of the deleted record; if there are multiple entries with the same search-key value, we search across all entries with the same search-key value until we find the entry that points to the record being deleted. We then remove the entry from the leaf node. All entries in the leaf node that are to the right of the deleted entry are shifted left by one position, so that there are no gaps in the entries after the entry is deleted.

We now consider the general case of insertion and deletion, dealing with node splitting and node coalescing.

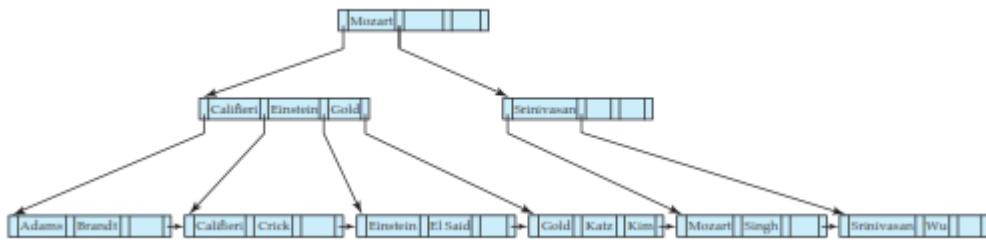


**Figure.** Split of leaf node on insertion of “Adams”

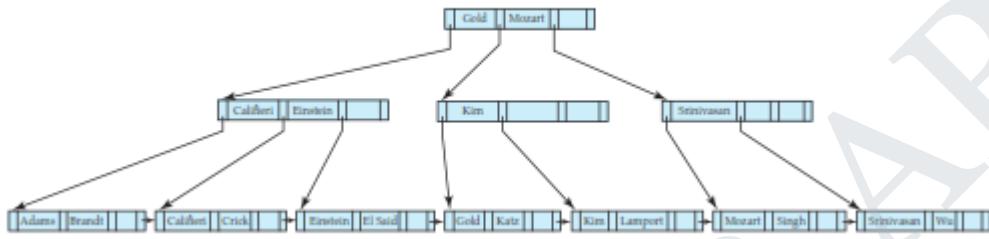
### Insertion:

- We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for “Adams” into the  $B^+$ -tree of Figure 11.9. Using the algorithm for lookup, we find that “Adams” should appear in the leaf node containing “Brandt”, “Califieri”, and “Crick.” There is no room in this leaf to insert the search-key value “Adams.” Therefore, the node is *split* into two nodes. Figure shows the two leaf nodes that result from the split of the leaf node on inserting “Adams”. The search-key values “Adams” and “Brandt” are in one leaf, and “Califieri” and “Crick” are in the other. In general, we take the  $n$  search-key values (the  $n - 1$  values in the leaf node plus the value being inserted), and put the first  $n/2$  in the existing node and the remaining values in a newly created node.
- Having split a leaf node, we must insert the new leaf node into the  $B^+$ -tree structure. In our example, the new node has “Califieri” as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The  $B^+$ -tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.
- Splitting of a nonleaf node is a little different from splitting of a leaf node. Figure shows the result of inserting a record with search key “Lamport” into the tree shown in Figure. The leaf node in which “Lamport” is to be inserted already has entries “Gold”, “Katz”, and “Kim”, and as a result the leaf node has to be split. The new right-hand-side node resulting from the split contains the search-key values “Kim” and “Lamport”. An entry (Kim,  $n1$ ) must then be added to the parent node, where  $n1$  is a pointer to the new node. However, there is no space in the parent node to add a new entry, and the parent node has

to be split. To do so, the parent node is conceptually expanded temporarily, the entry added, and the overfull node is then immediately split.



**Figure.** Insertion of “Adams” into the  $B^+$ -tree of Figure 11.9.



**Figure.** Insertion of “Lampert” into the  $B^+$ -tree of Figure 11.13.

- When an overfull nonleaf node is split, the child pointers are divided among the original and the newly created nodes; in our example, the original node is left with the first three pointers, and the newly created node to the right gets the remaining two pointers. The search key values are, however, handled a little differently. The search key values that lie between the pointers moved to the right node (in our example, the value “Kim”) are moved along with the pointers, while those that lie between the pointers that stay on the left (in our example, “Califieri” and “Einstein”) remain undisturbed.
- However, the search key value that lies between the pointers that stay on the left, and the pointers that move to the right node is treated differently. In our example, the search key value “Gold” lies between the three pointers that went to the left node, and the two pointers that went to the right node. The value “Gold” is not added to either of the split nodes. Instead, an entry (Gold,  $n_2$ ) is added to the parent node, where  $n_2$  is a pointer to the newly created node that resulted from the split. In this case, the parent node is the root, and it has enough space for the new entry.
- The general technique for insertion into a  $B^+$ -tree is to determine the leaf node  $l$  into which insertion must occur. If a split results, insert the new node into the parent of node  $l$ . If this insertion causes a split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.
- Figure 11.15 outlines the insertion algorithm in pseudocode. The procedure `insert` inserts a key-value pointer pair into the index, using two subsidiary procedures `insert in leaf` and `insert in parent`. In the pseudocode,  $L$ ,  $N$ ,  $P$  and  $T$  denote pointers to nodes, with  $L$  being used to denote a leaf node.  $L.K_i$  and  $L.P_i$  denote the  $i$ th value and the  $i$ th pointer in node  $L$ , respectively;  $T.K_i$  and  $T.P_i$  are used similarly. The pseudocode also makes use of the function `parent( $N$ )` to find the parent of a node  $N$ . We can compute a list of nodes in the path from the root to the leaf while initially finding the leaf node, and can use it later to find the parent of any node in the path efficiently.

- The procedure insert in parent takes as parameters  $N, K, N$ , where node  $N$  was split into  $N$  and  $N$ , with  $K$  being the least value in  $N$ . The procedure modifies the parent of  $N$  to record the split. The procedures insert into index and insert in parent use a temporary area of memory  $T$  to store the contents of a node being split. The procedures can be modified to copy data from the node being split directly to the newly created node, reducing the time required for copying data. However, the use of the temporary space  $T$  simplifies the procedures.

**procedure** insert(*value K, pointer P*)

**if** (tree is empty) create an empty leaf node  $L$ , which is also the root **else** Find the leaf node  $L$  that should contain key value  $K$

**if** ( $L$  has less than  $n - 1$  key values)  
**then** insert in leaf ( $L, K, P$ ) - - -

**else begin** /\*  $L$  has  $n - 1$  key values already, split it \*/ Create node  $L$

Copy  $L.P_1 \dots L.K_{n-1}$  to a block of memory  $T$  that can hold  $n$  (pointer, key-value) pairs  
insert in leaf ( $T, K, P$ ) - - -

Set  $L.P_n = L.P_n$ ; Set  $L.P_n = L$   
Erase  $L.P_1$  through  $L.K_{n-1}$  from  $L$   
Copy  $T.P_1$  through  $T.K_{n/2}$  from  $T$  into  $L$  starting at  $L.P_1$   
Copy  $T.P_{n/2+1}$  through  $T.K_n$  from  $T$  into  $L$  starting at  $L.P_1$   
Let  $K$  be the smallest key-value in  $L$   
insert in parent( $L, K, L$ ) - - -

**end**

**procedure** insert in leaf (*node L, value K, pointer P*)

**if** ( $K < L.K_1$ )  
**then** insert  $P, K$  into  $L$  just before  $L.P_1$   
**else begin**  
Let  $K_i$  be the highest value in  $L$  that is less than  $K$  Insert  $P, K$  into  $L$  just after  $T.K_i$   
**end**

**procedure** insert in parent(*node N, value K, node N*)

**if** ( $N$  is the root of the tree)  
**then begin**  
Create a new node  $R$  containing  $N, K, N$  /\*  $N$  and  $N$  are pointers \*/ Make  $R$  the root of the tree  
**return**  
**end**  
Let  $P = parent(N)$   
**if** ( $P$  has less than  $n$  pointers)  
**then** insert ( $K, N$ ) in  $P$  just after  $N$   
**else begin** /\* Split  $P$  \*/

Copy  $P$  to a block of memory  $T$  that can hold  $P$  and ( $K, N$ )  
Insert ( $K, N$ ) into  $T$  just after  $N$

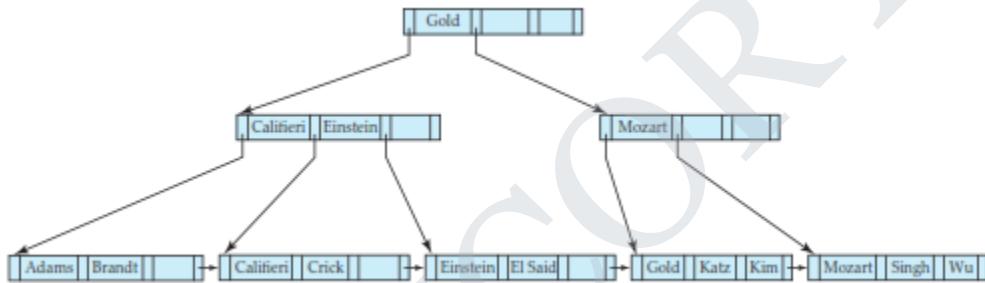
```

Erase all entries from P ; Create node P
Copy $T.P_1 \dots T.P_{n/2}$ into P
Let $K = T.K_{n/2}$
Copy $T.P_{n/2+1} \dots T.P_{n+1}$ into P
insert in parent(P, K, P)
 - - -
end

```

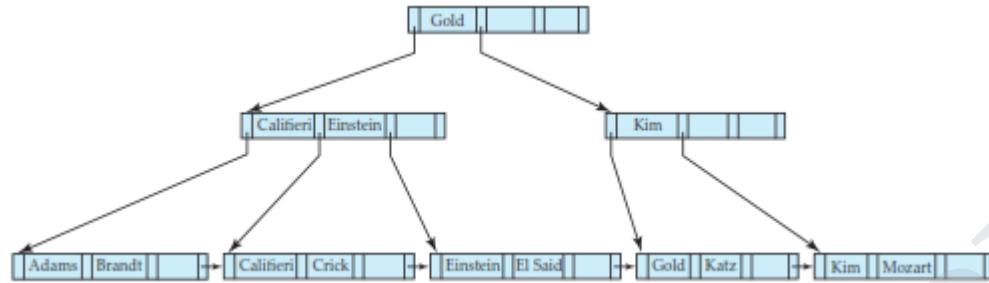
**Figure.** Insertion of entry in a  $B^+$ -tree.**Deletion:**

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete “Srinivasan” from the  $B^+$ -tree of Figure 11.13. The resulting  $B^+$ -tree appears in Figure. We now consider how the deletion is performed. We first locate the entry for “Srinivasan” by using our lookup algorithm. When we delete the entry for “Srinivasan” from its leaf node, the node is left with only one entry, “Wu”. Since, in our example,  $n = 4$  and  $1 < (n - 1)/2$ , we must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for “Wu” can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

**Figure.** Deletion of “Srinivasan” from the  $B^+$ -tree of Figure 11.13.

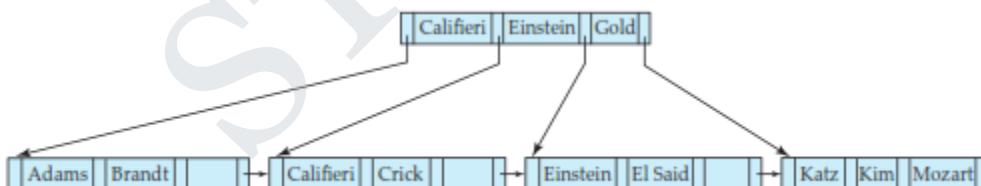
- In our example, the entry to be deleted is  $(\text{Srinivasan}, n_3)$ , where  $n_3$  is a pointer to the leaf containing “Srinivasan”. (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.)
- After deleting the above entry, the parent node, which had a search key value “Srinivasan” and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since  $1 < n/2$  for  $n = 4$ , the parent node is underfull. (For larger  $n$ , a node that becomes underfull would still have some values as well as pointers.)
- In this case, we look at a sibling node; in our example, the only sibling is the nonleaf node containing the search keys “Califieri”, “Einstein”, and “Gold”. If possible, we try to coalesce the node with its sibling. In this case, coalescing is not possible, since the node and its sibling together have five pointers, against a maximum of four. The solution in this case is to **redistribute** the pointers between the node and its sibling, such that each has at least  $n/2 = 2$  child pointers. To do so, we move the rightmost pointer from the left sibling (the one pointing to the leaf node containing “Mozart”) to the underfull right sibling. However, the underfull right sibling would now have two pointers, namely its leftmost pointer, and the newly moved pointer, with no value separating them.

- In fact, the value separating them is not present in either of the nodes, but is present in the parent node, between the pointers from the parent to the node and its sibling. In our example, the value “Mozart” separates the two pointers, and is present in the right sibling after the redistribution. Redistribution of the pointers also means that the value “Mozart” in the parent no longer correctly separates search-key values in the two siblings. In fact, the value that now correctly separates search-key values in the two sibling nodes is the value “Gold”, which was in the left sibling before redistribution.



**Figure.** Deletion of “Singh” and “Wu” from the B<sup>+</sup>-tree.

- As a result, as can be seen in the B<sup>+</sup>-tree in Figure 11.16, after redistribution of pointers between siblings, the value “Gold” has moved up into the parent, while the value that was there earlier, “Mozart”, has moved down into the right sibling.
- We next delete the search-key values “Singh” and “Wu” from the B<sup>+</sup>-tree of Figure. The deletion of the first of these values does not make the leaf node underfull, but the deletion of the second value does. It is not possible to merge the underfull node with its sibling, so a redistribution of values is carried out, moving the search-key value “Kim” into the node containing “Mozart”, resulting in the tree shown in Figure. The value separating the two siblings has been updated in the parent, from “Mozart” to “Kim”.
- Now we delete “Gold” from the above tree; the result is shown in Figure. This results in an underfull leaf, which can now be merged with its sibling. The resultant deletion of an entry from the parent node (the nonleaf node containing “Kim”) makes the parent underfull (it is left with just one pointer). This time around, the parent node can be merged with its sibling. This merge results in the search-key value “Gold” moving down from the parent into the merged node. As a result of this merge, an entry is deleted from its parent, which happens to be the root of the tree. And as a result of that deletion, the root is left with only one child pointer and no search-key value, violating the condition that the root have at least two children. As a result, the root node is deleted and its sole child becomes the root, and the depth of the B<sup>+</sup>-tree has been decreased by 1.



**Figure.** Deletion of “Gold” from the B<sup>+</sup>-tree of Figure 11.17.

- It is worth noting that, as a result of deletion, a key value that is present in a nonleaf node of the B<sup>+</sup>-tree may not be present at any leaf of the tree. For example, in Figure, the value “Gold” has been deleted from the leaf level, but is still present in a nonleaf node.

- In general, to delete a value in a  $B^+$ -tree, we perform a lookup on the value and delete it. If the node is too small, we delete it from its parent. This deletion results in recursive application of the deletion algorithm until the root is reached, a parent remains adequately full after deletion, or redistribution is applied.
- Figure outlines the pseudocode for deletion from a  $B^+$ -tree. The procedure swap variables( $N, N'$ ) merely swaps the values of the (pointer) variables  $N$  and  $N'$ ; this swap has no effect on the tree itself. The pseudocode uses the condition “too few pointers/values.” For nonleaf nodes, this criterion means less than  $n/2$  pointers; for leaf nodes, it means less than  $(n - 1)/2$  values. The pseudocode redistributes entries by borrowing a single entry from an adjacent node. We can also redistribute entries by repartitioning entries equally between the two nodes. The pseudocode refers to deleting an entry  $(K, P)$  from a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the pointer  $P$  precedes the key value  $K$ . For nonleaf nodes,  $P$  follows the key value  $K$ .

**procedure** delete(*value K, pointer P*)

find the leaf node  $L$  that contains  $(K, P)$

delete entry( $L, K, P$ )

**procedure** delete entry(*node N, value K, pointer P*)

delete  $(K, P)$  from  $N$

**if** ( $N$  is the root **and**  $N$  has only one remaining child)

**then** make the child of  $N$  the new root of the tree and delete  $N$  **else if** ( $N$  has too few values/pointers) **then begin**

Let  $N$  be the previous or next child of *parent(N)*

Let  $K$  be the value between pointers  $N$  and  $N'$  in *parent(N)*

**if** (entries in  $N$  and  $N'$  can fit in a single node)

**then begin** /\* Coalesce nodes \*/

**if** ( $N$  is a predecessor of  $N'$ ) **then** swap variables( $N, N'$ )

**if** ( $N$  is not a leaf)

**then** append  $K$  and all pointers and values in  $N$  to  $N'$

**else** append all  $(K_i, P_i)$  pairs in  $N$  to  $N'$ ; set  $N.P_n = N'.P_n$  delete entry(*parent(N), K, N*); delete node  $N$

**end**

**else begin** /\* Redistribution: borrow an entry from  $N$  \*/ **if** ( $N$  is a predecessor of  $N'$ ) **then begin**

**if** ( $N$  is a nonleaf node) **then begin**

let  $m$  be such that  $N.P_m$  is the last pointer in  $N$  remove  $(N.K_{m-1}, N.P_m)$  from  $N$

insert  $(N.P_m, K)$  as the first pointer and value in  $N$ , by shifting other pointers and values right

replace  $K$  in *parent(N)* by  $N.K_{m-1}$

**end**

**else begin**

let  $m$  be such that  $(N.P_m, N.K_m)$  is the last pointer/value pair in  $N$

remove  $(N.P_m, N.K_m)$  from  $N$

insert  $(N.P_m, N.K_m)$  as the first pointer and value in  $N$ , by shifting other pointers and values right

replace  $K$  in *parent(N)* by  $N.K_m$

**end**

**end**

**else . . .** symmetric to the **then** case . . .

**end**

**end**

**Figure.** Deletion of entry from a  $B^+$ -tree.

**Nonunique Search Keys:**

- If a relation can have more than one record containing the same search key value (that is, two or more records can have the same values for the indexed attributes), the search key is said to be a **nonunique search key**.
- One problem with nonunique search keys is in the efficiency of record deletion. Suppose a particular search-key value occurs a large number of times, and one of the records with that search key is to be deleted. The deletion may have to search through a number of entries, potentially across multiple leaf nodes, to find the entry corresponding to the particular record being deleted.
- A simple solution to this problem, used by most database systems, is to make search keys unique by creating a composite search key containing the original search key and another attribute, which together are unique across all records.
- The extra attribute can be a record-id, which is a pointer to the record, or any other attribute whose value is unique among all records with the same search-key value. The extra attribute is called a **uniquifier** attribute.
- When a record is to be deleted, the composite search-key value is computed from the record, and then used to look up the index. Since the value is unique, the corresponding leaf-level entry can be found with a single traversal from root to leaf, with no further accesses at the leaf level. As a result, record deletion can be done efficiently.
- A search with the original search-key attribute simply ignores the value of the uniquifier attribute when comparing search-key values.
- With nonunique search keys, our  $B^+$ -tree structure stores each key value as many times as there are records containing that value. An alternative is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle nonunique search keys. This approach is more space efficient since it stores the key value only once; however, it creates several complications when  $B^+$ -trees are implemented.
- If the buckets are kept in the leaf node, extra code is needed to deal with variable-size buckets, and to deal with buckets that grow larger than the size of the leaf node. If the buckets are stored in separate blocks, an extra I/O operation may be required to fetch records. In addition to these problems, the bucket approach also has the problem of inefficiency for record deletion if a search-key value occurs a large number of times.

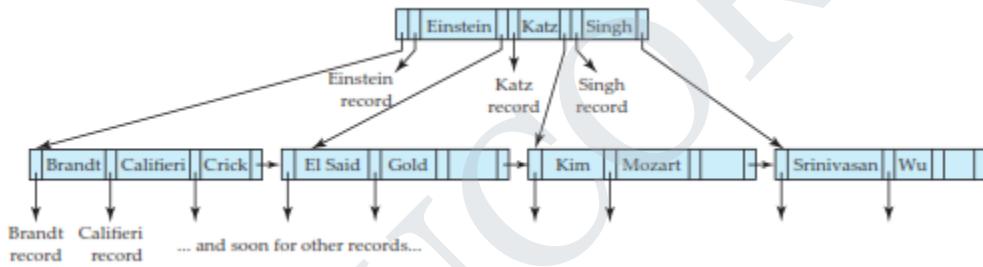
**Complexity of  $B^+$ -Tree Updates:**

- Although insertion and deletion operations on  $B^+$ -trees are complicated, they require relatively few I/O operations, which is an important benefit since I/O operations are expensive. It can be shown that the number of I/O operations needed in the worst case for an insertion is proportional to  $\log_{n/2}(N)$ , where  $n$  is the maximum number of pointers in a node, and  $N$  is the number of records in the file being indexed.
- The worst-case complexity of the deletion procedure is also proportional to  $\log_{n/2}(N)$ , provided there are no duplicate values for the search key. If there are duplicate values, deletion may have to search across multiple records with the same search-key value to find the correct entry to be deleted, which can be inefficient. However, making the search key unique by adding a uniquifier attribute, ensures the worst-case complexity of deletion is the same even if the original search key is nonunique.
- In other words, the cost of insertion and deletion operations in terms of I/O operations is proportional to the height of the  $B^+$ -tree, and is therefore low. It is the speed of operation on  $B^+$ -trees that makes them a frequently used index structure in database implementations.
- In practice, operations on  $B^+$ -trees result in fewer I/O operations than the worst-case bounds. With fanout of 100, and assuming accesses to leaf nodes are uniformly distributed, the parent of a leaf node is 100

times more likely to get accessed than the leaf node. Conversely, with the same fanout, the total number of nonleaf nodes in a  $B^+$ -tree would be just a little more than 1/100th of the number of leaf nodes. As a result, with memory sizes of several gigabytes being common today, for  $B^+$ -trees that are used frequently, even if the relation is very large it is quite likely that most of the nonleaf nodes are already in the database buffer when they are accessed. Thus, typically only one or two I/O operations are required to perform a lookup.

#### **4.7. B-TREE INDEX FILES:**

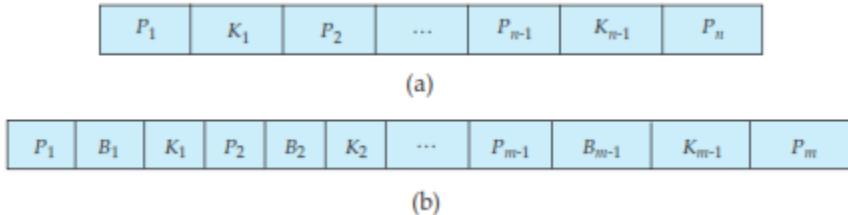
- **B-tree indices** are similar to  $B^+$ -tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. In the  $B^+$ -tree of Figure, the search keys “Califieri”, “Einstein”, “Gold”, “Mozart”, and “Srinivasan” appear in nonleaf nodes, in addition to appearing in the leaf nodes. Every search-key value appears in some leaf node; several are repeated in nonleaf nodes.
- A B-tree allows search-key values to appear only once (if they are unique), unlike a  $B^+$ -tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node. Figure shows a B-tree that represents the same search keys as the  $B^+$ -tree of Figure. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding  $B^+$ -tree index. However, since search keys that appear in nonleaf nodes appear nowhere else in the B-tree, we are forced to include an additional pointer field for each search key in a nonleaf node. These additional pointers point to either file records or buckets for the associated search key.



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

- It is worth noting that many database system manuals, articles in industry literature, and industry professionals use the term B-tree to refer to the data structure that we call the  $B^+$ -tree. In fact, it would be fair to say that in current usage, the term B-tree is assumed to be synonymous with  $B^+$ -tree. However, in this book we use the terms B-tree and  $B^+$ -tree as they were originally defined, to avoid confusion between the two data structures.
- A generalized B-tree leaf node appears in Figure (a); a nonleaf node appears in Figure (b). Leaf nodes are the same as in  $B^+$ -trees. In nonleaf nodes, the pointers  $P_i$  are the tree pointers that we used also for  $B^+$ -trees, while the pointers  $B_i$  are bucket or file-record pointers. In the generalized B-tree in the figure, there are  $n - 1$  keys in the leaf node, but there are  $m - 1$  keys in the nonleaf node. This discrepancy occurs because nonleaf nodes must include pointers  $B_i$ , thus reducing the number of search keys that can be held in these nodes. Clearly,  $m < n$ , but the exact relationship between  $m$  and  $n$  depends on the relative size of search keys and pointers.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is located. A lookup on a B<sup>+</sup>-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<sup>+</sup>-trees, implies that a B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding B<sup>+</sup>-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<sup>+</sup>-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<sup>+</sup>-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<sup>+</sup>-tree data structure, even if (as we discussed earlier) they refer to the data structure as a B-tree.

#### 4.8 . STATIC HASHING:

- One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of **hashing** allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices. .
- The term **bucket is used** to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.
- Let  $K$  denote the set of all search-key values, and let  $B$  denote the set of all bucket addresses. A **hash function**  $h$  is a function from  $K$  to  $B$ . Let  $h$  denote a hash function.
- To insert a record with search key  $K_i$ , we compute  $h(K_i)$ , which gives the address of the bucket for that record. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket.

- To perform a lookup on a search-key value  $K_i$ , we simply compute  $h(K_i)$ , then search the bucket with that address. Suppose that two search keys,  $K_5$  and  $K_7$ , have the same hash value; that is,  $h(K_5) = h(K_7)$ . If we perform a lookup on  $K_5$ , the bucket  $h(K_5)$  contains records with search-key values  $K_5$  and records with search-key values  $K_7$ . Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.
- Deletion is equally straightforward. If the search-key value of the record to be deleted is  $K_i$ , we compute  $h(K_i)$ , then search the corresponding bucket for that record, and delete the record from the bucket.
- Hashing can be used for two different purposes. In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record. In a **hash index organization** we organize the search keys, with their associated pointers, into a hash file structure.

### Hash Functions:

- The worst possible hash function maps all search-key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired. An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.
- Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key values to buckets in such a way that the distribution has these qualities:
- The distribution is *uniform*. That is, the hash function assigns each bucket the same number of search-key values from the set of *all* possible search-key values.
- The distribution is *random*. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.
- As an illustration of these principles, let us choose a hash function for the *instructor* file using the search key *dept name*. The hash function that we choose must have the desirable properties not only on the example *instructor* file that we have been using, but also on an *instructor* file of realistic size for a large university with many departments.
- Assume that we decide to have 26 buckets, and we define a hash function that maps names beginning with the  $i$ th letter of the alphabet to the  $i$ th bucket. This hash function has the virtue of simplicity, but it fails to provide a uniform distribution, since we expect more names to begin with such letters as B and R than Q and X, for example.
- Now suppose that we want a hash function on the search key *salary*. Suppose that the minimum salary is \$30,000 and the maximum salary is \$130,000, and we use a hash function that divides the values into 10 ranges, \$30,000 – \$40,000, \$40,001 – \$50,000 and so on. The distribution of search-key values is uniform (since each bucket has the same number of different *salary* values), but is not random. Records with salaries between \$60,001 and \$70,000 are far more common than are records with salaries between \$30,001 and \$40,000.

- As a result, the distribution of records is not uniform — some buckets receive more records than others do. If the function has a random distribution, even if there are such correlations in the search keys, the randomness of the distribution will make it very likely that all buckets will have roughly the same number of records, as long as each search key occurs in only a small fraction of the records. (If a single search key occurs in a large fraction of the records, the bucket containing it is likely to have more records than other buckets, regardless of the hash function used.)

The figure displays eight buckets, labeled bucket 0 through bucket 7, representing the hash organization of an *instructor* file. Each bucket is represented by a table with four columns: ID, Name, Department, and Salary.

- Bucket 0:** Empty table.
- Bucket 1:**

|       |        |       |       |
|-------|--------|-------|-------|
| 15151 | Mozart | Music | 40000 |
|       |        |       |       |
|       |        |       |       |
|       |        |       |       |
- Bucket 2:**

|       |           |         |       |
|-------|-----------|---------|-------|
| 32343 | El Said   | History | 80000 |
| 58583 | Califieri | History | 60000 |
|       |           |         |       |
|       |           |         |       |
- Bucket 3:**

|       |          |            |       |
|-------|----------|------------|-------|
| 22222 | Einstein | Physics    | 95000 |
| 33456 | Gold     | Physics    | 87000 |
| 98345 | Kim      | Elec. Eng. | 80000 |
|       |          |            |       |
- Bucket 4:**

|       |       |         |       |
|-------|-------|---------|-------|
| 12121 | Wu    | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
|       |       |         |       |
|       |       |         |       |
- Bucket 5:**

|       |       |         |       |
|-------|-------|---------|-------|
| 76766 | Crick | Biology | 72000 |
|       |       |         |       |
|       |       |         |       |
|       |       |         |       |
- Bucket 6:**

|       |            |            |       |
|-------|------------|------------|-------|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 45565 | Katz       | Comp. Sci. | 75000 |
| 83821 | Brandt     | Comp. Sci. | 92000 |
|       |            |            |       |
- Bucket 7:** Empty table.

**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:

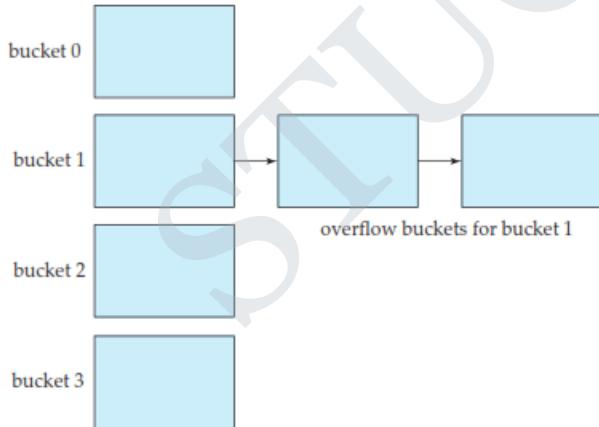
$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \dots + s[n-1]$$

- The function can be implemented efficiently by setting the hash result initially to 0, and iterating from the first to the last character of the string, at each step multiplying the hash value by 31 and then adding the next character (treated as an integer). The above expression would appear to result in a very large number, but it is actually computed with fixed-size positive integers; the result of each multiplication and addition is thus automatically computed modulo the largest possible integer value plus 1. The result of the above function modulo the number of buckets can then be used for indexing.

- Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file.

### **Handling of Bucket Overflows:**

- So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur. Bucket overflow can occur for several reasons:
- **Insufficient buckets.** The number of buckets, which we denote  $n_B$ , must be chosen such that  $n_B > n_r / f_r$ , where  $n_r$  denotes the total number of records that will be stored and  $f_r$  denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.
- **Skew.** Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:
  - 1) Multiple records may have the same search key.
  - 2) The chosen hash function may result in nonuniform distribution of search keys.
- So that the probability of bucket overflow is reduced, the number of buckets is chosen to be  $(n_r / f_r) * (1 + d)$ , where  $d$  is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.
- Despite allocation of a few more buckets than required, bucket overflow can still occur. We handle bucket overflow by using **overflow buckets**. If a record must be inserted into a bucket  $b$ , and  $b$  is already full, the system provides an overflow bucket for  $b$ , and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list. Overflow handling using such a linked list is called **overflow chaining**.



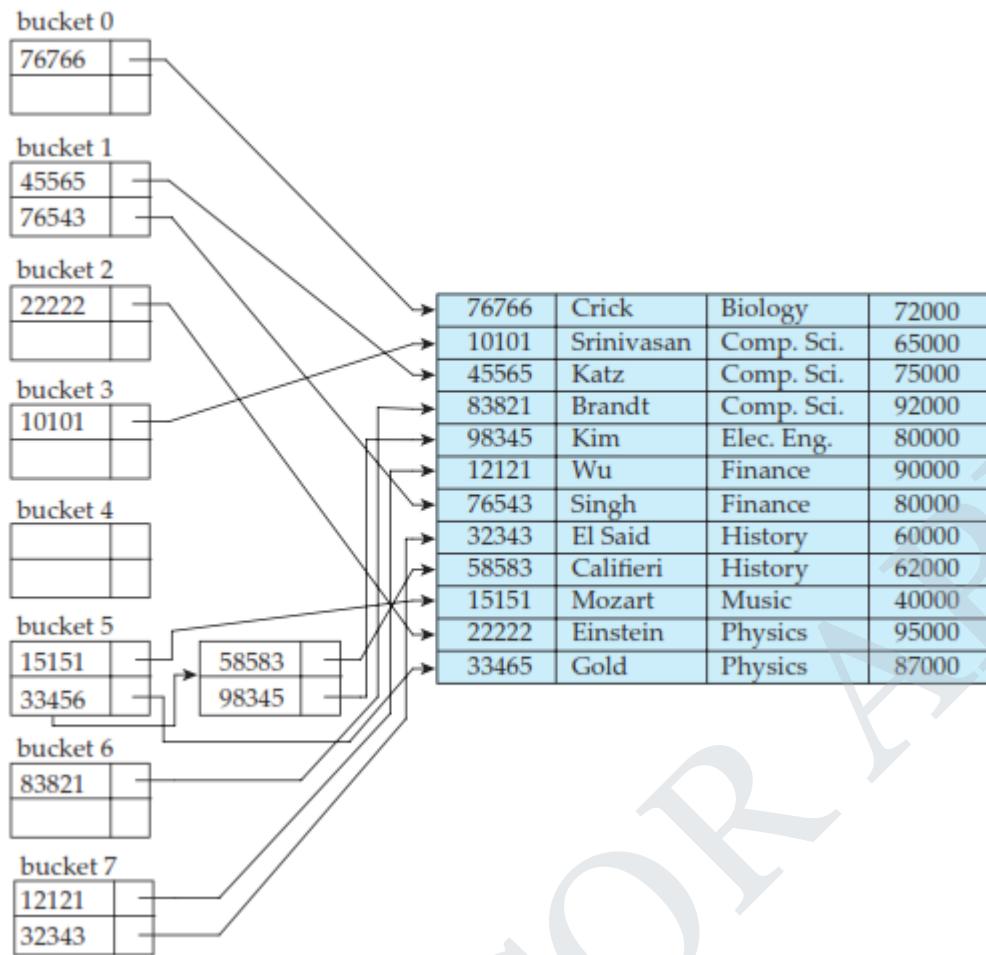
**Figure .** Overflow chaining in a hash structure.

- We must change the lookup algorithm slightly to handle overflow chaining. As before, the system uses the hash function on the search key to identify a bucket  $b$ . The system must examine all the records in bucket  $b$  to see whether they match the search key, as before. In addition, if bucket  $b$  has overflow buckets, the system must examine the records in all the overflow buckets also.

- The form of hash structure that we have just described is sometimes referred to as **closed hashing**.
- Under an alternative approach, called **open hashing**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets  $B$ . One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*. Other policies, such as computing further hash functions, are also used.
- Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.
- An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function  $h$  maps search-key values to a fixed set  $B$  of bucket addresses, we waste space if  $B$  is made large to handle future growth of the file. If  $B$  is too small, the buckets contain records of many different search-key values, and bucket overflows can occur. As the file grows, performance suffers.

### **HASH INDICES:**

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). Figure shows a secondary hash index on the *instructor* file, for the search key *ID*. The hash function in the figure computes the sum of the digits of the *ID* modulo 8.
- The hash index has eight buckets, each of size 2 (realistic indices would, of course, have much larger bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *ID* is a primary key for *instructor*, so each search key has only one associated pointer. In general, multiple pointers can be associated with each key.



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

- We use the term *hash index* to denote hash file structures as well as secondary hash indices. Strictly speaking, hash indices are only secondary index structures. A hash index is never needed as a clustering index structure, since, if a file itself is organized by hashing, there is no need for a separate hash index structure on it. However, since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a clustering hash index on it.

#### **4.9. DYNAMIC HASHING:**

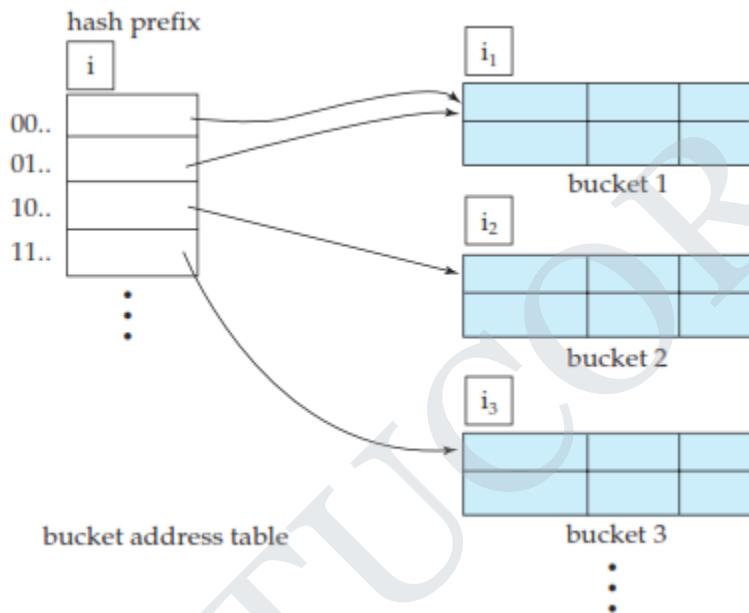
- The need to fix the set  $B$  of bucket addresses presents a serious problem with the static hashing technique. Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:
- Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
- Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
- Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating

new bucket assignments. This reorganization is a massive, time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

- **Dynamic hashing** techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database.

### Data Structure:

- Extendable hashing copes with changes in database size by splitting and join together buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since the reorganization is performed on only one bucket at a time, the resulting performance overhead is acceptably low.
- With extendable hashing, we choose a hash function  $h$  with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range — namely,  $b$ -bit binary integers. A typical value for  $b$  is 32.



**Figure.** General extendable hash structure.

- We do not create a bucket for each hash value. Indeed,  $2^{32}$  is over 4 billion, and that many buckets is unreasonable for all but the largest databases. Instead, we create buckets on demand, as records are inserted into the file. We do not use the entire  $b$  bits of the hash value initially. At any point, we use  $i$  bits, where
- $0 \leq i \leq b$ . These  $i$  bits are used as an offset into an additional table of bucket addresses. The value of  $i$  grows and shrinks with the size of the database.
- Figure shows a general extendable hash structure. The  $i$  appearing above the bucket address table in the figure indicates that  $i$  bits of the hash value  $h(K)$  are required to determine the correct bucket for  $K$ . This number will change as the file grows. Although  $i$  bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket.
- All such entries will have a common hash prefix, but the length of this prefix may be less than  $i$ . Therefore, we associate with each bucket an integer giving the length of the common hash prefix. In

Figure the integer associated with bucket  $j$  is shown as  $i_j$ . The number of bucket-address-table entries that point to bucket  $j$  is  $2(i - i_j)$

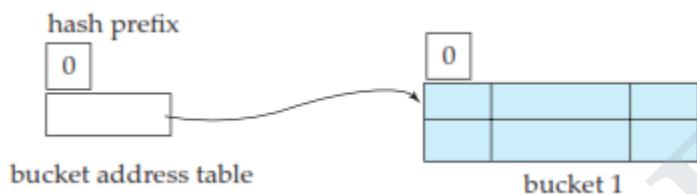
### Queries and Updates:

- To locate the bucket containing search-key value  $K_l$ , the system takes the first  $i$  high-order bits of  $h(K_l)$ , looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.
- To insert a record with search-key value  $K_l$ , the system follows the same procedure for lookup as before, ending up in some bucket—say,  $j$ . If there is room in the bucket, the system inserts the record in the bucket. If, on the other hand, the bucket is full, it must split the bucket and redistribute the current records, plus the new one. To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses.
- If  $i = i_j$ , only one entry in the bucket address table points to bucket  $j$ . Therefore, the system needs to increase the size of the bucket address table so that it can include pointers to the two buckets that result from splitting bucket  $j$ . It does so by considering an additional bit of the hash value. It increments the value of  $i$  by 1, thus doubling the size of the bucket address table. It replaces each entry by two entries, both of which contain the same pointer as the original entry. Now two entries in the bucket address table point to bucket  $j$ . The system allocates a new bucket (bucket  $z$ ), and sets the second entry to point to the new bucket. It sets  $i_j$  and  $i_z$  to  $i$ . Next, it rehashes each record in bucket  $j$  and, depending on the first  $i$  bits (remember the system has added 1 to  $i$ ), either keeps it in bucket  $j$  or allocates it to the newly created bucket.
- The system now reattempts the insertion of the new record. Usually, the attempt will succeed. However, if all the records in bucket  $j$ , as well as the new record, have the same hash-value prefix, it will be necessary to split a bucket again, since all the records in bucket  $j$  and the new record are assigned to the same bucket. If the hash function has been chosen carefully, it is unlikely that a single insertion will require that a bucket be split more than once, unless there are a large number of records with the same search key. If all the records in bucket  $j$  have the same search-key value, no amount of splitting will help. In such cases, overflow buckets are used to store the records, as in static hashing.
- If  $i > i_j$ , then more than one entry in the bucket address table points to bucket  $j$ . Thus, the system can split bucket  $j$  without increasing the size of the bucket address table. Observe that all the entries that point to bucket  $j$  correspond to hash prefixes that have the same value on the leftmost  $i_j$  bits. The system allocates a new bucket (bucket  $z$ ), and sets  $i_j$  and  $i_z$  to the value that results from adding 1 to the original  $i_j$  value. Next, the system needs to adjust the entries in the bucket address table that previously pointed to bucket  $j$ . (Note that with the new value for  $i_j$ , not all the entries correspond to hash prefixes that have the same value on the leftmost  $i_j$  bits.) The system leaves the first half of the entries as they were (pointing to bucket  $j$ ), and sets all the remaining entries to point to the newly created bucket (bucket  $z$ ). Next, as in the previous case, the system rehashes each record in bucket  $j$ , and allocates it either to bucket  $j$  or to the newly created bucket  $z$ .

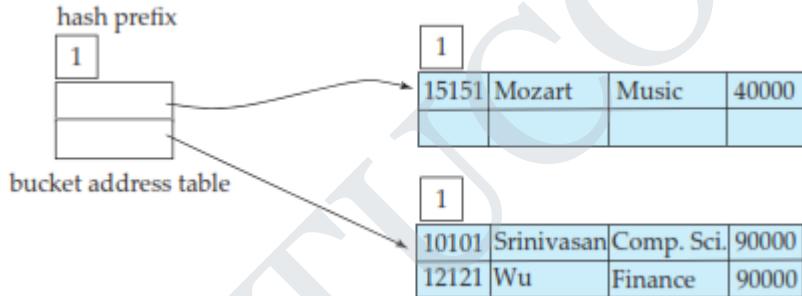
| <i>dept_name</i> | <i>h(dept_name)</i>                     |
|------------------|-----------------------------------------|
| Biology          | 0010 1101 1111 1011 0010 1100 0011 0000 |
| Comp. Sci.       | 1111 0001 0010 0100 1001 0011 0110 1101 |
| Elec. Eng.       | 0100 0011 1010 1100 1100 0110 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.



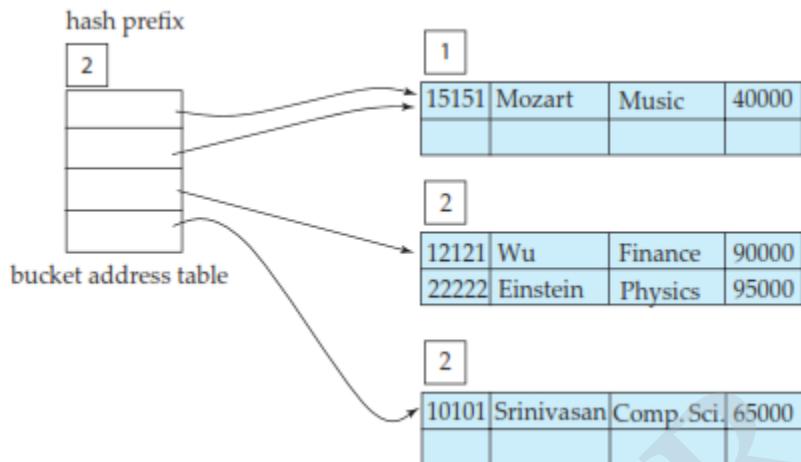
**Figure.** Initial extendable hash structure.



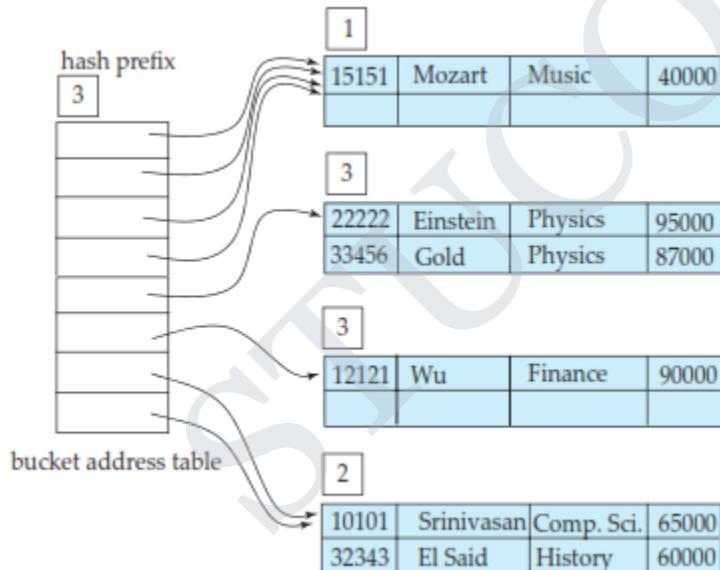
**Figure.** Hash structure after three insertions.

- 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.

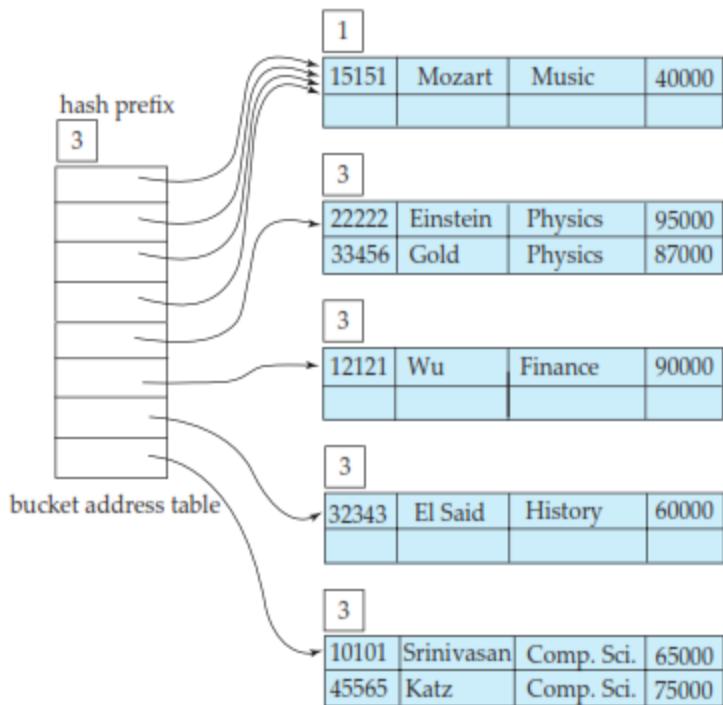
- Next, we insert (22222, Einstein, Physics, 95000). Since the first bit of  $h(\text{Physics})$  is 1, we must insert this record into the bucket pointed to by the “1” entry in the bucket address table. Once again, we find the bucket full and  $i = i_1$ . We increase the number of bits that we use from the hash to 2. This increase in the number of bits necessitates doubling the size of the bucket address table to four entries, as in Figure 11.30. Since the bucket of Figure 11.29 for hash prefix 0 was not split, the two entries of the bucket address table of 00 and 01 both point to this bucket.
- For each record in the bucket of Figure 11.29 for hash prefix 1 (the bucket being split), the system examines the first 2 bits of the hash value to determine which bucket of the new structure should hold it.



**Figure .** Hash structure after four insertions.

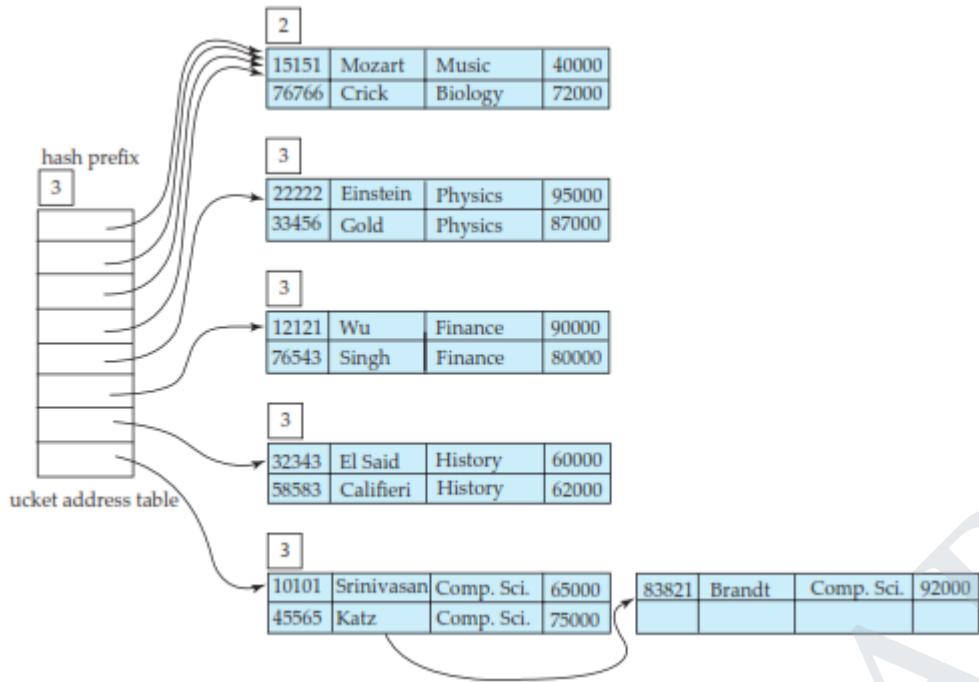
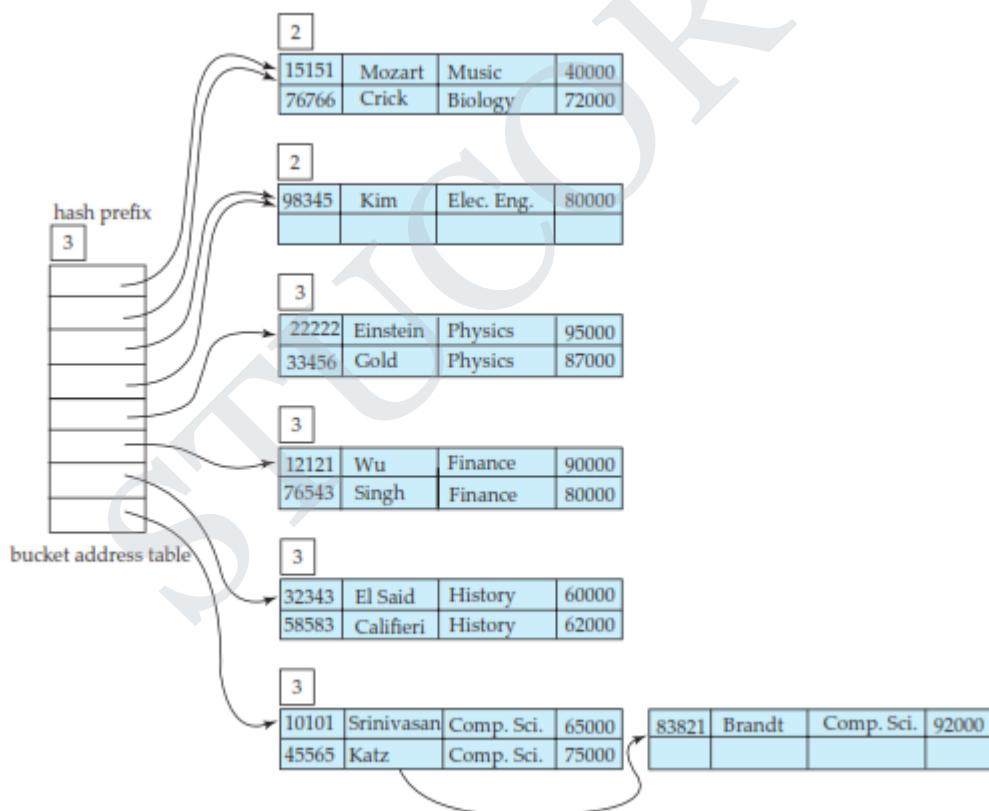


**Figure .** Hash structure after six insertions.

**Figure.** Hash structure after seven insertions.

- Next, we insert (32343, El Said, History, 60000), which goes in the same bucket as Comp. Sci. The following insertion of (33456, Gold, Physics, 87000) results in a bucket overflow, leading to an increase in the number of bits, and a doubling of the size of the bucket address table.
- The insertion of (45565, Katz, Comp. Sci., 75000) leads to another bucket over-flow; this overflow, however, can be handled without increasing the number of bits, since the bucket in question has two pointers pointing to it.

Next, we insert the records of “Califieri”, “Singh”, and “Crick” without any bucket overflow. The insertion of the third Comp. Sci. record (83821, Brandt, Comp. Sci., 92000), however, leads to another overflow. This overflow cannot be handled by increasing the number of bits, since there are three records with exactly the same hash value. Hence the system uses an overflow bucket, as in figure.

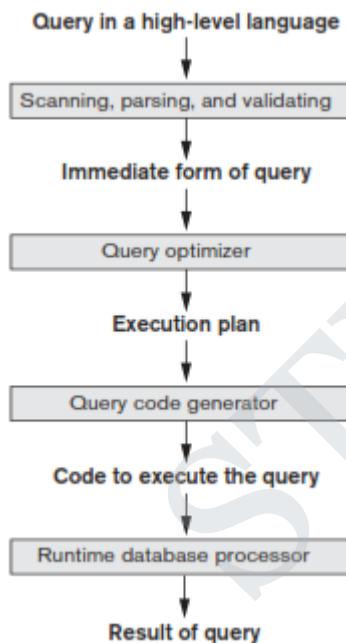
**Figure.** Hash structure after eleven insertions.**Figure.** Extendable hash structure for the *instructor* file.

**Static Hashing versus Dynamic Hashing:**

- We now examine the advantages and disadvantages of extendable hashing, compared 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. Although the hash structures that we discussed in Section 11.6 do not have this extra level of indirection, they lose their minor performance advantage as they become full.
- Thus, extendable hashing appears to be a highly attractive technique, provided 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.

#### **4.10. QUERY PROCESSING OVERVIEW:**

- A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.
- The **scanner** identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query, whereas the **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language.
- The query must also be **validated** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried.
- An internal representation of the query is then created, usually as a tree data structure called a **query tree**. It is also possible to represent the query using a graph data structure called a **query graph**.
- The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.
- Figure shows the different steps of processing a high-level query. The **query optimizer** module has the task of producing a good execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.



Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

**Figure .** Typical steps when processing a high-level query.

- The term *optimization* is actually an inaccurate name because in some cases the chosen execution plan is not the optimal (or absolute best) strategy—it is just a *reasonably efficient strategy* for executing the query.

Finding the optimal strategy is usually too time-consuming—except for the simplest of queries. In addition, trying to find the optimal query execution strategy may require detailed information on how the files are implemented and even on the contents of the files—information that may not be fully available in the DBMS catalog. Hence, *planning of a good execution strategy* may be a more accurate description than *query optimization*.

- For lower-level navigational database languages in legacy systems—such as the network DML or the hierarchical DL/1—the programmer must choose the query execution strategy while writing a database program.
- If a DBMS provides only a navigational language, there is *limited need or opportunity* for extensive query optimization by the DBMS; instead, the programmer is given the capability to choose the query execution strategy. On the other hand, a high-level query language—such as SQL for relational DBMSs (RDBMSs) or OQL for object DBMSs (ODBMSs)—is more declarative in nature because it specifies what the intended results of the query are, rather than identifying the details of *how* the result should be obtained. Query optimization is thus necessary for queries that are specified in a high-level query language.
- We will concentrate on describing query optimization in the *context of an RDBMS* because many of the techniques we describe have also been adapted for other types of database management systems, such as ODBMSs. A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or near-optimal strategy. Each DBMS typically has a number of general database access algorithms that implement relational algebra operations such as SELECT or JOIN or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query, as well as to the *particular physical database design*, can be considered by the query optimization module.

### Translating SQL Queries into Relational Algebra:

- In practice, SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into *query blocks*, which form the basic units that can be translated into the algebraic operators and optimized. A **query block** contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT—these operators must also be included in the extended algebra.

Consider the following SQL query on the EMPLOYEE relation :

```
SELECT Lname, Fname FROM EMPLOYEE WHERE Salary > (SELECT MAX (Salary) FROM EMPLOYEE WHERE Dno=5);
```

This query retrieves the names of employees (from any department in the company) who earn a salary that is greater than the *highest salary in department 5*. The query includes a nested subquery and hence would be decomposed into two blocks. The inner block is:

```
SELECT MAX (Salary) FROM EMPLOYEE WHERE Dno=5)
```

This retrieves the highest salary in department 5. The outer query block is:

```
SELECT Lname, Fname FROM EMPLOYEE WHERE Salary > c ;
```

where  $c$  represents the result returned from the inner block. The inner block could be translated into the following extended relational algebra expression:

$$\exists_{\text{MAX Salary}}(\sigma_{Dno=5}(\text{EMPLOYEE}))$$

and the outer block into the expression:

$$\pi_{\text{Lname}, \text{Fname}}(\sigma_{\text{Salary} > c}(\text{EMPLOYEE}))$$

- The *query optimizer* would then choose an execution plan for each query block. Notice that in the above example, the inner block needs to be evaluated only once to produce the maximum salary of employees in department 5, which is then used—as the constant  $c$ —by the outer block. We called this a *nested query (without correlation with the outer query)* in Section 5.1.2. It is much harder to optimize the more complex *correlated nested queries* (see Section 5.1.3), where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block.

## 4.11. ALGORITHMS FOR SELECT OPERATIONS

### Implementing the SELECT Operation

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions.

We discuss some of the algorithms for implementing SELECT in this section. We will use the following operations, specified on the relational database in Figure 3.5, to illustrate our discussion:

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

**Search Methods for Simple Selection.** A number of search algorithms are possible for selecting records from a file. These are also known as **file scans**, because they scan the records of a file to search for and retrieve records that satisfy a selection condition. If the search algorithm involves the use of an index, the index search is called an **index scan**. The following search methods (S1 through S6) are examples of some of the search algorithms that can be used to implement a select operation:

**S1—Linear search (brute force algorithm).** Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition. Since the records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

**S2—Binary search.** If the selection condition involves an equality comparison on a key attribute on which the file is **ordered**, binary search—which is more efficient than linear search—can be used. An example is OP1 if Ssn is the ordering attribute for the EMPLOYEE file.

**S3a—Using a primary index.** If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn = ‘123456789’ in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).

**S3b—Using a hash key.** If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = ‘123456789’ in OP1—use the hash key to retrieve the record. Note that this condition retrieves a single record (at most).

**S4—Using a primary index to retrieve multiple records.** If the comparison condition is  $>$ ,  $\geq$ ,  $<$ , or  $\leq$  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  $>$ ,  $\geq$ ,  $<$ , or  $\leq$ .

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,  $30000 \leq Salary \leq 35000$ . Queries involving such conditions are called **range queries**.

**Search Methods for Complex Selection.** If a condition of a SELECT operation is a **conjunctive condition**—that is, if it is made up of several simple conditions connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

**S7—Conjunctive selection using an individual index.** If an attribute involved in any **single simple condition** in the conjunctive select condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record *satisfies the remaining simple conditions* in the conjunctive select condition.

**S8—Conjunctive selection using a composite index.** If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index (or hash structure) exists on the combined fields—for example, if an index has been created on the composite key (Essn, Pno) of the WORKS\_ON file for OP5—we can use the index directly.

**S9—Conjunctive selection by intersection of record pointers.** If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers gives the record pointers that satisfy the conjunctive select condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions. In general, method S9 assumes that each of the indexes is on a *nonkey field* of the file, because if one of the conditions is an equality condition on a key field, only one record will satisfy the whole condition.

- Whenever a single condition specifies the selection—such as OP1, OP2, or OP3—the DBMS can only check whether or not an access path exists on the attribute involved in that condition. If an access path (such as index or hash key

or sorted file) exists, the method corresponding to that access path is used; otherwise, the brute force, linear search approach of method S1 can be used. Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever *more than one* of the attributes involved in the conditions have an access path. The optimizer should choose the access path that *retrieves the fewest records* in the most efficient way by estimating the different costs (see Section 19.8) and choosing the method with the least estimated cost.

- Selectivity of a Condition. When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the
- *selectivity* of each condition. The **selectivity** (*sl*) is defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus is a number between zero and one. *Zero selectivity* means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition. In general, the selectivity will not be either of these two extremes, but will be a fraction that estimates the percentage of file records that will be retrieved.
- Although exact selectivities of all conditions may not be available, **estimates of selectivities** are often kept in the DBMS catalog and are used by the optimizer. For example, for an equality condition on a key attribute of relation  $r(R)$ ,  $s = 1/|r(R)|$ , where  $|r(R)|$  is the number of tuples in relation  $r(R)$ . For an equality condition on a nonkey attribute with  $i$  distinct values,  $s$  can be estimated by  $(|r(R)|/i)/|r(R)|$  or  $1/i$ , assuming that the records are evenly or **uniformly distributed** among the distinct values.<sup>8</sup> Under this assumption,  $|r(R)|/i$  records will satisfy an equality condition on this attribute. In general, the number of records satisfying a selection condition with selectivity *sl* is estimated to be  $|r(R)| * sl$ . The smaller this estimate is, the higher the desirability of using that condition first to retrieve records. In certain cases, the actual distribution of records among the various distinct values of the attribute is kept by the DBMS in the form of a *histogram*, in order to get more accurate estimates of the number of records that satisfy a particular condition.
- Disjunctive Selection Conditions. Compared to a conjunctive selection condition, a **disjunctive condition** (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process and optimize. For example, consider OP4 :

OP4 : $\sigma_{Dno=5 \text{ OR } Salary > 30000 \text{ OR } Sex='F'}(EMPLOYEE)$

- With such a condition, little optimization can be done, because the records satisfying the disjunctive condition are the *union* of the records satisfying the individual conditions. Hence, if any *one* of the conditions does not have an access path, we are compelled to use the brute force, linear search approach. Only if an access path exists on *every* simple condition in the disjunction can we optimize the selection by retrieving the records satisfying each condition—or their record ids—and then applying the *union* operation to eliminate duplicates.
- A DBMS will have available many of the methods discussed above, and typically many additional methods. The query optimizer must choose the appropriate one for executing each SELECT operation in a query. This optimization uses formulas that estimate the costs for each available access method. The optimizer chooses the access method with the lowest estimated cost.

#### **4.12. IMPLEMENTING THE JOIN OPERATION:**

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN varieties, so we consider just these two here since we are only giving an overview of query processing and optimization. The term **join** refers to an EQUIJOIN (or NATURAL JOIN).

There are many possible ways to implement a **two-way join**, which is a join on two files. Joins involving more than two files are called **multiway joins**. The number of possible ways to execute multiway joins grows very rapidly. In this section we discuss techniques for implementing *only two-way joins*.

consider the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we discuss next are for a join operation of the form:

$R \bowtie_{A=B} S$

where  $A$  and  $B$  are the **join attributes**, which should be domain-compatible attributes of  $R$  and  $S$ , respectively. The methods we discuss can be extended to more general 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  $t[A] = s[B]$ .<sup>9</sup>

**J2—Single-loop join (using an access structure to retrieve the matching records).** If an index (or hash key) exists for one of the two join attributes—say, attribute  $B$  of file  $S$ —retrieve each record  $t$  in  $R$  (loop over file  $R$ ), and then use the access structure (such as an index or a hash key) to retrieve directly all matching records  $s$  from  $S$  that satisfy  $s[B] = t[A]$ .

**J3—Sort-merge join.** If the records of  $R$  and  $S$  are *physically sorted* (ordered) by value of the join attributes  $A$  and  $B$ , respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for  $A$  and  $B$ . If the files are not sorted, they may be sorted first by using external sorting (see Section 19.2). In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file—unless both  $A$  and  $B$  are nonkey attributes, in which case the method needs to be modified slightly. A sketch of the sort-merge join algorithm is given in Figure 19.3(a). We use  $R(i)$  to refer to the  $i$ th record in file  $R$ . A variation of the sort-merge join can be used when secondary indexes exist on both join attributes. The indexes provide the ability to access (scan) the records in order of the join attributes, but the records themselves are physically scattered all over the file blocks, so this method may be quite inefficient, as every record access may involve accessing a different disk block.

**J4—Partition-hash join.** The records of files  $R$  and  $S$  are partitioned into smaller files. The partitioning of each file is done using the same hashing function  $h$  on the join attribute  $A$  of  $R$  (for partitioning file  $R$ ) and  $B$  of  $S$  (for partitioning file  $S$ ). First, a single pass through the file with fewer records (say,  $R$ ) hashes its records to the various partitions of  $R$ ; this is called the **partitioning phase**, since the records of  $R$  are partitioned into the hash buckets. In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned, so that the partitioned subfiles of  $R$  are all kept in main memory. The collection of records with the same value of  $h(A)$  are placed in the same partition, which is a **hash bucket** in a hash table in main memory. In the second phase,

called the **probing phase**, a single pass through the other file ( $S$ ) then hashes each of its records using the same hash function  $h(B)$  to *probe* the appropriate bucket, and that record is combined with all matching records from  $R$  in that bucket. This simplified description of partition-hash join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss the general case of partition-hash join that does not require this assumption below. In practice, techniques J1 to J4 are implemented by accessing *whole disk blocks* of a file, rather than individual records. Depending on the available number of buffers in memory, the number of blocks read in from the file can be adjusted.

- How Buffer Space and Choice of Outer-Loop File Affect Performance of Nested-Loop Join. The buffer space available has an important effect on some of the join algorithms. First, let us consider the nested-loop approach (J1). Looking again at the operation OP6 above, assume that the number of buffers available in main memory for implementing the join is  $n_B = 7$  blocks (buffers). Recall that we assume that each memory buffer is the same size as one disk block.
- For illustration, assume that the DEPARTMENT file consists of  $r_D = 50$  records stored in  $b_D = 10$  disk blocks and that the EMPLOYEE file consists of  $r_E = 6000$  records stored in  $b_E = 2000$  disk blocks. It is advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop (that is,  $n_B - 2$  blocks).
- The algorithm can then read one block at a time for the inner-loop file and use its records to **probe** (that is, search) the outer-loop blocks that are currently in main memory for matching records. This reduces the total number of block accesses.
- An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the **result file**—the disk file that will contain the join result—whenever it is filled. This result buffer block then is reused to hold additional join result records.

```
(a) sort the tuples in R on attribute A ;

 sort the tuples in S on attribute B ;

 set $i \leftarrow 1, j \leftarrow 1$;

 while ($i \leq n$) and ($j \leq m$)

 do { if $R(i)[A] > S(j)[B]$

 then set $j \leftarrow j + 1$

 elseif $R(i)[A] < S(j)[B]$

 then set $i \leftarrow i + 1$

 else { (* $R(i)[A] = S(j)[B]$, so we output a matched tuple *)

 output the combined tuple $\langle R(i), S(j) \rangle$ to T ;

 (* output other tuples that match $R(i)$, if any *)

 set $l \leftarrow j + 1$;

 while ($l \leq m$) and ($R(i)[A] = S(l)[B]$)

 do { output the combined tuple $\langle R(i), S(l) \rangle$ to T ;

 set $l \leftarrow l + 1$
 }
 }
 }
 }
}

(* output other tuples that match $S(j)$, if any *)
set $k \leftarrow i + 1$;

while ($k \leq n$) and ($R(k)[A] = S(j)[B]$)

do { output the combined tuple $\langle R(k), S(j) \rangle$ to T ;

 set $k \leftarrow k + 1$
}
set $i \leftarrow k, j \leftarrow l$
```

**Figure.** Implementing JOIN by using sort-merge, where  $R$  has  $n$  tuples and  $S$  has  $m$  tuples. (a) Implementing the operation  $T \leftarrow R \bowtie_{A=B} S$ .

#### 4.13. USING HEURISTICS IN QUERY OPTIMIZATION

- Optimization techniques apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance.
- The scanner and parser of an SQL query first generate a data structure that corresponds to an *initial query representation*, which is then optimized according to heuristic rules. This leads to an *optimized query representation*, which corresponds to the query execution strategy. Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations, because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

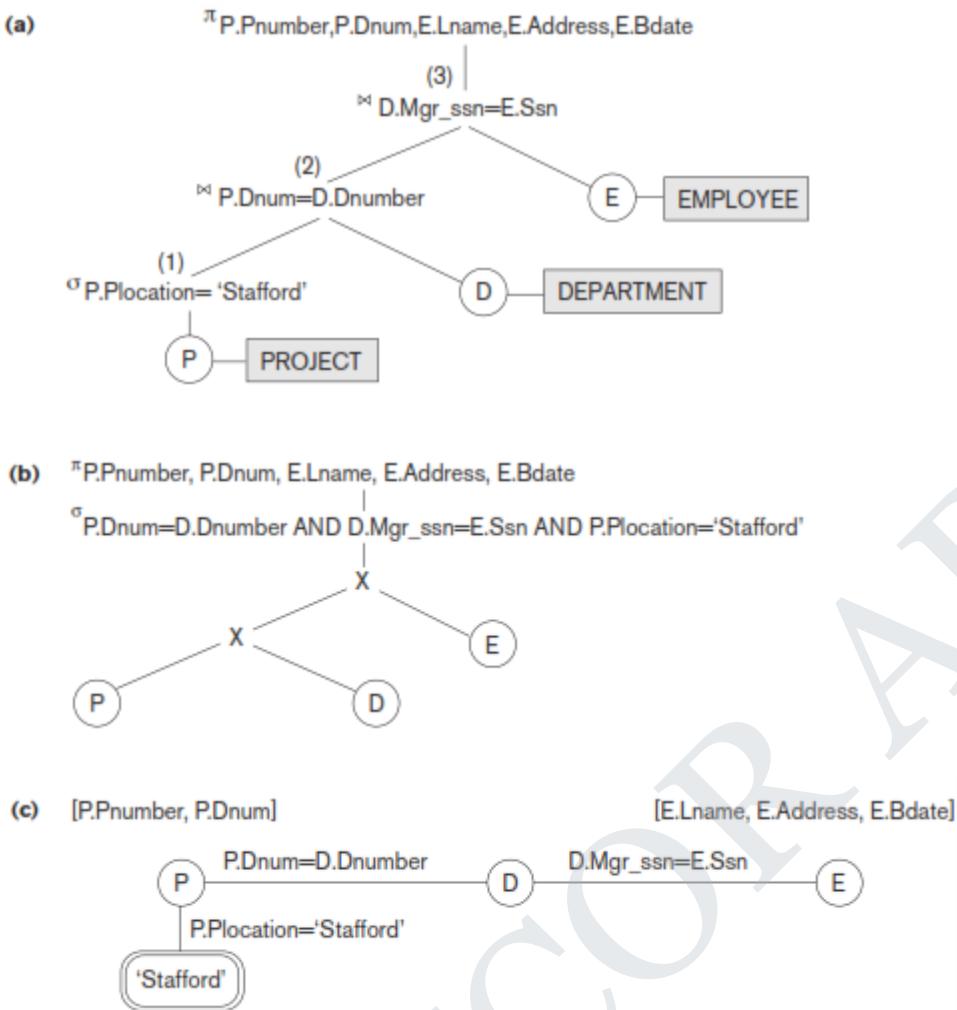
#### Notation for Query Trees and Query Graphs:

- A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query.
- Figure (a) shows a query tree for query Q2: For every project located in ‘Stafford’, retrieve the project number, the controlling department number, and the department manager’s last name, address, and birthdate. This query is specified on the COMPANY relational schema and corresponds to the following relational algebra expression:

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

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’;



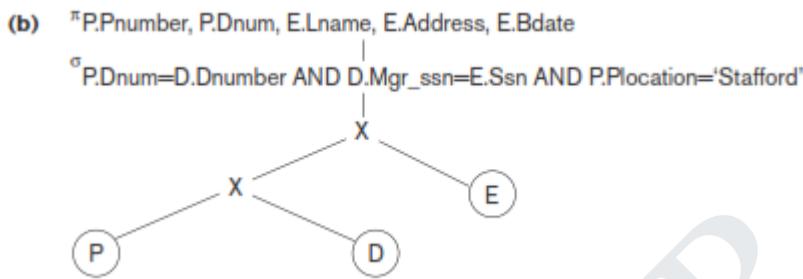
**Figure.** Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

- 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.



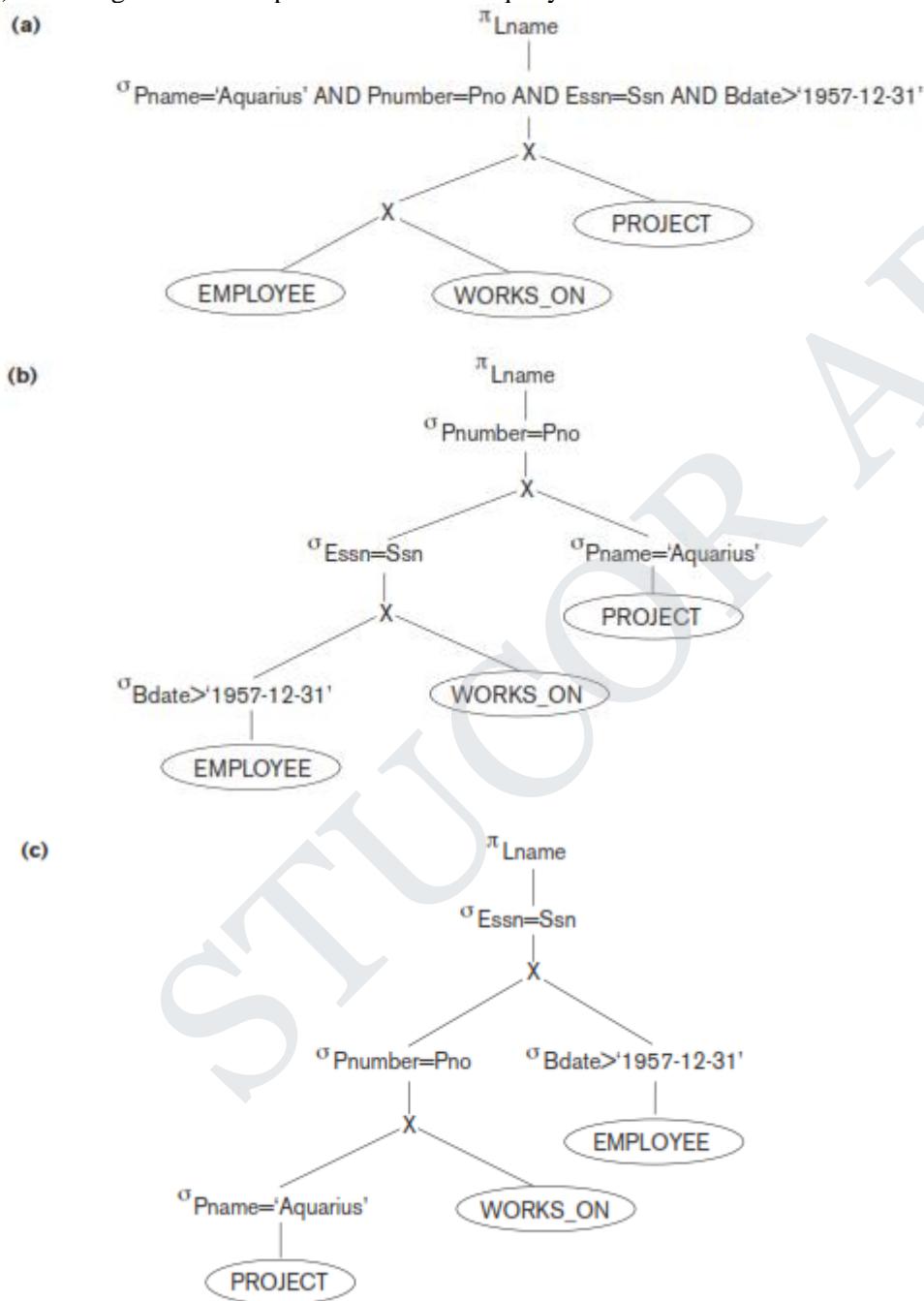
- Such a canonical query tree represents a relational algebra expression that is *very inefficient if executed directly*, because of the CARTESIAN PRODUCT ( $\times$ ) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 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:

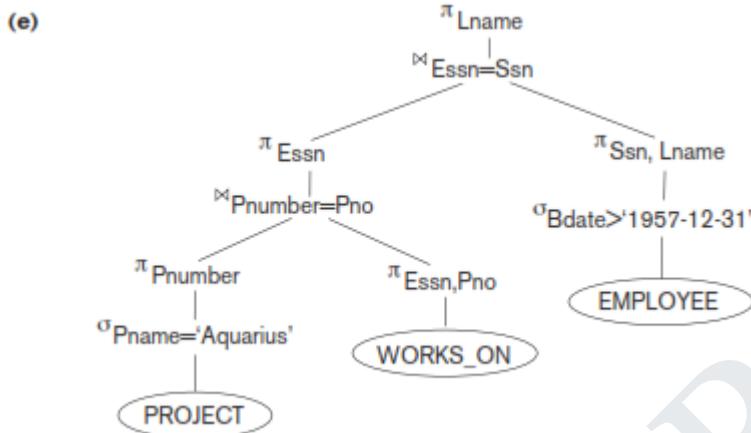
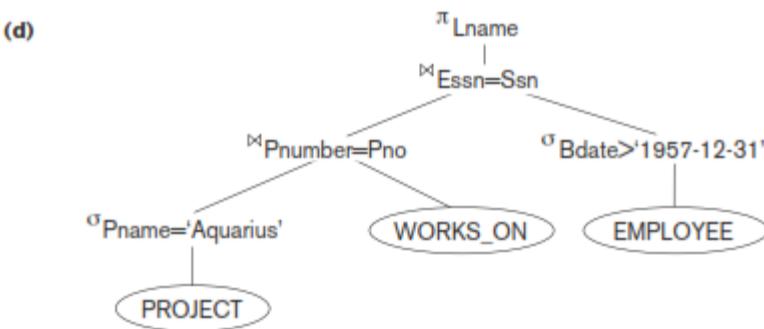
```
SELECT Lname FROM EMPLOYEE, WORKS_ON, PROJECT WHERE Pname='Aquarius'
AND Pnumber=Pno AND Essn=Ssn AND Bdate > '1957-12-31';
```

- 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.**

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





- 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 ( $\pi$ ) operations as early as possible in the query tree, as shown in Figure(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).
- As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules *preserve this equivalence*. We discuss some of these transformation rules next.
- General Transformation Rules for Relational Algebra Operations. There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a *different order* but the two relations represent the same information, we consider the relations to be equivalent.

- 1. Cascade of  $\sigma$**  A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual  $\sigma$  operations:

$$\sigma_{c_1} \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

- 2. Commutativity of  $\sigma$ .** The  $\sigma$  operation is commutative:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

- 3. Cascade of  $\pi$ .** In a cascade (sequence) of  $\pi$  operations, all but the last one can be ignored:

$$\pi_{L_{11}}(\pi_{L_{12}}(\dots(\pi_{L_{1n}}(R))\dots)) \equiv \pi_{L_{11}}(R)$$

- 4. Commuting  $\sigma$  with  $\pi$ .** If the selection condition  $c$  involves only those attributes  $A_1, \dots, A_n$  in the projection list, the two operations can be commuted:

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \dots, A_n}(R))$$

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

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

If the join condition  $c$  contains additional attributes not in  $L$ , these must be added to the projection list, and a final  $\pi$  operation is needed. For example, if attributes  $A_{n+1}, \dots, A_{n+k}$  of  $R$  and  $B_{m+1}, \dots, B_{m+p}$  of  $S$  are involved in the join condition  $c$  but are not in the projection list  $L$ , the operations commute as follows:

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

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  $\theta$  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  $\sigma$  with set operations.** The  $\sigma$  operation commutes with  $\cup$ ,  $\cap$ , and  $-$ . If  $\theta$  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  $\pi$  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  $\sigma$  that follows a  $\times$  corresponds to a join condition, convert the  $(\sigma, \times)$  sequence into a  $\bowtie$  as follows:

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

There are other possible transformations. For example, a selection or join condition  $c$  can be converted into an equivalent condition by using the following standard rules from Boolean algebra (DeMorgan's laws):

$$\text{NOT } (c_1 \text{ AND } c_2) \equiv (\text{NOT } c_1) \text{ OR } (\text{NOT } c_2)$$

$$\text{NOT } (c_1 \text{ OR } c_2) \equiv (\text{NOT } c_1) \text{ AND } (\text{NOT } c_2)$$

Additional transformations discussed in Chapters 4, 5, and 6 are not repeated here. We discuss next how transformations can be used in heuristic optimization.

**Outline of a Heuristic Algebraic Optimization Algorithm.** We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example in Figure. The steps of the algorithm are as follows:

Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.

Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from *only one table*, which means that it represents a *selection condition*, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from *two tables*, which means that it represents a *join condition*, the condition is moved to a location down the tree after the two tables are combined.

Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of *most restrictive* SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.<sup>17</sup> Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS

catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.<sup>18</sup>

Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.

Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

Identify subtrees that represent groups of operations that can be executed by a single algorithm.

In our example, Figure (b) shows the tree in Figure(a) after applying steps 1 and 2 of the algorithm; Figure(c) shows the tree after step 3; Figure 19.5(d) after step 4; and Figure(e) after step 5. In step 6 we may group together the operations in the subtree whose root is the operation  $\pi_{\text{Essn}}$  into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation  $\pi_{\text{Essn}}$ , because the first grouping means that this subtree is executed first.

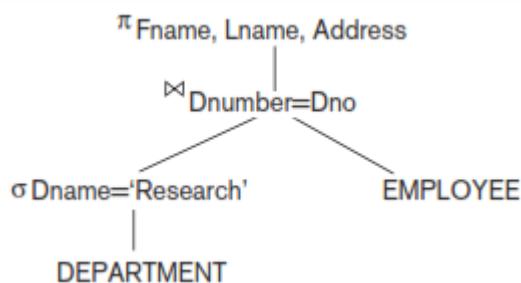
### Summary of Heuristics for Algebraic Optimization.

The main heuristic is to apply first the operations that reduce the size of intermediate results. This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible. Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

### Converting Query Trees into Query Execution Plans

An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree. As a simple example, consider query Q1 from Chapter 4, whose corresponding relational algebra expression is

$$\pi^{\text{Fname, Lname, Address}} \sigma^{\text{Dname} = \text{'Research'}} (\text{DEPARTMENT} \bowtie \text{EMPLOYEE}) \text{ on } \text{Dnumber} = \text{Dno}$$



**Figure.** A query tree for query Q1.

The query tree is shown in Figure 19.6. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), a single-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE), and a scan of the JOIN result for input to the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible.

With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is *physically materialized*). For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table. On the other hand, with **pipelined evaluation**, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence. For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

#### **4.14. USING SELECTIVITY AND COST ESTIMATES IN QUERY OPTIMIZATION:**

A query optimizer does not depend solely on heuristic rules; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the *lowest cost estimate*. For this approach to work, accurate *cost estimates* are required so that different strategies can be compared fairly and realistically. In addition, the optimizer must limit the number of execution strategies to be considered; otherwise, too much time will be spent making cost estimates for the many possible execution strategies. Hence, this approach is more suitable for **compiled queries** where the optimization is done at compile time and the resulting execution strategy code is stored and executed directly at runtime. For **interpreted queries**, where the entire process shown in

A full-scale optimization may slow down the response time. A more elaborate optimization is indicated for compiled queries, whereas a partial, less time-consuming optimization works best for interpreted queries.

This approach is generally referred to as **cost-based query optimization**. It uses traditional optimization techniques that search the *solution space* to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal (absolute best) one.

#### **Cost Components for Query Execution:**

The cost of executing a query includes the following components:

- 1) **Access cost to secondary storage.** This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as *disk I/O (input/output) cost*. The cost of searching 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 ( $b_{11}$ )** 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 equality condition on the attribute. This allows estimation of the **selection cardinality ( $s = sl * r$ )** of an attribute, which is the *average* number of records that will satisfy an equality selection condition on that attribute. For a *key attribute*,  $d = r$ ,  $sl = 1/r$  and  $s=1$ . For a *nonkey attribute*, by making an assumption that the  $d$  distinct values are uniformly distributed among the records, we estimate  $sl = (1/d)$  and so  $s = (r/d)$ .

Information such as the number of index levels is easy to maintain because it does not change very often. However, other information may change frequently; for example, the number of records  $r$  in a file changes every time a record is inserted or deleted. The query optimizer will need reasonably close but not necessarily

completely up-to-the-minute values of these parameters for use in estimating the cost of various execution strategies.

For a nonkey attribute with  $d$  distinct values, it is often the case that the records are not uniformly distributed among these values. For example, suppose that a company has 5 departments numbered 1 through 5, and 200 employees who are distributed among the departments as follows: (1, 5), (2, 25), (3, 70), (4, 40), (5, 60). In such cases, the optimizer can store a **histogram** that reflects the distribution of employee records over different departments in a table with the two attributes (Dno, Selectivity), which would contain the following values for our example: (1, 0.025), (2, 0.125), (3, 0.35), (4, 0.2), (5, 0.3). The selectivity values stored in the histogram can also be estimates if the employee table changes frequently.

In the next two sections we examine how some of these parameters are used in cost functions for a cost-based query optimizer.

### **Examples of Cost Functions for SELECT:**

We now give cost functions for the selection algorithms S1 to S8 in terms of *number of block transfers* between memory and disk. Algorithm S9 involves an intersection of record pointers after they have been retrieved by some other means, such as algorithm S6, and so the cost function will be based on the cost for S6. These cost functions are estimates that ignore computation time, storage cost, and other factors. The cost for method Si 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  $>$ ,  $\geq$ ,  $<$ , or  $\leq$  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  $>$ ,  $\geq$ ,  $<$ , or  $\leq$  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  $C_{S6b} = x + (b_{11}/2) + (r/2)$ . 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 S3a, S5, or S6a, 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,  $0 \leq js \leq 1$ . 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  $S$ ,<sup>21</sup> 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.

CS8492 DATABASE MANAGEMENT SYSTEMS

---

## UNIT V ADVANCED TOPICS

Distributed Databases: Architecture, Data Storage, Transaction Processing – Object-based Databases: Object Database Concepts, Object-Relational features, ODMG Object Model, ODL, OQL - XML Databases: XML Hierarchical Model, DTD, XML Schema, XQuery – Information Retrieval: IR Concepts, Retrieval Models, Queries in IR systems.

---

**5.1. DISTRIBUTED DATABASE:**

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

**Parallel versus Distributed Architectures:**

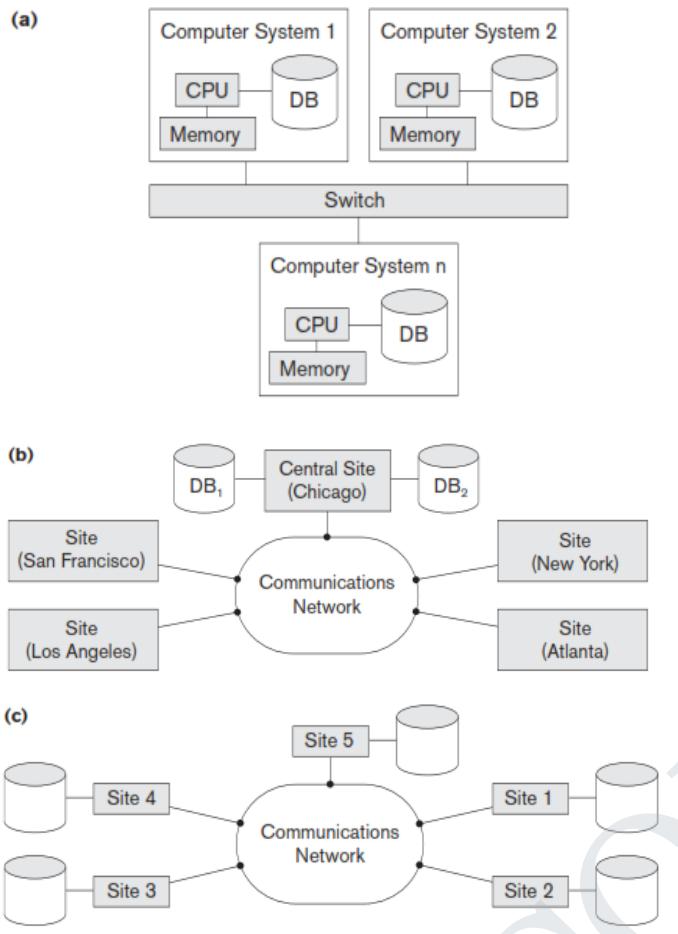
There are two main types of multiprocessor system architectures that are common-place:

- 1) **Shared memory (tightly coupled) architecture.** Multiple processors share secondary (disk) storage and also share primary memory.
- 2) **Shared disk (loosely coupled) architecture.** Multiple processors share sec-ondary (disk) storage but each has their own primary memory.

- These architectures enable processors to communicate without the overhead of exchanging messages over a network. Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared nothing architecture**.
- In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment where heterogeneity of hardware and operating system at each node is very common. Shared nothing architecture is also considered as an environment for parallel databases.

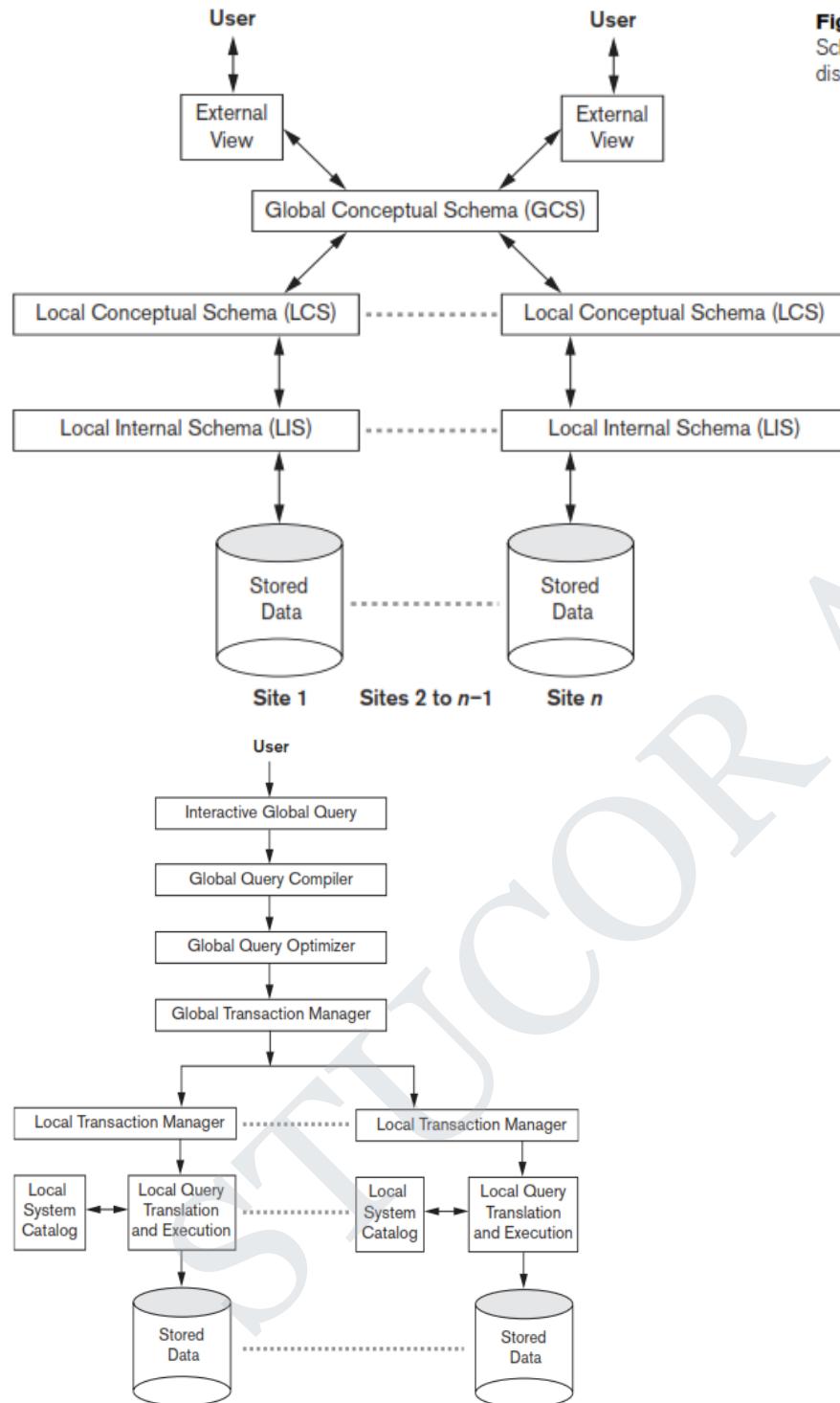
**Figure.** Some different database system architectures.

- (a) Shared nothing architecture.
- (b) A networked architecture with a centralized database at one of the sites.
- (c) A truly distributed database architecture.



## 5.2. GENERAL ARCHITECTURE OF PURE DISTRIBUTED DATABASES

- In Figure, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency.
- To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency.



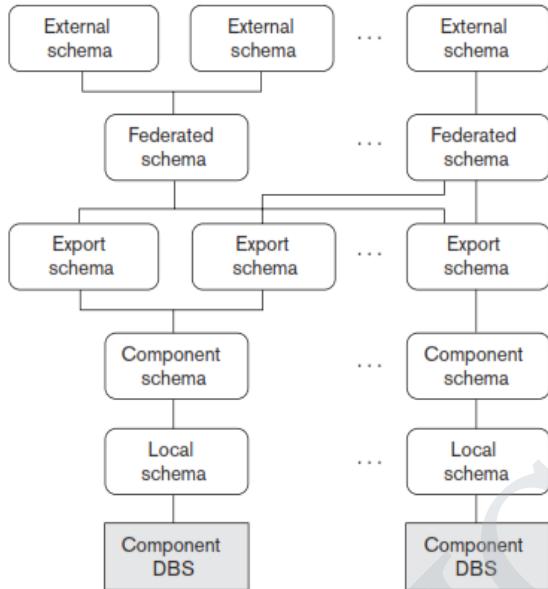
**Figure 25.4**  
Schema architecture of distributed databases.

**Figure.** Data Component architecture of distributed databases.

- Figure shows the component architecture of a DDB. It is an extension of its centralized counterpart. For the sake of simplicity, common elements are not shown here. The global query compiler references the global conceptual schema from the global system catalog to verify and impose defined constraints.

- The global query optimizer references both global and local conceptual schemas and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU, I/O, and network latencies) and estimated sizes of intermediate results. The latter is particularly important in queries involving joins. Having computed the cost for each candidate, the optimizer selects the candidate with the minimum cost for execution.
- Each local DBMS would have their local query optimizer, transaction manager, and execution engines as well as the local system catalog, which houses the local schemas. The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.

### Federated Database Schema Architecture:



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

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

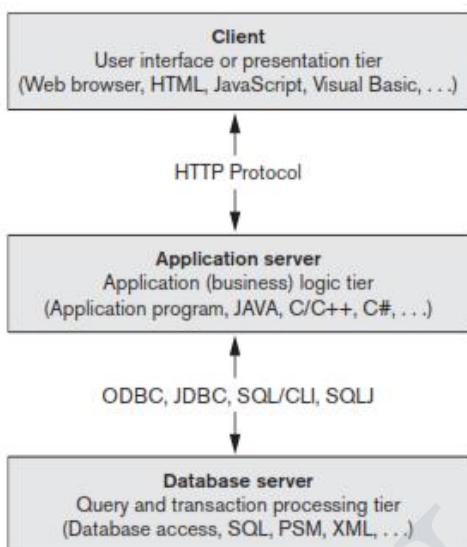
- 1) The **local schema** is the conceptual schema (full database definition) of a component database.
- 2) The **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema.
- 3) The **export schema** represents the subset of a component schema that is available to the FDBS.
- 4) The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas.
- 5) The **external schemas** define the schema for a user group or an application.

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations.

### An Overview of Three-Tier Client-Server Architecture:

In the three-tier client-server architecture, the following three layers exist:

**Presentation layer (client).** This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.



**Figure .** The three-tier client-server architecture.

**Application layer (business logic).** This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

**Database server.** This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML when transmitted between the application server and the database server.

- Exactly how to divide the DBMS functionality between the client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, where an **SQL server** is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL

is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, and SQL/CLI.

- In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between an application server and database server might proceed as follows during the processing of an SQL query:
  - The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
  - Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange, so the database server may format the query result into XML before sending it to the application server.
  - The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.
  - The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.
  - If the DDBMS has the capability to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify the sites at which the data referenced in the query or transaction resides. This property is called **distribution transparency**. Some DDBMSs do not provide distribution transparency, instead requiring that applications are aware of the details of data distribution.

### **5.3. DATA STORAGE:**

- The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.

#### **1. Data Fragmentation:**

In a DDB, decisions must be made regarding which site should be used to store which portions of the database.

The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS\_ON, and DEPENDENT. In many cases, however, a relation can be divided into smaller logical units for distribution.

We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* can be used to partition each relation by department.

### a) Horizontal Fragmentation.

A **horizontal fragment** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment are specified by a condition on one or more attributes of the relation. Often, only a single attribute is involved.

For example, we may define three horizontal fragments on the EMPLOYEE relation with the following conditions: (Dno = 5), (Dno = 4), and (Dno = 1)—each fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions (Dnum = 5), (Dnum = 4), and (Dnum = 1)—each fragment contains the PROJECT tuples controlled by a particular department.

**Horizontal fragmentation** divides a relation *horizontally* by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites in the distributed system. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. This way, related data between the primary and the secondary relations gets fragmented in the same way.

### b) Vertical Fragmentation.

Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation** divides a relation “vertically” by columns. A **vertical fragment** of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—Name, Bdate, Address, and Sex—and the second includes work-related information—Ssn, Salary, Super\_ssn, and Dno. This vertical fragmentation is not quite proper, because if the two fragments are stored separately, we cannot put the original employee tuples back together, since there is *no common attribute* between the two fragments.

It is necessary to include the primary key or some candidate key attribute in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the Ssn attribute to the personal information fragment.

Notice that each horizontal fragment on a relation  $R$  can be specified in the relational algebra by a  $\sigma_{C_i}(R)$  operation. A set of horizontal fragments whose conditions  $C_1, C_2, \dots, C_n$  include all the tuples in  $R$ —that is, every tuple in  $R$  satisfies  $(C_1 \text{ OR } C_2 \text{ OR } \dots \text{ OR } C_n)$ —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  $(C_i \text{ AND } C_j)$  for any  $i \neq j$ . Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation  $R$  from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation  $R$  can be specified by a  $\pi_{L_i}(R)$  operation in the relational algebra. A set of vertical fragments whose projection lists  $L_1, L_2, \dots, L_n$  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:

$$L_1 \cup L_2 \cup \dots \cup L_n = \text{ATTRS}(R).$$

$L_i \cap L_j = \text{PK}(R)$  for any  $i \neq j$ , where  $\text{ATTRS}(R)$  is the set of attributes of  $R$  and  $\text{PK}(R)$  is the primary key of  $R$ .

To reconstruct the relation  $R$  from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the vertical fragments (assuming no horizontal fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal fragmentation may also have been applied. The two vertical fragments of the EMPLOYEE relation

with projection lists  $L_1 = \{\text{Ssn}, \text{Name}, \text{Bdate}, \text{Address}, \text{Sex}\}$  and  $L_2 = \{\text{Ssn}, \text{Salary}, \text{Super\_ssn}, \text{Dno}\}$  constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation by the conditions ( $\text{Salary} > 50000$ ) and ( $\text{Dno} = 4$ ); they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists  $L_1 = \{\text{Name}, \text{Address}\}$  and  $L_2 = \{\text{Ssn}, \text{Name}, \text{Salary}\}$ ; these lists violate both conditions of a complete vertical fragmentation.

**Mixed (Hybrid) Fragmentation.** We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case, the original relation can be reconstructed by applying UNION and OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a **fragment** of a relation  $R$  can be specified by a SELECT-PROJECT combination of operations  $\pi_L(\sigma_C(R))$ . If

$C = \text{TRUE}$  (that is, all tuples are selected) and  $L \neq \text{ATTRS}(R)$ , we get a vertical fragment, and if  $C \neq \text{TRUE}$  and  $L = \text{ATTRS}(R)$ , we get a horizontal fragment. Finally, if

$C \neq \text{TRUE}$  and  $L \neq \text{ATTRS}(R)$ , we get a mixed fragment. Notice that a relation can itself be considered a fragment with  $C = \text{TRUE}$  and  $L = \text{ATTRS}(R)$ . In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful—although not necessary—to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to sites of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is stored. If a fragment is stored at more than one site, it is said to be **replicated**. We discuss data replication and allocation next.

## DATA REPLICATION AND ALLOCATION

- Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a **fully replicated distributed database**. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval for global queries because the results of such queries can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations drastically, since a single logical update must be performed on every copy of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there was no replication.
- The other extreme from full replication involves having **no replication**—that is, each fragment is stored at exactly one site. In this case, all fragments *must be* disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation**.
- Between these two extremes, we have a wide spectrum of **partial replication** of the data—that is, some fragments of the database may be replicated whereas others may not. The number of copies of each

fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers—such as sales forces, financial planners, and claims adjustors—carry partially replicated databases with them on laptops and PDAs and synchronize them periodically with the server database. A description of the replication of fragments is sometimes called a **replication schema**.

- Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called **data distribution** (or **data allocation**). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For example, if high availability is required, transactions can be submitted at any site, and most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

### Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database. Suppose that the company has three computer sites—one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the EMPLOYEE and PROJECT information for the employees *who work in that department* and the projects *controlled by that department*. Further, we assume that these sites mainly access the Name, Ssn, Salary, and Super\_ssn attributes of EMPLOYEE. Site 1 is used by company headquarters and accesses all employee and project information regularly, in addition to keeping track of DEPENDENT information for insurance purposes.

According to these requirements, the whole database can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, first we can horizontally fragment DEPARTMENT by its key Dnumber. Then we apply derived fragmentation to the EMPLOYEE, PROJECT, and DEPT\_LOCATIONS relations based on their foreign keys for department number—called Dno, Dnum, and Dnumber, respectively.

We can vertically fragment the resulting EMPLOYEE fragments to include only the attributes {Name, Ssn, Salary, Super\_ssn, Dno}. Figure shows the mixed fragments EMPD\_5 and EMPD\_4, which include the EMPLOYEE tuples satisfying the conditions Dno = 5 and Dno = 4, respectively. The horizontal fragments of PROJECT, DEPARTMENT, and DEPT\_LOCATIONS are similarly fragmented by department number. All these fragments—stored at sites 2 and 3—are replicated because they are also stored at headquarters—site 1.

We must now fragment the WORKS\_ON relation and decide which fragments of WORKS\_ON to store at sites 2 and 3. We are confronted with the problem that no attribute of WORKS\_ON directly indicates the department to which each tuple belongs. In fact, each tuple in WORKS\_ON relates an employee e to a project P. We could fragment WORKS\_ON based on the department D in which e works *or* based on the department D that controls P. Fragmentation becomes easy if we have a constraint stating that  $D = D$  for all WORKS\_ON tuples—that is, if employees can work only on projects controlled by the department they work for. However, there is no such constraint in our database. For example, the WORKS\_ON tuple  $\langle 333445555, 10, 10.0 \rangle$  relates an employee who works for department 5 with a project controlled by department 4. In this case, we could fragment WORKS\_ON based on the department in which the employee works (which is expressed by the condition C) and then fragment further based on the department that controls the projects that employee is working on, as shown in Figure.

In Figure, the union of fragments  $G_1$ ,  $G_2$ , and  $G_3$  gives all WORKS\_ON tuples for employees who work for department 5. Similarly, the union of fragments  $G_4$ ,  $G_5$ , and  $G_6$  gives all WORKS\_ON tuples for employees who work for department 4. On the other hand, the union of fragments  $G_1$ ,  $G_4$ , and  $G_7$  gives all WORKS\_ON tuples for projects controlled by department 5. The condition for each of the fragments  $G_1$  through  $G_9$  is shown in figure. The relations that represent M:N relationships, such as WORKS\_ON, often have several possible logical fragmentations. In our distribution in Figure 25.8, we choose to include all fragments that can be joined to either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments  $G_1$ ,  $G_2$ ,  $G_3$ ,  $G_4$ , and  $G_7$  at site 2 and the union of fragments  $G_4$ ,  $G_5$ ,  $G_6$ ,  $G_2$ , and  $G_8$  at site 3. Notice that fragments  $G_2$  and  $G_4$  are replicated at both sites. This allocation strategy permits the join between the local EMPLOYEE or PROJECT fragments at site 2 or site 3 and the local WORKS\_ON fragment to be performed completely locally. This clearly demonstrates how complex the problem of database fragmentation and allocation is for large databases. The Selected Bibliography at the end of this chapter discusses some of the work done in this area.

**Figure.** Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.

(a)

EMPD\_5

| Fname    | Minit | Lname   | Ssn       | Salary | Super_ssn | Dno |
|----------|-------|---------|-----------|--------|-----------|-----|
| John     | B     | Smith   | 123456789 | 30000  | 333445555 | 5   |
| Franklin | T     | Wong    | 333445555 | 40000  | 888665555 | 5   |
| Ramesh   | K     | Narayan | 666884444 | 38000  | 333445555 | 5   |
| Joyce    | A     | English | 453453453 | 25000  | 333445555 | 5   |

DEP\_5

| Dname    | Dnumber | Mgr_ssn   | Mgr_start_date |
|----------|---------|-----------|----------------|
| Research | 5       | 333445555 | 1988-05-22     |

DEP\_5\_LOCS

| Dnumber | Location  |
|---------|-----------|
| 5       | Bellaire  |
| 5       | Sugarland |
| 5       | Houston   |

(b)

EMPD\_4

| Fname    | Minit | Lname   | Ssn       | Salary | Super_ssn | Dno |
|----------|-------|---------|-----------|--------|-----------|-----|
| Alicia   | J     | Zelaya  | 999887777 | 25000  | 987654321 | 4   |
| Jennifer | S     | Wallace | 987654321 | 43000  | 888665555 | 4   |
| Ahmad    | V     | Jabbar  | 987987987 | 25000  | 987654321 | 4   |

DEP\_4

| Dname          | Dnumber | Mgr_ssn   | Mgr_start_date |
|----------------|---------|-----------|----------------|
| Administration | 4       | 987654321 | 1995-01-01     |

| Dnumber | Location |
|---------|----------|
| 4       | Stafford |

WORKS\_ON\_4

| Essn      | Pno | Hours |
|-----------|-----|-------|
| 333445555 | 10  | 10.0  |
| 999887777 | 30  | 30.0  |
| 999887777 | 10  | 10.0  |
| 987987987 | 10  | 35.0  |
| 987987987 | 30  | 5.0   |
| 987654321 | 30  | 20.0  |
| 987654321 | 20  | 15.0  |

Data at site 3

WORKS\_ON\_5

| Essn      | Pno | Hours |
|-----------|-----|-------|
| 123456789 | 1   | 32.5  |
| 123456789 | 2   | 7.5   |
| 666884444 | 3   | 40.0  |
| 453453453 | 1   | 20.0  |
| 453453453 | 2   | 20.0  |
| 333445555 | 2   | 10.0  |
| 333445555 | 3   | 10.0  |
| 333445555 | 10  | 10.0  |
| 333445555 | 20  | 10.0  |

PROJS\_5

| Pname     | Pnumber | Plocation | Dnum |
|-----------|---------|-----------|------|
| Product X | 1       | Bellaire  | 5    |
| Product Y | 2       | Sugarland | 5    |
| Product Z | 3       | Houston   | 5    |

Data at site 2

**Figure.** Complete and disjoint fragments of the WORKS\_ON relation.

- (a) Fragments of WORKS\_ON for employees working in department 5 ( $C=[\text{Essn} \in (\text{SELECT Ssn FROM EMPLOYEE WHERE Dno=5})]$ ).
- (b) Fragments of WORKS\_ON for employees working in department 4 ( $C=[\text{Essn} \in (\text{SELECT Ssn FROM EMPLOYEE WHERE Dno=4})]$ ).
- (c) Fragments of WORKS\_ON for employees working in department 1 ( $C=[\text{Essn} \in (\text{SELECT Ssn FROM EMPLOYEE WHERE Dno=1})]$ ).

## (a) Employees in Department 5

G1

| Essn      | Pno | Hours |
|-----------|-----|-------|
| 123456789 | 1   | 32.5  |
| 123456789 | 2   | 7.5   |
| 666884444 | 3   | 40.0  |
| 453453453 | 1   | 20.0  |
| 453453453 | 2   | 20.0  |
| 333445555 | 2   | 10.0  |
| 333445555 | 3   | 10.0  |

C1 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G2

| Essn      | Pno | Hours |
|-----------|-----|-------|
| 333445555 | 10  | 10.0  |

C2 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G3

| Essn      | Pno | Hours |
|-----------|-----|-------|
| 333445555 | 20  | 10.0  |

C3 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

## (b) Employees in Department 4

G4

C4 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G5

| Essn      | Pno | Hours |
|-----------|-----|-------|
| 999887777 | 30  | 30.0  |
| 999887777 | 10  | 10.0  |
| 987987987 | 10  | 35.0  |
| 987987987 | 30  | 5.0   |
| 987654321 | 30  | 20.0  |

C5 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G6

| Essn      | Pno | Hours |
|-----------|-----|-------|
| 987654321 | 20  | 15.0  |

C6 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

## (c) Employees in Department 1

G7

C7 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G8

C8 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G9

C9 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

**5.4. TRANSACTION MANAGEMENT IN DISTRIBUTED DATABASES:**

- The global and local transaction management software modules, along with the concurrency control and recovery manager of a DDBMS, collectively guarantee the ACID properties of transactions.
- The **global transaction manager** is supporting distributed transactions. The site where the transaction originated can temporarily assume the role of global transaction manager and coordinate the execution of database operations with transaction managers across multiple sites. Transaction managers export their functionality as an interface to the application programs.
- The operations exported by this interface are BEGIN\_TRANSACTION, READ or WRITE, END\_TRANSACTION, COMMIT\_TRANSACTION, and ROLLBACK (or ABORT).
- The manager stores bookkeeping information related to each transaction, such as a unique identifier, originating site, name, and so on. For READ operations, it returns a local copy if valid and available. For WRITE operations, it ensures that updates are visible across all sites containing copies (replicas) of the data item. For ABORT operations, the manager ensures that no effects of the transaction are reflected in any site of the distributed database. For COMMIT operations, it ensures that the effects of a write are persistently recorded on all databases containing copies of the data item. Atomic

termination (COMMIT/ ABORT) of distributed transactions is commonly implemented using the two-phase commit protocol.

- The transaction manager passes to the concurrency controller the database operation and associated information. The controller is responsible for acquisition and release of associated locks. If the transaction requires access to a locked resource, it is delayed until the lock is acquired. Once the lock is acquired, the operation is sent to the runtime processor, which handles the actual execution of the database operation. Once the operation is completed, locks are released and the transaction manager is updated with the result of the operation.

### **Two-Phase Commit Protocol**

The *two-phase commit protocol (2PC)* requires a **global recovery manager**, or **coordinator**, to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). The two-phase commit protocol has certain drawbacks that led to the development of the three-phase commit protocol.

### **Three-Phase Commit Protocol**

- 1) The biggest drawback of 2PC is that it is a blocking protocol. Failure of the coordinator blocks all participating sites, causing them to wait until the coordinator recovers. This can cause performance degradation, especially if participants are holding locks to shared resources.
- 2) Another problematic scenario is when both the coordinator and a participant that has committed crash together. In the two-phase commit protocol, a participant has no way to ensure that all participants got the commit message in the second phase. Hence once a decision to commit has been made by the coordinator in the first phase, participants will commit their transactions in the second phase independent of receipt of a global commit message by other participants. Thus, in the situation that both the coordinator and a committed participant crash together, the result of the transaction becomes uncertain or nondeterministic. Since the transaction has already been committed by one participant, it cannot be aborted on recovery by the coordinator. Also, the transaction cannot be optimistically committed on recovery since the original vote of the coordinator may have been to abort.

These problems are solved by the three-phase commit (3PC) protocol, which essentially divides the second commit phase into two subphases called **prepare-to-commit** and **commit**. The prepare-to-commit phase is used to communicate the result of the vote phase to all participants. If all participants vote yes, then the coordinator instructs them to move into the prepare-to-commit state. The commit subphase is identical to its two-phase counterpart. Now, if the coordinator crashes during this subphase, another participant can see the transaction through to completion. It can simply ask a crashed participant if it received a prepare-to-commit message. If it did not, then it safely assumes to abort. Thus the state of the protocol can be recovered irrespective of which participant crashes. Also, by limiting the time required for a transaction to commit or abort to a maximum time-out period, the protocol ensures that a transaction attempting to commit via 3PC releases locks on time-out.

The main idea is to limit the wait time for participants who have committed and are waiting for a global commit or abort from the coordinator. When a participant receives a precommit message, it knows that the rest of the participants have voted to commit. If a precommit message has not been received, then the participant will abort and release all locks.

### **Operating System Support for Transaction Management**

The following are the main benefits of operating system (OS)-supported transaction management:

Typically, DBMSs use their own semaphores to guarantee mutually exclusive access to shared resources. Since these semaphores are implemented in user space at the level of the DBMS application software, the OS has no knowledge about them. Hence if the OS deactivates a DBMS process holding a lock, other DBMS processes wanting this lock resource get queued. Such a situation can cause serious performance degradation. OS-level knowledge of semaphores can help eliminate such situations.

Specialized hardware support for locking can be exploited to reduce associated costs. This can be of great importance, since locking is one of the most common DBMS operations.

Providing a set of common transaction support operations though the kernel allows application developers to focus on adding new features to their products as opposed to reimplementing the common functionality for each application. For example, if different DDBMSs are to coexist on the same machine and they chose the two-phase commit protocol, then it is more beneficial to have this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

## **5.5. OBJECT DATABASE CONCEPTS**

### **Introduction to Object-Oriented Concepts and Features**

- An **object** typically has two components:
  - 1) state (value)
  - 2) Behavior (operations).
- It can have a *complex data structure* as well as *specific operations* defined by the programmer.
- Objects in an OOPL exist only during program execution; therefore, they are called ***transient objects***.
- An OO database can extend the existence of objects so that they are stored permanently in a database, and hence the objects become ***persistent objects*** that exist beyond program termination and can be retrieved later and shared by other programs.
- In other words, OO databases store persistent objects permanently in secondary storage, and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as indexing mechanisms to efficiently locate the objects, concurrency control to allow object sharing among concurrent programs, and recovery from failures. An OO database system will typically interface with one or more OO programming languages to provide persistent and shared object capabilities.
- The internal structure of an object in OOPLs includes the specification of **instance variables**, which hold the values that define the internal state of the object. An instance variable is similar to the concept of an *attribute* in the relational model, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some models insist that all operations a user can apply to an object must be predefined. This forces a *complete encapsulation* of objects. This rigid approach has been relaxed in most OO data models for two reasons.
  - First, database users often need to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects. Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify on the fly.
  - To encourage encapsulation, an operation is defined in two parts. The first part, called the *signature* or *interface* of the operation, specifies the operation name and arguments (or parameters). The second part, called the *method* or *body*, specifies the *implementation* of the operation, usually written in some general-

purpose programming language. Operations can be invoked by passing a *message* to an object, which includes the operation name and the parameters. The object then executes the method for that operation. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations. Hence, encapsulation provides a form of data and operation independence.

- Another key concept in OO systems is that of type and class hierarchies and *inheritance*. This permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes. This makes it easier to develop the data types of a system incrementally, and to *reuse* existing type definitions when creating new types of objects.
- One problem in early OO database systems involved representing *relationships* among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships because it is useful to identify these relationships and make them visible to users. The ODMG object database standard has recognized this need and it explicitly represents binary relationships via a pair of *inverse references*.
- Another OO concept is *operator overloading*, which refers to an operation's ability to be applied to different types of objects; in such a situation, an *operation name* may refer to several distinct *implementations*, depending on the type of object it is applied to. This feature is also called *operator polymorphism*. For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle. This may require the use of *late binding* of the operation name to the appropriate method at runtime, when the type of object to which the operation is applied becomes known.

### **Object Identity, and Objects versus Literals:**

One goal of an ODMS (Object Data Management System) is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, an ODMS provides a **unique identity** to each independent object stored in the database. This unique identity is typically implemented via a unique, system-generated **object identifier (OID)**. The value of an OID is not visible to the external user, but is used internally by the system to identify each object uniquely and to create and manage inter-object references. The OID can be assigned to program variables of the appropriate type when needed.

The main property required of an OID is that it be **immutable**; that is, the OID value of a particular object should not change. This preserves the identity of the real-world object being represented. Hence, an ODMS must have some mechanism for generating OIDs and preserving the immutability property. It is also desirable that each OID be used only once; that is, even if an object is removed from the database, its OID should not be assigned to another object. These two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed or corrected.

We can compare this with the relational model, where each relation must have a primary key attribute whose value identifies each tuple uniquely. In the relational model, if the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to ascertain that the keys represent the same real-world object (for example, the object identifier may be represented as Emp\_id in one relation and as Ssn in another).

It is inappropriate to base the OID on the physical address of the object in storage, since the physical address can change after a physical reorganization of the database. However, some early ODMSs have used the physical address as the OID to increase the efficiency of object retrieval. If the physical address of the object

changes, an *indirect pointer* can be placed at the former address, which gives the new physical location of the object. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the current physical address of the object in storage.

Some early OO data models required that everything—from a simple value to a complex object—was represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. This allows two identical basic values to have different OIDs, which can be useful in some cases.

For example, the integer value 50 can sometimes be used to mean a weight in kilograms and at other times to mean the age of a person. Then, two basic objects with distinct OIDs could be created, but both objects would represent the integer value 50. Although useful as a theoretical model, this is not very practical, since it leads to the generation of too many OIDs. Hence, most OO database systems allow for the representation of both objects and **literals** (or values).

Every object must have an immutable OID, whereas a literal value has no OID and its value just stands for itself. Thus, a literal value is typically stored within an object and *cannot be referenced* from other objects. In many systems, complex structured literal values can also be created without having a corresponding OID if needed.

### **Complex Type Structures for Objects and Literals:**

Another feature of an ODMS (and ODBs in general) is that objects and literals may have a *type structure* of *arbitrary complexity* in order to contain all of the necessary information that describes the object or literal.

In contrast, in traditional database systems, information about a complex object is often *scattered* over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation. In ODBs, a complex type may be constructed from other types by *nesting* of **type constructors**.

The three most basic constructors are

- 1) atom
- 2) struct (or tuple)
- 3) collection.

#### **1)Atom:**

One type constructor has been called the **atom** constructor. This includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages: integers, strings, floating point numbers, enumerated types, Booleans, and so on. They are called **single-valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value.

#### **2) struct:**

A second type constructor is referred to as the **struct** (or **tuple**) constructor. This can create standard structured types, such as the tuples (record types) in the basic relational model. A structured type is made up of several components, and is also sometimes referred to as a *compound* or *composite* type. More accurately, the struct constructor is not considered to be a type, but rather a **type generator**, because many different structured types can be created.

For example, two different structured types that can be created are:

```
struct Name<FirstName: string, MiddleInitial: char, LastName: string>
struct CollegeDegree<Major: string, Degree: string, Year: date>.
```

To create complex nested type structures in the object model, the *collection* type constructors are needed. Notice that the type constructors *atom* and *struct* are the only ones available in the original (basic) relational model.

### 3) Collection:

**Collection** (or *multivalued*) type constructors include the **set(T)**, **list(T)**, **bag(T)**, **array(T)**, and **dictionary(K,T)** type constructors.

These allow part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be **type generators** because many different types can be created.

For example, **set(string)**, **set(integer)**, and **set(Employee)** are three different types that can be created from the **set** type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type **set(string)** must be string values.

The **atom constructor** is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly. The **tuple constructor** can create structured values and objects of the form  $\langle a_1:i_1, a_2:i_2, \dots, a_n:i_n \rangle$ , where each  $a_j$  is an attribute name and each  $i_j$  is a value or an OID.

The other commonly used constructors are collectively referred to as collection types, but have individual differences among them.

- 1) The **set constructor** will create objects or literals that are a set of *distinct* elements  $\{i_1, i_2, \dots, i_n\}$ , all of the same type.
- 2) The **bag constructor** (sometimes called a *multiset*) is similar to a set except that the elements in a bag *need not be distinct*.
- 3) The **list constructor** will create an *ordered list*  $[i_1, i_2, \dots, i_n]$  of OIDs or values of the same type. A list is similar to a **bag** except that the elements in a list are *ordered*, and hence we can refer to the first, second, or  $j$ th element.
- 4) The **array constructor** creates a single-dimensional array of elements of the same type. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size.
- 5) The **dictionary constructor** creates a collection of two tuples  $(K, V)$ , where the value of a key  $K$  can be used to retrieve the corresponding value  $V$ .

The main characteristic of a collection type is that its objects or values will be a *collection of objects or values of the same type* that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The **tuple** type constructor is often called a **structured type**, since it corresponds to the **struct** construct in the C and C++ programming languages.

An **object definition language (ODL)** that incorporates the preceding type constructors can be used to define the object types for a particular database application.

The type constructors can be used to define the *data structures* for an *OO database schema*. Figure.1 shows how we may declare EMPLOYEE and DEPARTMENT types.

In Figure.1, the attributes that refer to other objects—such as Dept of EMPLOYEE or Projects of DEPARTMENT—are basically OIDs that serve as **references** to other objects to represent *relationships* among the objects. For example, the attribute Dept of EMPLOYEE is of type DEPARTMENT, and hence is used to refer to a specific DEPARTMENT object (the DEPARTMENT object where the employee works). The value of such an attribute would be an OID for a specific DEPARTMENT object. A binary relationship can be represented in one direction, or it can have an *inverse reference*. The latter representation makes it easy to traverse the relationship in both directions. For example, in Figure.1 the attribute Employees of DEPARTMENT has as its value a *set of references* (that is, a set of OIDs) to objects of type EMPLOYEE; these are the employees who work for the DEPARTMENT. The inverse is the reference attribute Dept of EMPLOYEE.

```
define type
EMPLOYEE
```

```

tuple (Fname: string;
 Minit: char;
 Lname: string;
 Ssn: string;
 Birth_date: DATE;
 Address: string;
 Sex: char;
 Salary: float;
 Supervisor: EMPLOYEE;
 Dept: DEPARTMENT);;

define type DATE
tuple (Year: integer;
 Month: integer;
 Day: integer;);

define type
DEPARTMENT
tuple (Dname: string;
 Dnumber: integer;
 Mgr: tuple (Manager: EMPLOYEE;
 Start_date: DATE;);
 Locations: set(string);
 Employees: set(EMPLOYEE);
 Projects: set(PROJECT););

```

Figure.1. Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

### **Encapsulation of Operations and Persistence of Objects:**

#### **Encapsulation of Operations:**

- The concepts of encapsulation is applied to database objects in ODBs by defining the **behavior** of a type of object based on the **operations** that can be externally applied to objects of that type. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations. Still other operations may perform a combination of retrieval, calculation, and update.
- In general, the **implementation** of an operation can be specified in a *general-purpose programming language* that provides flexibility and power in defining the operations.
- **The external users of the object are only made aware of the interface of the operations**, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of any hidden internal data structures of the object and the implementation of the operations that access these structures. The interface part of an operation is sometimes called the **signature**, and the operation implementation is sometimes called the **method**.
- For database applications, the requirement that all objects be completely encapsulated is too stringent. One way to relax this requirement is to divide the structure of an object into **visible** and **hidden** attributes (instance variables).
- Visible attributes can be seen by and are directly accessible to the database users and programmers via the query language.

- The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations. Most ODMSSs employ high-level query languages for accessing visible attributes.
- The term class is often used to refer to a type definition, along with the definitions of the operations for that type.**
- Figure.2 shows how the type definitions in Figure.1 can be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition.
- A method (implementation) for each operation must be defined elsewhere using a programming language. Typical operations include the **object constructor** operation (often called *new*), which is used to create a new object, and the **destructor** operation, which is used to destroy (delete) an object. A number of **object modifier** operations can also be declared to modify the states (values) of various attributes of an object.
- Additional operations can **retrieve** information about the object.

```

define class DEPARTMENT
 type tuple (Dname: string;
 Dnumber: integer;
 Mgr: tuple (Manager: EMPLOYEE;
 Start_date: DATE;);
 Locations: set (string);
 Employees: set (EMPLOYEE);
 Projects: set(PROJECT););
 operations no_of_emps: integer;
 create_dept: DEPARTMENT;
 destroy_dept: boolean;
 assign_emp(e: EMPLOYEE): boolean;
 (* adds an employee to the department *)
 remove_emp(e: EMPLOYEE): boolean;
 (* removes an employee from the department *)
end DEPARTMENT;

define class EMPLOYEE
 type tuple (Fname: string;
 Minit: char;
 Lname: string;
 Ssn: string;
 Birth_date: DATE;
 Address: string;
 Sex: char;
 Salary: float;
 Supervisor: EMPLOYEE;
 Dept: DEPARTMENT;);
 operations age: integer;
 create_emp: EMPLOYEE;
 destroy_emp: boolean;
end EMPLOYEE;

```

**Figure .2. Adding operations to the definitions of EMPLOYEE and DEPARTMENT.**

- An operation is typically applied to an object by using the **dot notation**. For example, if *d* is a reference to a DEPARTMENT object, we can invoke an operation such as *no\_of\_emps* by writing *d.no\_of\_emps*. Similarly, by writing *d.destroy\_dept*, the object referenced by *d* is destroyed (deleted).
- The only exception is the constructor operation, which returns a reference to a new DEPARTMENT object. Hence, it is customary in some OO models to have a default name for the constructor operation that is the name of the class itself, although this was not used in Figure .2. The dot notation is also used to refer to attributes of an object—for example, by writing *d.Dnumber* or *d.Mgr\_Start\_date*.

#### Specifying Object Persistence via Naming and Reachability:

An ODBS is often closely coupled with an object-oriented programming language (OOPL). The OOPL is used to specify the method (operation) implementations as well as other application code. Not all objects are meant to be stored permanently in the data-base.

- Transient objects** exist in the executing program and disappear once the program terminates.
- Persistent objects** are stored in the database and persist after program termination. The typical mechanisms for making an object persistent are *naming* and *reachability*.
- The **naming mechanism** involves giving an object a unique persistent name within a particular database. This persistent **object name** can be given via a specific statement or operation in the

program, as shown in Figure 3. The named persistent objects are used as **entry points** to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called **reachability**.

- The reachability mechanism works by making the object reachable from some other persistent object. An object  $B$  is said to be **reachable** from an object  $A$  if a sequence of references in the database lead from object  $A$  to object  $B$ .

If we first create a named persistent object  $N$ , whose state is a *set* (or possibly a *bag*) of objects of some class  $C$ , we can make objects of  $C$  persistent by *adding them* to the set, thus making them reachable from  $N$ . Hence,  $N$  is a named object that defines a **persistent collection** of objects of class  $C$ . In the object model standard,  $N$  is called the **extent** of  $C$ .

For example, we can define a class DEPARTMENT\_SET whose objects are of type set(DEPARTMENT). We can create an object of type DEPARTMENT\_SET, and give it a persistent name ALL\_DEPARTMENTS, as shown in Figure 3. Any DEPARTMENT object that is added to the set of ALL\_DEPARTMENTS by using the add\_dept operation becomes persistent by virtue of its being reachable from ALL\_DEPARTMENTS.

```
define class DEPARTMENT_SET
 type set (DEPARTMENT);
 operations add_dept(d: DEPARTMENT): boolean;
 (* adds a department to the DEPARTMENT_SET object *)
 remove_dept(d: DEPARTMENT): boolean;
 (* removes a department from the DEPARTMENT_SET object *)
 create_dept_set: DEPARTMENT_SET;
 destroy_dept_set: boolean;
 end DEPARTMENT_SET;
 ...
 persistent name ALL_DEPARTMENTS: DEPARTMENT_SET;
 (* ALL_DEPARTMENTS is a persistent named object of type DEPARTMENT_SET *)
 ...
 d:= create_dept;
 (* create a new DEPARTMENT object in the variable d *)
 ...
 b:= ALL_DEPARTMENTS.add_dept(d);
 (* make d persistent by adding it to the persistent set ALL_DEPARTMENTS *)
```

**Figure.** Creating persistent objects by naming and reachability.

- In traditional database models, such as the relational model, *all* objects are assumed to be persistent. Hence, when a table such as EMPLOYEE is created in a relational database, it represents both the *type declaration* for EMPLOYEE and a *persistent set* of *all* EMPLOYEE records (tuples).
- In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set(EMPLOYEE) or bag(EMPLOYEE) whose value is the *collection of references* (OIDs) to all persistent EMPLOYEE objects, if this is desired, as shown in Figure 3. This allows transient and persistent objects to follow the same type and class declarations of the ODL and the OOPL. In general, it is possible to define several persistent collections for the same class definition, if desired.

**TYPE HIERARCHIES AND INHERITANCE:****Simplified Model for Inheritance:**

- Inheritance allows the definition of new types based on other predefined types, leading to a **type** (or **class**) **hierarchy**.
- A type is defined by assigning it a type name, and then defining a number of attributes (instance variables) and operations (methods) for the type. In the simplified model, the attributes and operations are together called *functions*, since attributes resemble functions with zero arguments.
- A function name can be used to refer to the value of an attribute or to refer to the resulting value of an operation (method). We use the term **function** to refer to both attributes *and* operations, since they are treated similarly in a basic introduction to inheritance.
- A type in its simplest form has a **type name** and a list of visible (*public*) **functions**. When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:

TYPE\_NAME: function, function, ..., function

For example, a type that describes characteristics of a PERSON may be defined as follows:

PERSON: Name, Address, Birth\_date, Age, Ssn

- In the PERSON type, the Name, Address, Ssn, and Birth\_date functions can be implemented as stored attributes, whereas the Age function can be implemented as an operation that calculates the Age from the value of the Birth\_date attribute and the current date.
- The concept of **subtype** is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which is referred to as the **supertype**. For example, suppose that we want to define two new types EMPLOYEE and STUDENT as follows:

EMPLOYEE: Name, Address, Birth\_date, Age, Ssn, Salary, Hire\_date, Seniority

STUDENT: Name, Address, Birth\_date, Age, Ssn, Major, Gpa

- Since both STUDENT and EMPLOYEE include all the functions defined for PERSON plus some additional functions of their own, we can declare them to be **subtypes** of PERSON. Each will inherit the previously defined functions of PERSON—namely, Name, Address, Birth\_date, Age, and Ssn.
- For STUDENT, it is only necessary to define the new (local) functions Major and Gpa, which are not inherited. Presumably, Major can be defined as a stored attribute, whereas Gpa may be implemented as an operation that calculates the student's grade point average by accessing the Grade values that are internally stored (hidden) within each STUDENT object as *hidden attributes*. For EMPLOYEE, the Salary and Hire\_date functions may be stored attributes, whereas Seniority may be an operation that calculates Seniority from the value of Hire\_date.

- Therefore, we can declare EMPLOYEE and STUDENT as follows:

EMPLOYEE subtype-of PERSON: Salary, Hire\_date, Seniority

STUDENT subtype-of PERSON: Major, Gpa

- In general, a subtype includes *all* of the functions that are defined for its supertype plus some additional functions that are *specific* only to the subtype. Hence, it is possible to generate a **type hierarchy** to show the supertype/subtype relationships among all the types declared in the system.

- Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are persistently stored in the database.
- Constraints on Extents Corresponding to a Type Hierarchy. In most ODBs, an **extent** is defined to store the collection of persistent objects for each type or sub-type. In this case, the constraint is that every object in an extent that corresponds to a subtype must also be a member of the *extent* that corresponds to its supertype. Some OO database systems have a predefined system type (called the ROOT class or the OBJECT class) whose extent contains all the objects in the system.<sup>18</sup>
- Classification then proceeds by assigning objects into additional subtypes that are meaningful to the application, creating a **type hierarchy** (or **class hierarchy**) for the system. All extents for system- and user-defined classes are subsets of the extent corresponding to the class OBJECT, directly or indirectly. In the ODMG model, the user may or may not specify an extent for each class (type), depending on the application.
- An extent is a named persistent object whose value is a **persistent collection** that holds a collection of objects of the same type that are stored permanently in the database. The objects can be accessed and shared by multiple programs. It is also possible to create a **transient collection**, which exists temporarily during the execution of a program but is not kept when the program terminates. For example, a transient collection may be created in a program to hold the result of a query that selects some objects from a persistent collection and copies those objects into the transient collection. The program can then manipulate the objects in the transient collection, and once the program terminates, the transient collection ceases to exist. In general, numerous collections—transient or persistent—may contain objects of the same type.

### Polymorphism of Operations (Operator Overloading).

- Another characteristic of OO systems in general is that they provide for **polymorphism** of operations, which is also known as **operator overloading**. This concept allows the same *operator name* or *symbol* to be bound to two or more different *implementations* of the operator, depending on the type of objects to which the operator is applied.
- A simple example from programming languages can illustrate this concept. In some languages, the operator symbol “+” can mean different things when applied to operands (objects) of different types. If the operands of “+” are of type *integer*, the operation invoked is integer addition. If the operands of “+” are of type *floating point*, the operation invoked is floating point addition. If the operands of “+” are of type *set*, the operation invoked is set union. The compiler can determine which operation to execute based on the types of operands supplied.
- In OO databases, a similar situation may occur. We can use the GEOMETRY\_OBJECT example to illustrate operation polymorphism in ODB.
- In this example, the function Area is declared for all objects of type GEOMETRY\_OBJECT. However, the implementation of the method for Area may differ for each subtype of GEOMETRY\_OBJECT. One possibility is to have a general implementation for calculating the area of a generalized GEOMETRY\_OBJECT (for example, by writing a general algorithm to calculate the area of a polygon) and then to rewrite more efficient algorithms to calculate the areas of specific types of geo-metric objects, such as a circle, a rectangle, a triangle, and so on. In this case, the Area function is *overloaded* by different implementations.
- The ODMS must now select the appropriate method for the Area function based on the type of geometric object to which it is applied. In strongly typed systems, this can be done at compile time, since the object types must be known. This is termed **early** (or **static**) **binding**. However, in systems with weak typing or no typing (such as Smalltalk and LISP), the type of the object to which a function is applied may not be known until runtime. In this case, the function must check the type of object at runtime and then invoke the appropriate method. This is often referred to as **late** (or **dynamic**) **binding**.

### Multiple Inheritance and Selective Inheritance.

- **Multiple inheritance** occurs when a certain subtype  $T$  is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes. For example, we may create a subtype ENGINEERING\_MANAGER that is a subtype of both MANAGER and ENGINEER. This leads to the creation of a **type lattice** rather than a type hierarchy.
- One problem that can occur with multiple inheritance is that the supertypes from which the subtype inherits may have distinct functions of the same name, creating an ambiguity. For example, both MANAGER and ENGINEER may have a function called Salary. If the Salary function is implemented by different methods in the MANAGER and ENGINEER supertypes, an ambiguity exists as to which of the two is inherited by the subtype ENGINEERING\_MANAGER.
- It is possible, however, that both ENGINEER and MANAGER inherit Salary from the same supertype (such as EMPLOYEE) higher up in the lattice. The general rule is that if a function is inherited from some *common supertype*, then it is inherited only once. In such a case, there is no ambiguity; the problem only arises if the functions are distinct in the two supertypes.
- There are several techniques for dealing with ambiguity in multiple inheritance.
  - 1) One solution is to have the system check for ambiguity when the subtype is created, and to let the user explicitly choose which function is to be inherited at this time.
  - 2) A second solution is to use some system default.
  - 3) A third solution is to disallow multiple inheritance altogether if name ambiguity occurs, instead forcing the user to change the name of one of the functions in one of the supertypes.
- Indeed, some OO systems do not permit multiple inheritance at all. In the object database standard, multiple inheritance is allowed for operation inheritance of interfaces, but is not allowed for EXTENDS inheritance of classes.
- **Selective inheritance** occurs when a subtype inherits only some of the functions of a supertype. Other functions are not inherited. In this case, an EXCEPT clause may be used to list the functions in a supertype that are *not* to be inherited by the sub-type. The mechanism of selective inheritance is not typically provided in ODBs, but it is used more frequently in artificial intelligence applications.

### 5.6 . OBJECT-RELATIONAL FEATURES:

#### Object Database Extensions to SQL

- SQL was first specified by Chamberlin and Boyce (1974) and underwent enhancements and standardization in 1989 and 1992. The language continued its evolution with a new standard, initially called SQL3 while being developed, and later known as SQL:99 for the parts of SQL3 that were approved into the standard. Starting with the version of SQL known as SQL3, features from object databases were incorporated into the SQL standard.
- At first, these extensions were known as SQL/Object, but later they were incorporated in the main part of SQL, known as SQL/Foundation. We will use that latest standard, SQL:2008.
- The relational model with object database enhancements is sometimes referred to as the **object-relational model**. Additional revisions were made to SQL in 2003 and 2006 to add features related to XML.
- The following are some of the object database features that have been included in SQL:

1) Some **type constructors** have been added to specify complex objects. These include the *row type*, which corresponds to the tuple (or struct) constructor. An *array type* for specifying collections is also provided. Other collection type constructors, such as *set*, *list*, and *bag* constructors, were not part of the original SQL/Object specifications but were later included in the standard.

2) A mechanism for specifying **object identity** through the use of *reference type* is included.

3) **Encapsulation of operations** is provided through the mechanism of user-defined types (UDTs) that may include operations as part of their declaration. These are somewhat similar to the concept of *abstract data types* that were developed in programming languages. In addition, the concept of user-defined routines (UDRs) allows the definition of general methods (operations).

4) **Inheritance** mechanisms are provided using the keyword UNDER.

#### User-Defined Types and Complex Structures for Objects:

- To allow the creation of complex-structured objects, and to separate the declaration of a type from the creation of a table, SQL now provides **user-defined types** (UDTs). In addition, four collection types have been included to allow for multivalued types and attributes in order to specify complex-structured objects rather than just simple (flat) records. The user will create the UDTs for a particular application as part of the database schema. A **UDT** may be specified in its simplest form using the following syntax:

CREATE TYPE TYPE\_NAME AS (<component declarations>);

- Figure 4 illustrates some of the object concepts in SQL. We will explain the examples in this figure gradually as we explain the concepts. First, a UDT can be used as either the type for an attribute or as the type for a table. By using a UDT as the type for an attribute within another UDT, a complex structure for objects (tuples) in a table can be created, much like that achieved by nesting type constructors. This is similar to using the *struct* type constructor.

```
(a) CREATE TYPE STREET_ADDR_TYPE AS (
 NUMBER VARCHAR (5),
 STREET_NAME VARCHAR (25),
 APT_NO VARCHAR (5),
 SUITE_NO VARCHAR (5)
);
CREATE TYPE USA_ADDR_TYPE AS (
 STREET_ADDR STREET_ADDR_TYPE,
 CITY VARCHAR (25),
 ZIP VARCHAR (10)
);
CREATE TYPE USA_PHONE_TYPE AS (
 PHONE_TYPE VARCHAR (5),
 AREA_CODE CHAR (3),
 PHONE_NUM CHAR (7)
);
```

```
(b) CREATE TYPE PERSON_TYPE AS (
 NAME VARCHAR (35),
 SEX CHAR,
 BIRTH_DATE DATE,
 PHONES USA_PHONE_TYPE ARRAY [4],
 ADDR USA_ADDR_TYPE
)
INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD AGE() RETURNS INTEGER;
CREATE INSTANCE METHOD AGE() RETURNS INTEGER
 FOR PERSON_TYPE
BEGIN
 RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
 TODAY'S DATE AND SELF.BIRTH_DATE */
END;
);
```

```
(c) CREATE TYPE GRADE_TYPE AS (
 COURSENO CHAR (8),
 SEMESTER VARCHAR (8),
 YEAR CHAR (4),
 GRADE CHAR
);
CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (
 MAJOR_CODE CHAR (4),
 STUDENT_ID CHAR (12),
 DEGREE VARCHAR (5),
 TRANSCRIPT GRADE_TYPE ARRAY [100]
```

```
INSTANTIABLE
NOT FINAL
INSTANCE METHOD GPA() RETURNS FLOAT;
CREATE INSTANCE METHOD GPA() RETURNS FLOAT
 FOR STUDENT_TYPE
BEGIN
 RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
 SELF.TRANSCRIPT */
END;
);
CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE AS (
 JOB_CODE CHAR (4),
 SALARY FLOAT,
 SSN CHAR (11)
)
INSTANTIABLE
NOT FINAL
);
CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
 DEPT_MANAGED CHAR (20)
)
INSTANTIABLE
);
```

```

(d) CREATE TABLE PERSON OF PERSON_TYPE
 REF IS PERSON_ID SYSTEM GENERATED;
CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
 UNDER PERSON;
CREATE TABLE MANAGER OF MANAGER_TYPE
 UNDER EMPLOYEE;
CREATE TABLE STUDENT OF STUDENT_TYPE
 UNDER PERSON;

(e) CREATE TYPE COMPANY_TYPE AS (
 COMP_NAME VARCHAR (20),
 LOCATION VARCHAR (20));
CREATE TYPE EMPLOYMENT_TYPE AS (
 Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
 Company REF (COMPANY_TYPE) SCOPE (COMPANY));
CREATE TABLE COMPANY OF COMPANY_TYPE (
 REF IS COMP_ID SYSTEM GENERATED,
 PRIMARY KEY (COMP_NAME));
CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;

```

**Figure.** Illustrating some of the object features of SQL.

- (a) Using UDTs as types for attributes such as Address and Phone,
- (b) Specifying UDT for PERSON\_TYPE,
- (c) Specifying UDTs for STUDENT\_TYPE and EMPLOYEE\_TYPE as two sub-types of PERSON\_TYPE
- (d) Creating tables based on some of the UDTs, and illustrating table inheritance,
- (e) Specifying relationships using REF and SCOPE.

- For example, in Figure 4(a), the UDT STREET\_ADDR\_TYPE is used as the type for the STREET\_ADDR attribute in the UDT USA\_ADDR\_TYPE. Similarly, the UDT USA\_ADDR\_TYPE is in turn used as the type for the ADDR attribute in the UDT PERSON\_TYPE in Figure 4(b).
- If a UDT does not have any operations, as in the examples in Figure 4(a), it is possible to use the concept of ROW TYPE to directly create a structured attribute by using the keyword ROW. For example, we could use the following instead of declaring STREET\_ADDR\_TYPE as a separate type as in Figure 4(a):

```

CREATE TYPE USA_ADDR_TYPE AS (
 STREET_ADDR ROW (NUMBER VARCHAR (5),
 STREET_NAME VARCHAR (25),
 APT_NO VARCHAR (5),
 SUITE_NO VARCHAR (5)),
 CITY VARCHAR (25),
 ZIP VARCHAR (10)
);

```

- To allow for collection types in order to create complex-structured objects, four constructors are now included in SQL: ARRAY, MULTISET, LIST, and SET. These are similar to the type constructors. In the initial specification of SQL/Object, only the ARRAY type was specified, since it can be used to simulate the other types, but the three additional collection types were included in the latest version of the SQL standard. In Figure 4(b), the PHONES attribute of PERSON\_TYPE has as its type an array whose elements are of the previously defined UDT USA\_PHONE\_TYPE. This array has a maximum of four

elements, meaning that we can store up to four phone numbers per person. An array can also have no maximum number of elements if desired.

- An array type can have its elements referenced using the common notation of square brackets. For example, PHONES[1] refers to the first location value in a PHONES attribute. A built-in function CARDINALITY can return the current number of elements in an array (or any other collection type). For example, PHONES[CARDINALITY (PHONES)] refers to the last element in the array.
- The commonly used dot notation is used to refer to components of a ROW TYPE or a UDT. For example, ADDR.CITY refers to the CITY component of an ADDR attribute.

### **Object Identifiers Using Reference Types:**

- Unique system-generated object identifiers can be created via the **reference type** in the latest version of SQL. For example, in Figure 4(b), the phrase:

REF IS SYSTEM GENERATED

indicates that whenever a new PERSON\_TYPE object is created, the system will assign it a unique system-generated identifier. It is also possible not to have a system-generated object identifier and use the traditional keys of the basic relational model if desired.

- In general, the user can specify that system-generated object identifiers for the individual rows in a table should be created. By using the syntax:

REF IS <OID\_ATTRIBUTE> <VALUE\_GENERATION\_METHOD> ;

the user declares that the attribute named <OID\_ATTRIBUTE> will be used to identify individual tuples in the table. The options for <VALUE\_GENERATION\_METHOD> are SYSTEM GENERATED or DERIVED. In the former case, the system will automatically generate a unique identifier for each tuple. In the latter case, the traditional method of using the user-provided primary key value to identify tuples is applied.

### **Creating Tables Based on the UDTs:**

- For each UDT that is specified to be instantiable via the phrase INSTANTIABLE (see Figure 11.4(b)), one or more tables may be created. This is illustrated in Figure 4(d), where we create a table PERSON based on the PERSON\_TYPE UDT. Notice that the UDTs in Figure 4(a) are non instantiable, and hence can only be used as types for attributes, but not as a basis for table creation. In Figure 4(b), the attribute PERSON\_ID will hold the system-generated object identifier whenever a new PERSON record (object) is created and inserted in the table.

### **Encapsulation of Operations:**

- In SQL, a **user-defined type** can have its own behavioral specification by specifying methods (or operations) in addition to the attributes. The general form of a UDT specification with methods is as follows:

```
CREATE TYPE <TYPE-NAME> (
<LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES> <DECLARATION OF
FUNCTIONS (METHODS)>);
```

- For example, in Figure 4(b), we declared a method Age() that calculates the age of an individual object of type PERSON\_TYPE.
- The code for implementing the method still has to be written. We can refer to the method implementation by specifying the file that contains the code for the method, or we can write the actual code within the type declaration itself..

- SQL provides certain built-in functions for user-defined types. For a UDT called TYPE\_T, the **constructor function** TYPE\_T() returns a new object of that type. In the new UDT object, every attribute is initialized to its default value. An **observer function** A is implicitly created for each attribute A to read its value. Hence, A(X) or X.A returns the value of attribute A of TYPE\_T if X is of type TYPE\_T. A **mutator function** for updating an attribute sets the value of the attribute to a new value. SQL allows these functions to be blocked from public use; an EXECUTE privilege is needed to have access to these functions.
- In general, a UDT can have a number of user-defined functions associated with it. The syntax is

INSTANCE METHOD <NAME> (<ARGUMENT\_LIST>) RETURNS <RETURN\_TYPE>;

- Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended PSM language of SQL (see Chapter 13). External functions are written in a host language, with only their signature (interface) appearing in the UDT definition. An external function definition can be declared as follows:

DECLARE EXTERNAL <FUNCTION\_NAME> <SIGNATURE> LANGUAGE  
<LANGUAGE\_NAME>;

Attributes and functions in UDTs are divided into three categories:

PUBLIC (visible at the UDT interface)

PRIVATE (not visible at the UDT interface)

PROTECTED (visible only to subtypes)

It is also possible to define virtual attributes as part of UDTs, which are computed and updated using functions.

### **Specifying Inheritance and Overloading of Functions:**

- SQL has rules for dealing with **type inheritance** (specified via the UNDER keyword). In general, both attributes and instance methods (operations) are inherited. The phrase NOT FINAL must be included in a UDT if subtypes are allowed to be created under that UDT (see Figure 4(a) and (b), where PERSON\_TYPE, STUDENT\_TYPE, and EMPLOYEE\_TYPE are declared to be NOT FINAL). Associated with type inheritance are the rules for overloading of function implementations and for resolution of function names. These inheritance rules can be summarized as follows:
  - 1) All attributes are inherited.
  - 2) The order of supertypes in the UNDER clause determines the inheritance hierarchy.
  - 3) An instance of a subtype can be used in every context in which a supertype instance is used.
  - 4) A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
  - 5) When a function is called, the best match is selected based on the types of all arguments.
  - 6) For dynamic linking, the runtime types of parameters is considered.
- Consider the following examples to illustrate type inheritance, which are illustrated in Figure 4(c). Suppose that we want to create two subtypes of PERSON\_TYPE: EMPLOYEE\_TYPE and STUDENT\_TYPE. In addition, we also create a subtype MANAGER\_TYPE that inherits all the attributes (and methods) of EMPLOYEE\_TYPE but has an additional attribute DEPT\_MANAGED. These subtypes are shown in Figure 4(c).
- In general, we specify the local attributes and any additional specific methods for the subtype, which inherits the attributes and operations of its supertype.

- Another facility in SQL is **table inheritance** via the supertable/subtable facility. This is also specified using the keyword UNDER (see Figure 4(d)). Here, a new record that is inserted into a subtable, say the MANAGER table, is also inserted into its supertables EMPLOYEE and PERSON. Notice that when a record is inserted in MANAGER, we must provide values for all its inherited attributes. INSERT, DELETE, and UPDATE operations are appropriately propagated.

### **Specifying Relationships via Reference:**

- A component attribute of one tuple may be a **reference** (specified using the keyword REF) to a tuple of another (or possibly the same) table. An example is shown in Figure 4(e).
- The keyword SCOPE specifies the name of the table whose tuples can be referenced by the reference attribute. Notice that this is similar to a foreign key, except that the system-generated value is used rather than the primary key value.
- SQL uses a **dot notation** to build **path expressions** that refer to the component attributes of tuples and row types. However, for an attribute whose type is REF, the dereferencing symbol  $\rightarrow$  is used. For example, the query below retrieves employees working in the company named ‘ABCXYZ’ by querying the EMPLOYMENT table:

```
SELECT E.Employee->NAME
FROM EMPLOYMENT AS E
WHERE E.Company->COMP_NAME = 'ABCXYZ';
```

- In SQL,  $\rightarrow$  is used for **dereferencing** and has the same meaning assigned to it in the C programming language. Thus, if  $r$  is a reference to a tuple and  $a$  is a component attribute in that tuple, then  $r \rightarrow a$  is the value of attribute  $a$  in that tuple.
- If several relations of the same type exist, SQL provides the SCOPE keyword by which a reference attribute may be made to point to a tuple within a specific table of that type.

## **5.8. ODMG(Object Data Management Group) OBJECT MODEL :**

One of the reasons for the success of commercial relational DBMSs is the SQL standard. The lack of a standard for ODMSs for several years may have caused some potential users to shy away from converting to this new technology. Subsequently, a consortium of ODMS vendors and users, called ODMG (Object Data Management Group), proposed a standard that is known as the ODMG-93 or ODMG 1.0 standard. This was revised into ODMG 2.0, and later to ODMG 3.0. The standard is made up of several parts, including the **object model**, the **object definition language (ODL)**, the **object query language (OQL)**, and the **bindings** to object-oriented programming languages.

### **Overview of the Object Model of ODMG**

The **ODMG object model** is the data model upon which the object definition language (ODL) and object query language (OQL) are based. It is meant to provide a standard data model for object databases, just as SQL describes a standard data model for relational databases. It also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts.

#### **Objects and Literals :**

Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an object identifier and a **state** (or current value), whereas a literal has a value (state) but *no object identifier*. In either case, the value can have a complex structure. The object state can change

over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, but it does not change.

An **object** has five aspects: **identifier, name, lifetime, structure, and creation**.

The **object identifier** is a unique system-wide identifier (or Object\_id). Every object must have an object identifier.

Some objects may optionally be given a unique **name** within a particular ODMS—this name can be used to locate the object, and the system should return the object given that name. Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object type—such as *extents*—will have a name. These names are used as **entry points** to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also have unique names, and it is possible to give *more than one* name to an object. All names within a particular ODMS must be unique.

The **lifetime** of an object specifies whether it is a *persistent object* (that is, a database object) or *transient object* (that is, an object in an executing program that disappears after the program terminates). Lifetimes are independent of types—that is, some objects of a particular type may be transient whereas others may be persistent.

The **structure** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is *atomic* or not.

An **atomic object** refers to a single object that follows a user-defined type, such as Employee or Department. If an object is not atomic, then it will be composed of other objects. For example, a *collection object* is not an atomic object, since its state will be a collection of other objects.

In the ODMG model, an atomic object is any *individual user-defined object*. All values of the basic built-in data types are considered to be *literals*.

Object **creation** refers to the manner in which an object can be created. This is typically accomplished via an operation *new* for a special Object\_Factory interface. We shall describe this in more detail later in this section.

In the object model, a **literal** is a value that *does not have* an object identifier. However, the value may have a simple or complex structure. There are three types of literals: atomic, structured, and collection.

**Atomic literals** correspond to the values of basic data types and are pre-defined. The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords long, short, unsigned long, and unsigned short in ODL), regular and double precision floating point numbers (float, double), Boolean values (boolean), single characters (char), character strings (string), and enumeration types (enum), among others.

**Structured literals** correspond roughly to values that are constructed using the tuple constructor. The built-in structured literals include Date, Interval, Time, and Timestamp.

Additional user-defined structured literals can be defined as needed by each application. User-defined structures are created using the STRUCT keyword in ODL, as in the C and C++ programming languages.

```
(a) interface Object {
 ...
 boolean same_as(in object other_object);
 object copy();
 void delete();
};
```

```

(b) Class Date : Object {
 enum Weekday
 { Sunday, Monday, Tuesday, Wednesday,
 Thursday, Friday, Saturday };
 enum Month
 { January, February, March, April, May, June,
 July, August, September, October, November,
 December };
 unsigned short year();
 unsigned short month();
 unsigned short day();
 ...
 boolean is_equal(in Date other_date);
 boolean is_greater(in Date other_date);
 ...
};

Class Time : Object {
 ...
 unsigned short hour();
 unsigned short minute();
 unsigned short second();
 unsigned short millisecond();
 ...
 boolean is_equal(in Time a_time);
 boolean is_greater(in Time a_time);
 ...
 Time add_interval(in Interval an_interval);
 Time subtract_interval(in Interval an_interval);
 Interval subtract_time(in Time other_time); };

class Timestamp : Object {
 ...
 unsigned short year();
 unsigned short month();
 unsigned short day();
 unsigned short hour();
 unsigned short minute();
 unsigned short second();
 unsigned short millisecond();
 ...
 Timestamp plus(in Interval an_interval);
};

```

```

Timestamp minus(in Interval an_interval);
boolean is_equal(in Timestamp a_timestamp);
boolean is_greater(in Timestamp a_timestamp);
... };
class Interval : Object {
unsigned short day();
unsigned short hour();
unsigned short minute();
unsigned short second();
unsigned short millisecond();
...
Interval plus(in Interval an_interval);
Interval minus(in Interval an_interval);
Interval product(in long a_value);
Interval quotient(in long a_value);
boolean is_equal(in interval an_interval);
boolean is_greater(in interval an_interval);
... };

(c) interface Collection : Object {
...
exception ElementNotFound{ Object element; };
unsigned long cardinality();
boolean is_empty();

...
boolean contains_element(in Object element);
void insert_element(in Object element);
void remove_element(in Object element)
 raises(ElementNotFound);
iterator create_iterator(in boolean stable);
...
};

interface Iterator {
exception NoMoreElements();
...
boolean at_end();
void reset();
Object get_element() raises(NoMoreElements);
void next_position() raises(NoMoreElements);
...
};

interface set : Collection {
set create_union(in set other_set);

...
boolean is_subset_of(in set other_set);
... };

interface bag : Collection {
unsigned long occurrences_of(in Object element); bag create_union(in Bag other_bag);
... };

```

```

...
interface list : Collection {
 exception Invalid_Index{unsigned_long index; };
 void remove_element_at(in unsigned long index)
 raises(InvalidIndex);
 Object retrieve_element_at(in unsigned long index)
 raises(InvalidIndex);
 void replace_element_at(in Object element, in unsigned long index)
 raises(InvalidIndex);
 void insert_element_after(in Object element, in unsigned long index)
 raises(InvalidIndex);
 ...
 void insert_element_first(in Object element);
 ...
 void remove_first_element() raises(ElementNotFound);
 ...
 Object retrieve_first_element() raises(ElementNotFound);
 ...
 list
 void concat(in list other_list);
 append(in list other_list);
};

interface array : Collection {
 exception Invalid_Index{unsigned_long index; };
 exception Invalid_Size{unsigned_long size; };
 void remove_element_at(in unsigned long index)
 raises(InvalidIndex);
 Object retrieve_element_at(in unsigned long index)
 raises(InvalidIndex);
 void replace_element_at(in unsigned long index, in Object element)
 raises(InvalidIndex);
 void resize(in unsigned long new_size)
 raises(InvalidSize);
};

struct association { Object key; Object value; };
interface dictionary : Collection {
 exception DuplicateName{string key; };
 exception KeyNotFound{Object key; };
 void bind(in Object key, in Object value)
 raises(DuplicateName);
 void unbind(in Object key) raises(KeyNotFound);
 Object lookup(in Object key) raises(KeyNotFound);
 boolean contains_key(in Object key);
};

```

**Collection literals** specify a literal value that is a collection of objects or values but the collection itself does not have an `Object_id`. The collections in the object model can be defined by the *type generators* `set<T>`, `bag<T>`, `list<T>`, and `array<T>`, where  $T$  is the type of objects or values in the collection.<sup>28</sup> Another collection type is `dictionary<K, V>`, which is a collection of associations  $\langle K, V \rangle$ , where each  $K$  is a key (a unique search value) associated with a value  $V$ ; this can be used to create an index on a collection of values  $V$ .

Figure gives a simplified view of the basic types and type generators of the object model. The notation of ODMG uses three concepts: interface, literal, and class. Following the ODMG terminology, we use the word

**behavior** to refer to *operations* and **state** to refer to *properties* (attributes and relationships). An interface specifies only behavior of an object type and is typically **noninstantiable** (that is, no objects are created corresponding to an interface). Although an interface may have state properties (attributes and relationships) as part of its specifications, these *cannot* be inherited from the interface. Hence, an interface serves to define operations that can be *inherited* by other interfaces, as well as by classes that define the user-defined objects for a particular application. A class specifies both state (attributes) and behavior (operations) of an object type, and is **instantiable**. Hence, database and application objects are typically created based on the user-specified class declarations that form a database schema. Finally, a **literal** declaration specifies state but no behavior. Thus, a literal instance holds a simple or complex structured value but has neither an object identifier nor encapsulated operations.

Figure is a simplified version of the object model. In the object model, all objects inherit the basic interface operations of Object, shown in Figure (a); these include operations such as copy (creates a new copy of the object), delete (deletes the object), and same\_as (compares the object's identity to another object). In general, operations are applied to objects using the **dot notation**. For example, given an object  $O$ , to compare it with another object  $P$ , we write

$$O.\text{same\_as}(P)$$

The result returned by this operation is Boolean and would be true if the identity of  $P$  is the same as that of  $O$ , and false otherwise. Similarly, to create a copy  $P$  of object  $O$ , we write

$$P = O.\text{copy}()$$

An alternative to the dot notation is the **arrow notation**:  $O \rightarrow \text{same\_as}(P)$  or  $O \rightarrow \text{copy}()$ .

### Inheritance in the Object Model of ODMG

In the ODMG object model, two types of inheritance relationships exist: behavior-only inheritance and state plus behavior inheritance. **Behavior inheritance** is also known as *ISA* or *interface inheritance*, and is specified by the colon (:) notation. Hence, in the ODMG object model, behavior inheritance requires the supertype to be an interface, whereas the subtype could be either a class or another interface.

The other inheritance relationship, called **EXTENDS inheritance**, is specified by the keyword extends. It is used to inherit both state and behavior strictly among classes, so both the supertype and the subtype must be classes. Multiple inheritance via extends is not permitted. However, multiple inheritance is allowed for behavior inheritance via the colon (:) notation. Hence, an interface may inherit behavior from several other interfaces. A class may also inherit behavior from several interfaces via colon (:) notation, in addition to inheriting behavior and state from *at most one* other class via extends. In Section 11.3.4 we will give examples of how these two inheritance relationships—“:” and extends—may be used.

### Built-in Interfaces and Classes in the Object Model

Figure shows the built-in interfaces and classes of the object model. All interfaces, such as Collection, Date, and Time, inherit the basic Object interface. In the object model, there is a distinction between collection objects, whose state contains multiple objects or literals, versus atomic (and structured) objects, whose state is an individual object or literal.

**Collection objects** inherit the basic Collection interface shown in Figure 11.5(c), which shows the operations for all collection objects. Given a collection object  $O$ , the  $O.\text{cardinality}()$  operation returns the number of elements in the collection. The operation  $O.\text{is\_empty}()$  returns true if the collection  $O$  is empty, and returns false otherwise. The operations  $O.\text{insert\_element}(E)$  and  $O.\text{remove\_element}(E)$  insert or remove an element  $E$  from the collection  $O$ . Finally, the operation  $O.\text{contains\_element}(E)$  returns true if the collection  $O$  includes element  $E$ , and returns false otherwise. The operation  $I = O.\text{create\_iterator}()$  creates an **iterator object**  $I$  for the collection object  $O$ , which can iterate over each element in the collection. The interface for

iterator objects is also shown in Figure 11.5(c). The  $I.\text{reset}()$  operation sets the iterator at the first element in a collection (for an unordered collection, this would be some arbitrary element), and  $I.\text{next\_position}()$  sets the iterator to the next element. The  $I.\text{get\_element}()$  retrieves the **current element**, which is the element at which the iterator is currently positioned.

The ODMG object model uses **exceptions** for reporting errors or particular conditions. For example, the  $\text{ElementNotFound}$  exception in the Collection interface would be raised by the  $O.\text{remove\_element}(E)$  operation if  $E$  is not an element in the collection  $O$ . The  $\text{NoMoreElements}$  exception in the iterator interface would be raised by the  $I.\text{next\_position}()$  operation if the iterator is currently positioned at the last element in the collection, and hence no more elements exist for the iterator to point to.

Collection objects are further specialized into set, list, bag, array, and dictionary, which inherit the operations of the Collection interface. A  $\text{set}\langle T \rangle$  type generator can be used to create objects such that the value of object  $O$  is a *set whose elements are of type T*. The Set interface includes the additional operation  $P = O.\text{create\_union}(S)$  (see Figure 11.5(c)), which returns a new object  $P$  of type  $\text{set}\langle T \rangle$  that is the union of the two sets  $O$  and  $S$ . Other operations similar to  $\text{create\_union}$  (not shown in Figure 11.5(c)) are  $\text{create\_intersection}(S)$  and  $\text{create\_difference}(S)$ . Operations for set comparison include the  $O.\text{is\_subset\_of}(S)$  operation, which returns true if the set object  $O$  is a subset of some other set object  $S$ , and returns false otherwise. Similar operations (not shown in Figure 11.5(c)) are  $\text{is\_proper\_subset\_of}(S)$ ,  $\text{is\_superset\_of}(S)$ , and  $\text{is\_proper\_superset\_of}(S)$ . The  $\text{bag}\langle T \rangle$  type generator allows duplicate elements in the collection and also inherits the Collection interface. It has three operations— $\text{create\_union}(b)$ ,  $\text{create\_intersection}(b)$ , and  $\text{create\_difference}(b)$ —that all return a new object of type  $\text{bag}\langle T \rangle$ .

A  $\text{list}\langle T \rangle$  object type inherits the Collection operations and can be used to create collections where the order of the elements is important. The value of each such object  $O$  is an *ordered list whose elements are of type T*. Hence, we can refer to the first, last, and  $i$ th element in the list. Also, when we add an element to the list, we must specify the position in the list where the element is inserted. Some of the list operations are shown in Figure 11.5(c). If  $O$  is an object of type  $\text{list}\langle T \rangle$ , the operation  $O.\text{insert\_element\_first}(E)$  inserts the element  $E$  before the first element in the list  $O$ , so that  $E$  becomes the first element in the list. A similar operation (not shown) is  $O.\text{insert\_element\_last}(E)$ . The operation  $O.\text{insert\_element\_after}(E, I)$  in Figure 11.5(c) inserts the element  $E$  after the  $i$ th element in the list  $O$  and will raise the exception  $\text{InvalidIndex}$  if no  $i$ th element exists in  $O$ . A similar operation (not shown) is  $O.\text{insert\_element\_before}(E, I)$ . To remove elements from the list, the operations are  $E$

$O.\text{remove\_first\_element}()$ ,  $E = O.\text{remove\_last\_element}()$ , and  $E = O.\text{remove\_element\_at}(I)$ ; these operations remove the indicated element from the list and return the

element as the operation's result. Other operations retrieve an element without removing it from the list. These are  $E = O.\text{retrieve\_first\_element}()$ ,  $E = O.\text{retrieve\_last\_element}()$ , and  $E = O.\text{retrieve\_element\_at}(I)$ . Also, two operations to manipulate lists are defined. They are  $P = O.\text{concat}(I)$ , which creates a new list  $P$  that is the concatenation of lists  $O$  and  $I$  (the elements in list  $O$  followed by those in list  $I$ ), and  $O.\text{append}(I)$ , which appends the elements of list  $I$  to the end of list  $O$  (without creating a new list object).

The  $\text{array}\langle T \rangle$  object type also inherits the Collection operations, and is similar to list. Specific operations for an array object  $O$  are  $O.\text{replace\_element\_at}(I, E)$ , which replaces the array element at position  $I$  with element  $E$ ;  $E = O.\text{remove\_element\_at}(I)$ , which retrieves the  $i$ th element and replaces it with a NULL value; and

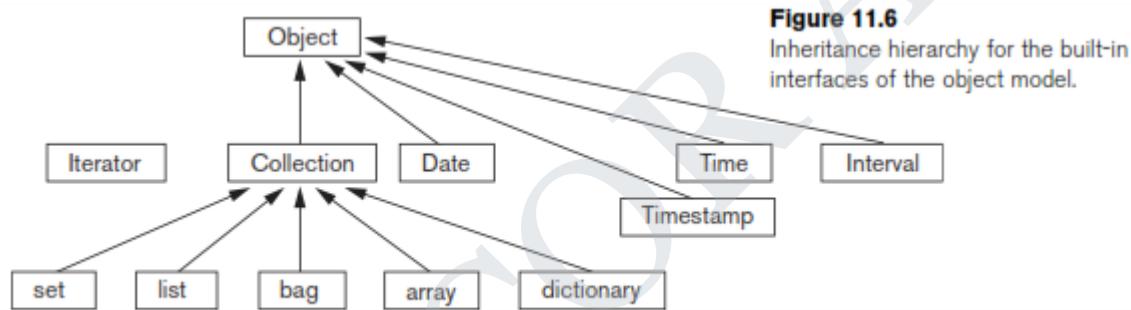
$E = O.\text{retrieve\_element\_at}(I)$ , which simply retrieves the  $i$ th element of the array. Any of these operations can raise the exception  $\text{InvalidIndex}$  if  $I$  is greater than the array's size. The operation  $O.\text{resize}(N)$  changes the number of array elements to  $N$ .

The last type of collection objects are of type  $\text{dictionary}\langle K, V \rangle$ . This allows the creation of a collection of association pairs  $\langle K, V \rangle$ , where all  $K$  (key) values are unique. This allows for associative retrieval of a particular pair given its key value (similar to an index). If  $O$  is a collection object of type  $\text{dictionary}\langle K, V \rangle$ ,

then  $O.\text{bind}(K, V)$  binds value  $V$  to the key  $K$  as an association  $\langle K, V \rangle$  in the collection, whereas  $O.\text{unbind}(K)$  removes the association with key  $K$  from  $O$ , and  $V = O.\text{lookup}(K)$  returns the value  $V$  associated with key  $K$  in  $O$ . The latter two operations can raise the exception `KeyNotFound`. Finally,  $O.\text{contains\_key}(K)$  returns true if key  $K$  exists in  $O$ , and returns false otherwise.

Figure 11.6 is a diagram that illustrates the inheritance hierarchy of the built-in constructs of the object model. Operations are inherited from the supertype to the sub-type. The collection interfaces described above are *not directly instantiable*; that is, one cannot directly create objects based on these interfaces. Rather, the interfaces can be used to generate user-defined collection types—of type set, bag, list, array, or dictionary—for a particular database application. If an attribute or class has a collection type, say a set, then it will inherit the operations of the set interface. For example, in a UNIVERSITY database application, the user can specify a type for  $\text{set} < \text{STUDENT} >$ , whose state would be sets of STUDENT objects. The programmer can then use the operations for  $\text{set} < T >$  to manipulate an instance of type  $\text{set} < \text{STUDENT} >$ . Creating application classes is typically done by utilizing the object definition language ODL.

It is important to note that all objects in a particular collection *must be of the same type*. Hence, although the keyword `any` appears in the specifications of collection interfaces in Figure (c), this does not mean that objects of any type can be inter-mixed within the same collection. Rather, it means that any type can be used when specifying the type of elements for a particular collection (including other collection types!).



**Figure 11.6**

Inheritance hierarchy for the built-in interfaces of the object model.

### Atomic (User-Defined) Objects

The previous section described the built-in collection types of the object model. Now we discuss how object types for *atomic objects* can be constructed. These are specified using the keyword `class` in ODL. In the object model, any user-defined object that is not a collection object is called an **atomic object**.

For example, in a UNIVERSITY database application, the user can specify an object type (class) for STUDENT objects. Most such objects will be **structured objects**; for example, a STUDENT object will have a complex structure, with many attributes, relationships, and operations, but it is still considered atomic because it is not a collection. Such a user-defined atomic object type is defined as a class by specifying its **properties** and **operations**. The properties define the state of the object and are further distinguished into **attributes** and **relationships**. In this subsection, we elaborate on the three types of components—attributes, relationships, and operations—that a user-defined object type for atomic (structured) objects can include. We illustrate our discussion with the two classes EMPLOYEE and DEPARTMENT shown in Figure 11.7.

An attribute is a property that describes some aspect of an object. Attributes have values (which are typically literals having a simple or complex structure) that are stored within the object. However, attribute values can also be `Object_ids` of other objects. Attribute values can even be specified via methods that are used to calculate the attribute value. In Figure the attributes for EMPLOYEE are Name, Ssn, Birth\_date, Sex, and Age, and those for DEPARTMENT are Dname, Dnumber, Mgr, Locations, and Projs. The Mgr and Projs attributes of DEPARTMENT have complex structure and are defined via struct, which corresponds to the *tuple constructor*. Hence, the value of Mgr in each DEPARTMENT object will have two components: Manager,

whose value is an Object\_id that references the EMPLOYEE object that manages the DEPARTMENT, and Start\_date, whose value is a date. The locations attribute of DEPARTMENT is defined via the set constructor, since each DEPARTMENT object can have a set of locations.

A relationship is a property that specifies that two objects in the database are related. In the object model of ODMG, only binary relationships are explicitly represented, and each binary relationship is represented by a *pair of inverse references* specified via the keyword relationship. In Figure, one relationship exists that relates each EMPLOYEE to the DEPARTMENT in which he or she works—the Works\_for relationship of EMPLOYEE. In the inverse direction, each DEPARTMENT is related to the set of EMPLOYEES that work in the DEPARTMENT—the Has\_emps relationship of DEPARTMENT. The keyword inverse specifies that these two properties define a single conceptual relationship in inverse directions.

```

class EMPLOYEE
{
 extent ALL_EMPLOYEES
 key Ssn
 {
 attribute string Name;
 attribute string Ssn;
 attribute date Birth_date;
 attribute enum Gender{M, F} Sex;
 attribute short Age;
 relationship DEPARTMENT Works_for
 inverse DEPARTMENT::Has_emps;
 void reassign_emp(in string New_dname)
 raises(dname_not_valid);
 };
};

class DEPARTMENT
{
 extent ALL_DEPARTMENTS
 key Dname, Dnumber
 {
 attribute string Dname;
 attribute short Dnumber;
 attribute struct Dept_mgr {EMPLOYEE Manager, date Start_date}
 Mgr;
 attribute set<string> Locations;
 attribute struct Projs (string Proj_name, time Weekly_hours)
 Projs;
 relationship set<EMPLOYEE> Has_emps inverse EMPLOYEE::Works_for;
 void add_emp(in string New_ename) raises(ename_not_valid);
 void change_manager(in string New_mgr_name; in date
 Start_date);
 };
};

```

**Figure .** The attributes, relationships, and operations in a class definition.

By specifying inverses, the database system can maintain the referential integrity of the relationship automatically. That is, if the value of Works\_for for a particular EMPLOYEE  $E$  refers to DEPARTMENT  $D$ , then the value of Has\_emps for DEPARTMENT  $D$  must include a reference to  $E$  in its set of EMPLOYEE references. If the database designer desires to have a relationship to be represented in *only one direction*, then

it has to be modeled as an attribute (or operation). An example is the Manager component of the Mgr attribute in DEPARTMENT.

In addition to attributes and relationships, the designer can include **operations** in object type (class) specifications. Each object type can have a number of **operation signatures**, which specify the operation name, its argument types, and its returned value, if applicable. Operation names are unique within each object type, but they can be overloaded by having the same operation name appear in distinct object types. The operation signature can also specify the names of **exceptions** that can

occur during operation execution. The implementation of the operation will include the code to raise these exceptions. In Figure the EMPLOYEE class has one operation: reassign\_emp, and the DEPARTMENT class has two operations: add\_emp and change\_manager.

### **Extents, Keys, and Factory Objects**

In the ODMG object model, the database designer can declare an *extent* (using the keyword extent) for any object type that is defined via a class declaration. The extent is given a name, and it will contain all persistent objects of that class. Hence, the extent behaves as a *set object* that holds all persistent objects of the class. In Figure the EMPLOYEE and DEPARTMENT classes have extents called ALL\_EMPLOYEES and ALL\_DEPARTMENTS, respectively. This is similar to creating two objects—one of type set<EMPLOYEE> and the second of type set<DEPARTMENT>—and making them persistent by naming them ALL\_EMPLOYEES and ALL\_DEPARTMENTS. Extents are also used to automatically enforce the set/subset relationship between the extents of a supertype and its subtype. If two classes A and B have extents ALL\_A and ALL\_B, and class B is a subtype of class A (that is, class B extends class A), then the collection of objects in ALL\_B must be a subset of those in ALL\_A at any point. This constraint is automatically enforced by the database system.

A class with an extent can have one or more keys. A **key** consists of one or more properties (attributes or relationships) whose values are constrained to be unique for each object in the extent. For example, in Figure the EMPLOYEE class has the Ssn attribute as key (each EMPLOYEE object in the extent must have a unique Ssn value), and the DEPARTMENT class has two distinct keys: Dname and Dnumber (each DEPARTMENT must have a unique Dname and a unique Dnumber). For a composite key that is made of several properties, the properties that form the key are contained in parentheses. For example, if a class VEHICLE with an extent ALL\_VEHICLES has a key made up of a combination of two attributes State and License\_number, they would be placed in parentheses as (State, License\_number) in the key declaration.

Next, we present the concept of **factory object**—an object that can be used to generate or create individual objects via its operations. Some of the interfaces of factory objects that are part of the ODMG object model are shown in Figure. The interface ObjectFactory has a single operation, new(), which returns a new object with an Object\_id. By inheriting this interface, users can create their own factory interfaces for each user-defined (atomic) object type, and the programmer can implement the operation *new* differently for each type of object. Figure also shows a DateFactory interface, which has additional operations for creating a new calendar\_date, and for creating an object whose value is the current\_date, among other operations. As we can see, a factory object basically provides the **constructor operations** for new objects.

```

interface ObjectFactory {
 Object new();
};

interface SetFactory : ObjectFactory {
 Set new_of_size(in long size);
};

interface ListFactory : ObjectFactory {
 List new_of_size(in long size);
};

interface ArrayFactory : ObjectFactory {
 Array new_of_size(in long size);
};

interface DictionaryFactory : ObjectFactory {
 Dictionary new_of_size(in long size);
};

interface DictionaryFactory : ObjectFactory {
 Dictionary new_of_size(in long size);
};

interface DateFactory : ObjectFactory {
 exception InvalidDate();
 ...
 Date calendar_date(in unsigned short year,
 in unsigned short month,
 in unsigned short day)
 raises(InvalidDate());
 ...
 Date current();
};

interface DatabaseFactory {
 Database new();
};

interface Database {
 ...
 void open(in string database_name)
 raises(DatabaseNotFound, DatabaseOpen);
 void close() raises(DatabaseClosed, ...);
 void bind(in Object an_object, in string name)
 raises(DatabaseClosed, ObjectNameNotUnique, ...);
 Object unbind(in string name)
 raises(DatabaseClosed, ObjectNameNotFound, ...);
 Object lookup(in string object_name)
 raises(DatabaseClosed, ObjectNameNotFound, ...);
 ...
};

```

**Figure.** Interfaces to illustrate factory objects and database objects.

Finally, we discuss the concept of a **database**. Because an ODBMS can create many different databases, each with its own schema, the ODMG object model has interfaces for DatabaseFactory and Database objects, as shown in Figure. Each database has its own *database name*, and the bind operation can be used to assign individual unique names to persistent objects in a particular database. The lookup operation returns an object from the database that has the specified *object\_name*, and the unbind operation removes the name of a persistent named object from the database.

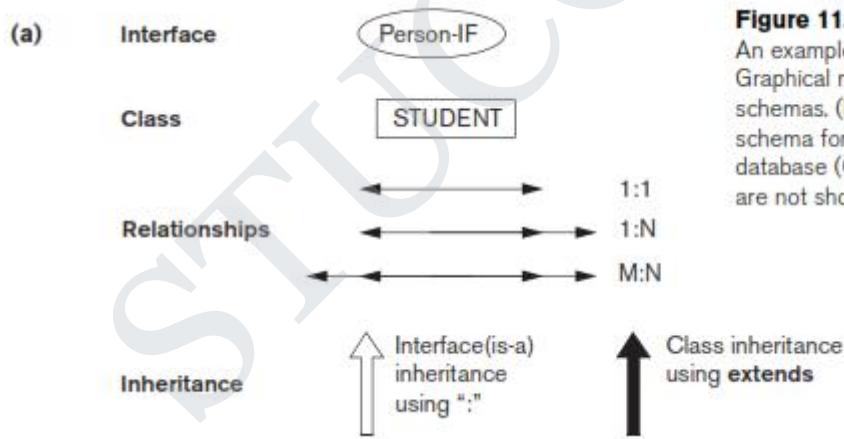
## 5.9. THE OBJECT DEFINITION LANGUAGE (ODL)

The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any particular programming language. Its main use is to create object specifications—that is, classes and interfaces. Hence, ODL is not a full programming language. A user can specify a database schema in ODL independently of any programming language, and then use the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming languages, such as C++, Smalltalk, and Java.

Figure (b) shows a possible object schema for part of the UNIVERSITY database. The graphical notation for Figure(b) is shown in Figure (a) and can be considered as a variation of EER diagrams with the added concept of interface inheritance but without several EER concepts, such as categories (union types) and attributes of relationships.

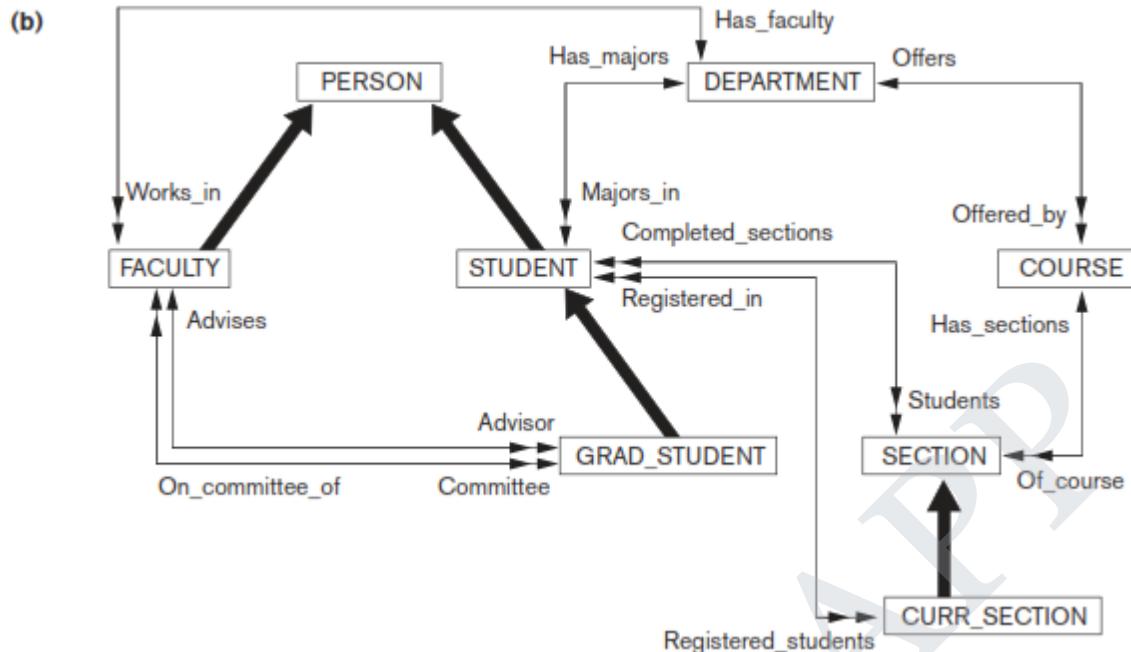
Figure shows one possible set of ODL class definitions for the UNIVERSITY database. In general, there may be several possible mappings from an object schema diagram (or EER schema diagram) into ODL classes.

Figure shows the straightforward way of mapping part of the UNIVERSITY database. Entity types are mapped into ODL classes, and inheritance is done using extends. However, there is no direct way to map categories (union types) or to do multiple inheritance. In Figure the classes PERSON, FACULTY, STUDENT, and GRAD\_STUDENT have the extents PERSONS, FACULTY, STUDENTS, and GRAD\_STUDENTS, respectively. Both FACULTY and STUDENT extends PERSON and GRAD\_STUDENT extends STUDENT. Hence, the collection of STUDENTS (and the collection of FACULTY) will be constrained to be a subset of the collection of PERSONS at any time. Similarly, the collection of GRAD\_STUDENTS will be a subset of STUDENTS. At the same time, individual STUDENT and FACULTY objects will inherit the properties (attributes and relationships) and operations of PERSON, and individual GRAD\_STUDENT objects will inherit those of STUDENT.



**Figure 11.9**

An example of a database schema. (a) Graphical notation for representing ODL schemas. (b) A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).



```

class PERSON
{
 extent PERSONS
 key Ssn
 attribute struct Pname {
 string Fname,
 string Mname,
 string Lname } Name;
 attribute string Ssn;
 attribute date Birth_date;
 attribute enum Gender{M, F} Sex;
 attribute struct Address {
 short No,
 string Street,
 short Apt_no,
 string City,
 string State,
 short Zip } Address;
 short Age();
}

```

```

class FACULTY extends PERSON
(extent FACULTY)
{
 attribute string Rank;
 attribute float Salary;
 attribute string Office;
 attribute string Phone;
 relationship DEPARTMENT Works_in inverse DEPARTMENT::Has_faculty;
 relationship set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
 relationship set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
 void give_raise(in float raise);
 void promote(in string new_rank); };

class GRADE
(extent GRADES)
{
 attribute enum GradeValues{A,B,C,D,F,I, P} Grade;
 relationship SECTION Section inverse SECTION::Students;
 relationship STUDENT Student inverse STUDENT::Completed_sections; };

class STUDENT extends PERSON
(extent STUDENTS)
{
 attribute string Class;
 attribute DEPARTMENT Minors_in;
 relationship DEPARTMENT Majors_in inverse DEPARTMENT::Has_majors;
 relationship set<GRADE> Completed_sections inverse GRADE::Student;
 relationship set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
 void change_major(in string dname) raises(dname_not_valid);
 float gpa();
 void register(in short secno) raises(section_not_valid);
 void assign_grade(in short secno; IN GradeValue grade)
 raises(section_not_valid,grade_not_valid); };

class DEGREE
{
 attribute string College;
 attribute string Degree;
 attribute string Year; };

class GRAD_STUDENT extends STUDENT
(extent GRAD_STUDENTS)
{
 attribute set<DEGREE> Degrees;
 relationship FACULTY Advisor inverse FACULTY::Advises;
 relationship set<FACULTY> Committee inverse FACULTY::On_committee_of;
 void assign_advisor(in string Lname; in string Fname)
 raises(faculty_not_valid);
 void assign_committee_member(in string Lname; in string Fname)
 raises(faculty_not_valid); };

```

```

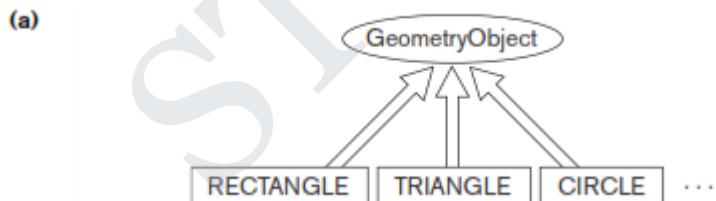
class DEPARTMENT
(extent DEPARTMENTS
 key Dname)
{
 attribute string Dname;
 attribute string Dphone;
 attribute string Doffice;
 attribute string College;
 attribute FACULTY Chair;
 relationship set<FACULTY> Has_faculty inverse FACULTY::Works_in;
 relationship set<STUDENT> Has_majors inverse STUDENT::Majors_in;
 relationship set<COURSE> Offers inverse COURSE::Offered_by; };

class COURSE
(extent COURSES
 key Cno)
{
 attribute string Cname;
 attribute string Cno;
 attribute string Description;
 relationship set<SECTION> Has_sections inverse SECTION::Of_course;
 relationship <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };

class SECTION
(extent SECTIONS)
{
 attribute short Sec_no;
 attribute string Year;
 attribute enum Quarter{Fall, Winter, Spring, Summer}
 Qtr;
 relationship set<GRADE> Students inverse GRADE::Section;
 relationship course Of_course inverse COURSE::Has_sections; };

class CURR_SECTION extends SECTION
(extent CURRENT_SECTIONS)
{
 relationship set<STUDENT> Registered_students
 inverse STUDENT::Registered_in
 register_student(in string Ssn)
 raises(student_not_valid, section_full); };

```

**Figure .** Possible ODL schema for the UNIVERSITY database

```

(b) interface GeometryObject
 { attribute enum Shape{RECTANGLE, TRIANGLE, CIRCLE, ... }
 attribute struct Point {short x, short y} Reference_point;
 float perimeter();
 float area();
 void translate(in short x_translation; in short y_translation);
 void rotate(in float angle_of_rotation); };

class RECTANGLE : GeometryObject
(extent RECTANGLES)
{ attribute struct Point {short x, short y} Reference_point;
 attribute short Length;
 attribute short Height;
 attribute float Orientation_angle; };

class TRIANGLE : GeometryObject
(extent TRIANGLES)
{ attribute struct Point {short x, short y} Reference_point;
 attribute short Side_1;
 attribute short Side_2;
 attribute float Side1_side2_angle;
 attribute float Side1_orientation_angle; };

class CIRCLE : GeometryObject
(extent CIRCLES)
{ attribute struct Point {short x, short y} Reference_point;
 attribute short Radius; };

...

```

**Figure.** An illustration of interface inheritance via “:”.

- (a) Graphical schema representation,
- (b) Corresponding interface and class definitions in ODL.

The classes DEPARTMENT, COURSE, SECTION, and CURR\_SECTION in Figure are straightforward mappings of the corresponding entity types in figure(b). However, the class GRADE requires some explanation. The GRADE class corresponds to the M:N relationship between STUDENT and SECTION in Figure (b). The reason it was made into a separate class (rather than as a pair of inverse relationships) is because it includes the relationship attribute Grade.

Hence, the M:N relationship is mapped to the class GRADE, and a pair of 1:N relationships, one between STUDENT and GRADE and the other between SECTION and GRADE. These relationships are represented by the following relationship proper-ties: Completed\_sections of STUDENT; Section and Student of GRADE; and Students of SECTION. Finally, the class DEGREE is used to represent the composite, multivalued attribute degrees of GRAD\_STUDENT.

Because the previous example does not include any interfaces, only classes, we now utilize a different example to illustrate interfaces and interface (behavior) inheritance. Figure(a) is part of a database schema for storing geometric objects. An interface GeometryObject is specified, with operations to calculate the perimeter and area of a geometric object, plus operations to translate (move) and rotate an object. Several classes (RECTANGLE, TRIANGLE, CIRCLE, ...) inherit the GeometryObject interface. Since GeometryObject is an interface, it is *noninstantiable*—that is, no objects can be created based on this interface directly. However, objects of type RECTANGLE, TRIANGLE, CIRCLE, ... can be created, and these objects inherit all the operations of the GeometryObject interface. Note that with interface inheritance, only operations are inherited,

not properties (attributes, relationships). Hence, if a property is needed in the inheriting class, it must be repeated in the class definition, as with the `Reference_point` attribute in Figure(b). Notice that the inherited operations can have different implementations in each class. For example, the implementations of the area and perimeter operations may be different for RECTANGLE, TRIANGLE, and CIRCLE.

*Multiple inheritance* of interfaces by a class is allowed, as is multiple inheritance of interfaces by another interface. However, with the `extends` (class) inheritance, multiple inheritance is *not permitted*. Hence, a class can inherit via `extends` from at most one class (in addition to inheriting from zero or more interfaces).

## 5.10. OBJECT QUERY LANGUAGE( OQL)

The object query language OQL is the query language proposed for the ODMG object model. It is designed to work closely with the programming languages for which an ODMG binding is defined, such as C++, Smalltalk, and Java. Hence, an OQL query embedded into one of these programming languages can return objects that match the type system of that language. Additionally, the implementations of class operations in an ODMG schema can have their code written in these programming languages. The OQL syntax for queries is similar to the syntax of the relational standard query language SQL, with additional features for ODMG concepts, such as object identity, complex objects, operations, inheritance, polymorphism, and relationships.

### Simple OQL Queries, Database Entry Points, and Iterator Variables

The basic OQL syntax is a `select ... from ... where ...` structure, as it is for SQL. For example, the query to retrieve the names of all departments in the college of ‘Engineering’ can be written as follows:

```
Q0: select D.Dname
 from D in DEPARTMENTS
 where D.College = 'Engineering';
```

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object*. For many queries, the entry point is the name of the extent of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. Looking at the extent names in Figure, the named object DEPARTMENTS is of type `set<DEPARTMENT>`; PERSONS is of type `set<PERSON>`; FACULTY is of type `set<FACULTY>`; and so on.

The use of an extent name—DEPARTMENTS in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable**—`D` in Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the `where` clause. In Q0, only persistent objects `D` in the collection of DEPARTMENTS that satisfy the condition `D.College = 'Engineering'` are selected for the query result. For each selected object `D`, the value of `D.Dname` is retrieved in the query result. Hence, the *type of the result* for Q0 is `bag<string>` because the type of each `Dname` value is string (even though the actual result is a set because `Dname` is a key attribute). In general, the result of a query would be of type `bag` for `select ... from ...` and of type `set` for `select distinct ... from ...`, as in SQL (adding the keyword `distinct` eliminates duplicates).

Using the example in Q0, there are three syntactic options for specifying iterator variables:

in DEPARTMENTS DEPARTMENTS `D` DEPARTMENTS AS `D`

We will use the first construct in our examples.

The named objects used as database entry points for OQL queries are not limited to the names of extents. Any named persistent object, whether it refers to an atomic (single) object or to a collection object, can be used as a database entry point.

### **Query Results and Path Expressions**

In general, the result of a query can be of any type that can be expressed in the ODMG object model. A query does not have to follow the select ... from ... where ...

structure; in the simplest case, any persistent name on its own is a query, whose result is a reference to that persistent object. For example, the query

Q1: DEPARTMENTS;

returns a reference to the collection of all persistent DEPARTMENT objects, whose type is set<DEPARTMENT>. Similarly, suppose we had given a persistent name CS\_DEPARTMENT to a single DEPARTMENT object (the Computer Science department); then, the query

Q1A: CS\_DEPARTMENT;

returns a reference to that individual object of type DEPARTMENT. Once an entry point is specified, the concept of a **path expression** can be used to specify a *path* to related attributes and objects. A path expression typically starts at a *persistent object name*, or at the iterator variable that ranges over individual objects in a collection. This name will be followed by zero or more relationship names or attribute names connected using the *dot notation*. For example, referring to the UNIVERSITY data-base in Figure, the following are examples of path expressions, which are also valid queries in OQL:

Q2: CS\_DEPARTMENT.Chiar;

Q2A: CS\_DEPARTMENT.Chiar.Rank;

Q2B: CS\_DEPARTMENT.Has\_faculty;

The first expression Q2 returns an object of type FACULTY, because that is the type of the attribute Chair of the DEPARTMENT class. This will be a reference to the FACULTY object that is related to the DEPARTMENT object whose persistent name is CS\_DEPARTMENT via the attribute Chair; that is, a reference to the FACULTY object who is chairperson of the Computer Science department. The second expression Q2A is similar, except that it returns the Rank of this FACULTY object (the Computer Science chair) rather than the object reference; hence, the type returned by Q2A is string, which is the data type for the Rank attribute of the FACULTY class.

Path expressions Q2 and Q2A return single values, because the attributes Chair (of DEPARTMENT) and Rank (of FACULTY) are both single-valued and they are applied to a single object. The third expression, Q2B, is different; it returns an object of type set<FACULTY> even when applied to a single object, because that is the type of the relationship Has\_faculty of the DEPARTMENT class. The collection returned will include references to all FACULTY objects that are related to the DEPARTMENT object whose persistent name is CS\_DEPARTMENT via the relationship Has\_faculty; that is, references to all FACULTY objects who are working in the Computer Science department. Now, to return the ranks of Computer Science faculty, we *cannot* write

Q3 : CS\_DEPARTMENT.Has\_faculty.Rank;

because it is not clear whether the object returned would be of type set<string> or bag<string> (the latter being more likely, since multiple faculty may share the same rank). Because of this type of ambiguity problem, OQL does not allow expressions such as Q3 . Rather, one must use an iterator variable over any collections, as in Q3A or Q3B below:

Q3A: select F.Rank  
from F in

```

 CS_DEPARTMENT.Has_faculty;
Q3B: select distinct F.Rank
 F in
from CS_DEPARTMENT.Has_faculty;

```

Here, Q3A returns bag<string> (duplicate rank values appear in the result), whereas Q3B returns set<string> (duplicates are eliminated via the distinct keyword). Both Q3A and Q3B illustrate how an iterator variable can be defined in the from clause to range over a restricted collection specified in the query. The variable *F* in Q3A and Q3B ranges over the elements of the collection CS\_DEPARTMENT.Has\_faculty, which is of type set<FACULTY>, and includes only those faculty who are members of the Computer Science department.

In general, an OQL query can return a result with a complex structure specified in the query itself by utilizing the struct keyword. Consider the following examples:

Q4: CS\_DEPARTMENT.Chamber.Advises;

```

Q4A: select struct (name: struct (last_name: S.name.Lname, first_name:
 S.name.Fname), degrees:(select struct (deg: D.Degree, yr: D.Year,
 college: D.College) from D in S.Degrees)) from S in
 CS_DEPARTMENT.Chamber.Advises;

```

Here, Q4 is straightforward, returning an object of type set<GRAD\_STUDENT> as its result; this is the collection of graduate students who are advised by the chair of the Computer Science department. Now, suppose that a query is needed to retrieve the last and first names of these graduate students, plus the list of previous degrees of each. This can be written as in Q4A, where the variable *S* ranges over the collection of graduate students advised by the chairperson, and the variable *D* ranges over the degrees of each such student *S*. The type of the result of Q4A is a collection of (first-level) structs where each struct has two components: name and degrees.

The name component is a further struct made up of last\_name and first\_name, each being a single string. The degrees component is defined by an embedded query and is itself a collection of further (second level) structs, each with three string components: deg, yr, and college.

Note that OQL is *orthogonal* with respect to specifying path expressions. That is, attributes, relationships, and operation names (methods) can be used interchangeably within the path expressions, as long as the type system of OQL is not compromised. For example, one can write the following queries to retrieve the grade point average of all senior students majoring in Computer Science, with the result ordered by GPA, and within that by last and first name:

```

select struct (last_name: S.name.Lname, first_name:
Q5A: S.name.Fname,
 gpa: S.gpa)
from S in CS_DEPARTMENT.Has_majors
where S.Class = 'senior'
order by gpa desc, last_name asc, first_name asc;
select struct (last_name: S.name.Lname, first_name:
Q5B: S.name.Fname,
 gpa: S.gpa)
from S in STUDENTS
where S.Majors_in.Dname = 'Computer Science' and
 S.Class = 'senior'
order by gpa desc, last_name asc, first_name asc;

```

Q5A used the named entry point CS\_DEPARTMENT to directly locate the reference to the Computer Science department and then locate the students via the relationship Has\_majors, whereas Q5B

searches the STUDENTS extent to locate all students majoring in that department. Notice how attribute names, relationship names, and operation (method) names are all used interchangeably (in an orthogonal manner) in the path expressions: gpa is an operation; Majors\_in and Has\_majors are relationships; and Class, Name, Dname, Lname, and Fname are attributes. The implementation of the gpa operation computes the grade point average and returns its value as a float type for each selected STUDENT.

The order by clause is similar to the corresponding SQL construct, and specifies in which order the query result is to be displayed. Hence, the collection returned by a query with an order by clause is of type *list*.

### **Other Features of OQL:**

**Specifying Views as Named Queries.** The view mechanism in OQL uses the concept of a **named query**. The define keyword is used to specify an identifier of the named query, which must be a unique name among all named objects, class names, method names, and function names in the schema. If the identifier has the same name as an existing named query, then the new definition replaces the previous definition. Once defined, a query definition is persistent until it is redefined or deleted. A view can also have parameters (arguments) in its definition.

For example, the following view V1 defines a named query Has\_minors to retrieve the set of objects for students minoring in a given department:

```
V1: define Has_minors(Dept_name) as
 select S
 from S in STUDENTS
 where S.Minors_in.Dname = Dept_name;
```

Because the ODL schema in Figure 11.10 only provided a unidirectional Minors\_in attribute for a STUDENT, we can use the above view to represent its inverse without having to explicitly define a relationship. This type of view can be used to represent inverse relationships that are not expected to be used frequently. The user can now utilize the above view to write queries such as

```
Has_minors('Computer Science');
```

which would return a bag of students minoring in the Computer Science department. Note that in Figure, we defined Has\_majors as an explicit relationship, presumably because it is expected to be used more often.

**Extracting Single Elements from Singleton Collections.** An OQL query will, in general, return a collection as its result, such as a bag, set (if distinct is specified), or list (if the order by clause is used). If the user requires that a query only return a single element, there is an element operator in OQL that is guaranteed to return a single element *E* from a singleton collection *C* that contains only one element. If *C* contains more than one element or if *C* is empty, then the element operator *raises an exception*. For example, Q6 returns the single object reference to the Computer Science department:

```
Q6: element (select D
 from D in DEPARTMENTS
 where D.Dname = 'Computer Science');
```

Since a department name is unique across all departments, the result should be one department. The type of the result is *D:DEPARTMENT*.

**Collection Operators (Aggregate Functions, Quantifiers).** Because many query expressions specify collections as their result, a number of operators have been defined that are applied to such collections. These include aggregate operators as well as membership and quantification (universal and existential) over a collection.

The aggregate operators (min, max, count, sum, avg) operate over a collection. The operator count returns an integer type. The remaining aggregate operators (min, max, sum, avg) return the same type as the type of the operand collection. Two examples follow. The query Q7 returns the number of students minoring in Computer Science and Q8 returns the average GPA of all seniors majoring in Computer Science.

Q7: count ( S in Has\_minors('Computer Science'));

Q8: avg ( select S.Gpa  
from S in STUDENTS  
where S.Majors\_in.Dname = 'Computer Science' and  
S.Class = 'Senior');

Notice that aggregate operations can be applied to any collection of the appropriate type and can be used in any part of a query. For example, the query to retrieve all department names that have more than 100 majors can be written as in Q9:

Q9: select D.Dname  
from D in DEPARTMENTS  
where count (D.Has\_majors) > 100;

The *membership* and *quantification* expressions return a Boolean type—that is, true or false. Let  $V$  be a variable,  $C$  a collection expression,  $B$  an expression of type Boolean (that is, a Boolean condition), and  $E$  an element of the type of elements in collection  $C$ . Then:

$(E \text{ in } C)$  returns true if element  $E$  is a member of collection  $C$ .

$(\text{for all } V \text{ in } C : B)$  returns true if *all* the elements of collection  $C$  satisfy  $B$ .

$(\text{exists } V \text{ in } C : B)$  returns true if there is at least one element in  $C$  satisfying  $B$ .

To illustrate the membership condition, suppose we want to retrieve the names of all students who completed the course called 'Database Systems I'. This can be written as in Q10, where the nested query returns the collection of course names that each STUDENT  $S$  has completed, and the membership condition returns true if 'Database Systems I' is in the collection for a particular STUDENT  $S$ :

Q10: select S.name.Lname, S.name.Fname  
From S in STUDENTS  
Where 'Database Systems I' in  
( select C.Section.Of\_course.Cname  
From C in S.Completed\_sections);

Q10 also illustrates a simpler way to specify the select clause of queries that return a collection of structs; the type returned by Q10 is bag<struct(string, string)>.

One can also write queries that return true/false results. As an example, let us assume that there is a named object called JEREMY of type STUDENT. Then, query Q11 answers the following question: *Is Jeremy a Computer Science minor?* Similarly, Q12 answers the question *Are all Computer Science graduate students advised by Computer Science faculty?* Both Q11 and Q12 return true or false, which are interpreted as yes or no answers to the above questions:

Q11: JEREMY in Has\_minors('Computer Science');

Q12: for all  $G$  in  
( select S  
from S in GRAD\_STUDENTS  
where S.Majors\_in.Dname = 'Computer Science' )  
: G.Advisor in CS\_DEPARTMENT.Has\_faculty;

Note that query Q12 also illustrates how attribute, relationship, and operation inheritance applies to queries. Although  $S$  is an iterator that ranges over the extent GRAD\_STUDENTS, we can write  $S$ .Majors\_in because the Majors\_in relationship is inherited by GRAD\_STUDENT from STUDENT via extends. Finally, to illustrate the exists quantifier, query Q13 answers the following question: *Does any graduate Computer Science major have a 4.0 GPA?* Here, again, the operation gpa is inherited by GRAD\_STUDENT from STUDENT via extends.

Q13: exists  $G$  in

```
select S from S in GRAD_STUDENTS
where S .Majors_in.Dname = 'Computer Science') : G .Gpa = 4;
```

**Ordered (Indexed) Collection Expressions.** As we discussed in Section 11.3.3, collections that are lists and arrays have additional operations, such as retrieving the  $i$ th, first, and last elements. Additionally, operations exist for extracting a subcollection and concatenating two lists. Hence, query expressions that involve lists or arrays can invoke these operations. We will illustrate a few of these operations using sample queries. Q14 retrieves the last name of the faculty member who earns the highest salary:

Q14: first ( select struct(facname:  $F$ .name.Lname, salary:  $F$ .Salary)
from  $F$  in FACULTY
order by salary desc );

Q14 illustrates the use of the first operator on a list collection that contains the salaries of faculty members sorted in descending order by salary. Thus, the first element in this sorted list contains the faculty member with the highest salary. This query assumes that only one faculty member earns the maximum salary. The next query, Q15, retrieves the top three Computer Science majors based on GPA.

Q15: ( select struct( last\_name:  $S$ .name.Lname, first\_name:  $S$ .name.Fname,
gpa:  $S$ .Gpa )
from  $S$  in CS\_DEPARTMENT.Has\_majors
order by gpa desc )[0:2];

The select-from-order-by query returns a list of Computer Science students ordered by GPA in descending order. The first element of an ordered collection has an index position of 0, so the expression [0:2] returns a list containing the first, second, and third elements of the select ... from ... order by ... result.

**The Grouping Operator.** The group by clause in OQL, although similar to the corresponding clause in SQL, provides explicit reference to the collection of objects within each *group* or *partition*. First we give an example, and then we describe the general form of these queries.

Q16 retrieves the number of majors in each department. In this query, the students are grouped into the same partition (group) if they have the same major; that is, the

same value for  $S$ .Majors\_in.Dname:

Q16: ( select struct( dept\_name, number\_of\_majors: count(partition) )
from  $S$  in STUDENTS
group by dept\_name:  $S$ .Majors\_in.Dname;

The result of the grouping specification is of type set<struct(dept\_name: string, partition: bag<struct(S:STUDENT)>)>, which contains a struct for each group (partition) that has two components: the grouping attribute value (dept\_name) and the bag of the STUDENT objects in the group

(partition). The select clause returns the grouping attribute (name of the department), and a count of the number of elements in each partition (that is, the number of students in each department), where partition is the keyword used to refer to each partition. The result type of the select clause is set<struct(dept\_name: string, number\_of\_majors: integer)>. In general, the syntax for the group by clause is

group by  $F_1: E_1, F_2: E_2, \dots, F_k: E_k$

where  $F_1: E_1, F_2: E_2, \dots, F_k: E_k$  is a list of partitioning (grouping) attributes and each partitioning attribute specification  $F_i: E_i$  defines an attribute (field) name  $F_i$  and an expression  $E_i$ . The result of applying the grouping (specified in the group by clause) is a set of structures:

set<struct( $F_1: T_1, F_2: T_2, \dots, F_k: T_k$ , partition: bag< $B$ >)>

where  $T_i$  is the type returned by the expression  $E_i$ , partition is a distinguished field name (a keyword), and  $B$  is a structure whose fields are the iterator variables ( $S$  in Q16) declared in the from clause having the appropriate type.

Just as in SQL, a having clause can be used to filter the partitioned sets (that is, select only some of the groups based on group conditions). In Q17, the previous query is modified to illustrate the having clause (and also shows the simplified syntax for the select clause). Q17 retrieves for each department having more than 100 majors, the average GPA of its majors. The having clause in Q17 selects only those partitions (groups) that have more than 100 elements (that is, departments with more than 100 students).

```
Q17: select dept_name, avg_gpa: avg (select P.gpa from P in partition)
 from S in STUDENTS
 group by dept_name: S.Majors_in.Dname
 having count (partition) > 100;
```

Note that the select clause of Q17 returns the average GPA of the students in the partition. The expression

select  $P$ .Gpa from  $P$  in partition

returns a bag of student GPAs for that partition. The from clause declares an iterator variable  $P$  over the partition collection, which is of type bag<struct( $S$ : STUDENT)>. Then the path expression  $P$ .gpa is used to access the GPA of each student in the partition.

## **5.11. XML DATABASES:**

- XML (Extensible Markup Language)—has emerged as the standard for structuring and exchanging data over the Web. XML can be used to provide information about the structure and meaning of the data in the Web pages rather than just specifying how the Web pages are formatted for display on the screen.
- The formatting aspects are specified separately—for example, by using a formatting language such as XSL (Extensible Stylesheet Language) or a transformation language such as XSLT (Extensible Stylesheet Language for Transformations or simply XSL Transformations). XML has also been proposed as a possible model for data storage and retrieval, although only a few experimental database systems based on XML have been developed so far.
- Basic HTML is useful for generating *static* Web pages with fixed text and other objects, but most e-commerce applications require Web pages that provide interactive features with the user. For example, consider the case of an airline customer who wants to check the arrival time and gate information of a particular flight. The user may enter information such as a date and flight number in certain form fields of the Web page. The Web program must first submit a query to the airline database to retrieve this information, and then display it. Such Web pages, where part of the information is extracted from databases or other data sources are called *dynamic* Web pages, because the data extracted and displayed each time will be for different flights and dates.

### Structured, Semistructured, and Unstructured Data:

The information stored in databases is known as **structured data** because it is represented in a strict format. For example, each record in a relational database table follows the same format as the other records in that table.

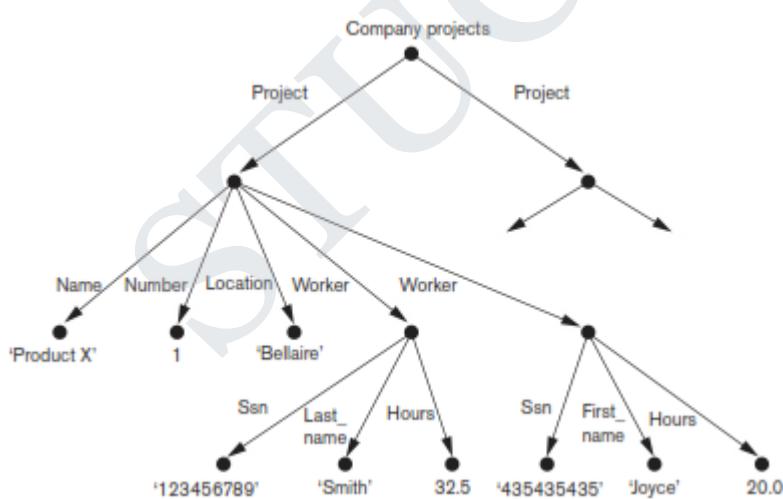
In some applications, data is collected in an ad hoc manner before it is known how it will be stored and managed. This data may have a certain structure, but not all the information collected will have the identical structure. Some attributes may be shared among the various entities, but other attributes may exist only in a few entities. Moreover, additional attributes can be introduced in some of the newer data items at any time, and there is no predefined schema. This type of data is known as **semistructured data**. A number of data models have been introduced for representing semistructured data, often based on using tree or graph data structures rather than the flat relational model structures.

A key difference between structured and semistructured data concerns how the schema constructs (such as the names of attributes, relationships, and entity types) are handled. In semistructured data, the schema information is *mixed in* with the data values, since each data object can have different attributes that are not known in advance. Hence, this type of data is sometimes referred to as **self-describing data**.

In Figure, the **labels** or **tags** on the directed edges represent the schema names: the *names of attributes*, *object types* (or *entity types* or *classes*), and *relationships*. The internal nodes represent individual objects or composite attributes. The leaf nodes represent actual data values of simple (atomic) attributes.

**There are two main differences between the semistructured model and the object model:**

- 1) The schema information—names of attributes, relationships, and classes (object types) in the semistructured model is intermixed with the objects and their data values in the same data structure.
- 2) In the semistructured model, there is no requirement for a predefined schema to which the data objects must conform, although it is possible to define a schema if necessary.



**Figure.** Representing semistructured data as a graph.

- In addition to structured and semistructured data, a third category exists, known as **unstructured data** because there is very limited indication of the type of data. A typical example is a text document that contains information embedded within it. Web pages in HTML that contain some data are considered to be unstructured data.

### **5.12. XML HIERARCHICAL (TREE) DATA MODEL**

The basic object in XML is the XML document. Two main structuring concepts are used to construct an XML document: **elements** and **attributes**. It is important to note that the term *attribute* in XML is not used in the same manner as is customary in database terminology, but rather as it is used in document description languages such as HTML and SGML. Attributes in XML provide additional information that describes elements, as we will see. There are additional concepts in XML, such as entities, identifiers, and references, but first we concentrate on describing elements and attributes to show the essence of the XML model.

Figure shows an example of an XML element called <Projects>. As in HTML, elements are identified in a document by their start tag and end tag. The tag names are enclosed between angled brackets < ... >, and end tags are further identified by a slash, </ ... >.

```
<?xml version="1.0" standalone="yes"?>
<Projects>
 <Project>
 <Name>ProductX</Name>
 <Number>1</Number>
 <Location>Bellaire</Location>
 <Dept_no>5</Dept_no>
 <Worker>
 <Ssn>123456789</Ssn>
 <Last_name>Smith</Last_name>
 <Hours>32.5</Hours>
 </Worker>
 <Worker>
 <Ssn>453453453</Ssn>
 <First_name>Joyce</First_name>
 <Hours>20.0</Hours>
 </Worker>
 </Project>
```

```

</Project>
<Project>
 <Name>ProductY</Name>
 <Number>2</Number>
 <Location>Sugarland</Location>
 <Dept_no>5</Dept_no>
 <Worker>
 <Ssn>123456789</Ssn>
 <Hours>7.5</Hours>
 </Worker>
 <Worker>
 <Ssn>453453453</Ssn>
 <Hours>20.0</Hours>
 </Worker>
 <Worker>
 <Ssn>333445555</Ssn>
 <Hours>10.0</Hours>
 </Worker>
</Project>
...
</Projects>

```

**Figure .** A complex XML element called <Projects>.

**Complex elements** are constructed from other elements hierarchically, whereas **simple elements** contain data values. A major difference between XML and HTML is that XML tag names are defined to describe the meaning of the data elements in the document, rather than to describe how the text is to be displayed. This makes it possible to process the data elements in the XML document automatically by computer programs. Also, the XML tag (element) names can be defined in another document, known as the *schema document*, to give a semantic meaning to the tag names that can be exchanged among multiple users. In HTML, all tag names are predefined and fixed; that is why they are not extendible.

It is straightforward to see the correspondence between the XML textual representation shown in Figure and the tree structure shown in Figure. In the tree representation, internal nodes represent complex elements, whereas leaf nodes represent simple elements. That is why the XML model is called a **tree model** or a **hierarchical model**. In Figure, the simple elements are the ones with the tag names <Name>, <Number>, <Location>, <Dept\_no>, <Ssn>, <Last\_name>, <First\_name>, and <Hours>. The complex elements are the ones with the tag names <Projects>, <Project>, and <Worker>. In general, there is no limit on the levels of nesting of elements.

It is possible to characterize three main types of XML documents:

**Data-centric XML documents.** These documents have many small data items that follow a specific structure and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them over or display them on the Web. These usually follow a *predefined schema* that defines the tag names.

**Document-centric XML documents.** These are documents with large amounts of text, such as news articles or books. There are few or no structured data elements in these documents.

**Hybrid XML documents.** These documents may have parts that contain structured data and other parts that are predominantly textual or unstructured. They may or may not have a predefined schema.

XML documents that do not follow a predefined schema of element names and corresponding tree structure are known as **schemaless XML documents**. It is important to note that data-centric XML documents can be considered either as semistructured data or as structured data as defined in Section 12.1. If an XML

document conforms to a predefined XML schema or DTD (see Section 12.3), then the document can be considered as *structured data*. On the other hand, XML allows documents that do not conform to any schema; these would be considered as *semistructured data* and are *schemaless XML documents*. When the value of the `standalone` attribute in an XML document is yes, as in the first line in Figure, the document is standalone and schemaless.

XML attributes are generally used in a manner similar to how they are used in HTML to describe properties and characteristics of the elements (tags) within which they appear. It is also possible to use XML attributes to hold the values of simple data elements; however, this is generally not recommended. An exception to this rule is in cases that need to **reference** another element in another part of the XML document. To do this, it is common to use attribute values in one element as the references. This resembles the concept of foreign keys in relational databases, and is a way to get around the strict hierarchical model that the XML tree model implies.

### **5.13.XML DOCUMENTS, DTD, AND XML SCHEMA:**

#### **Well-Formed and Valid XML Documents and XML DTD:**

An XML document is **well formed** if it follows a few conditions. In particular, it must start with an **XML declaration** to indicate the version of XML being used as well as any other relevant attributes, as shown in the first line in Figure. It must also follow the syntactic guidelines of the tree data model. This means that there should be a *single root element*, and every element must include a matching pair of start and end tags *within* the start and end tags of the *parent element*. This ensures that the nested elements specify a well-formed tree structure.

A well-formed XML document is syntactically correct. This allows it to be processed by generic processors that traverse the document and create an internal tree representation. A standard model with an associated set of API (application programming interface) functions called **DOM** (Document Object Model) allows programs to manipulate the resulting tree representation corresponding to a well-formed XML document. However, the whole document must be parsed beforehand when using DOM in order to convert the document to that standard DOM internal data structure representation. Another API called **SAX** (Simple API for XML) allows processing of XML documents on the fly by notifying the processing program through callbacks whenever a start or end tag is encountered. This makes it easier to process large documents and allows for processing of so-called **streaming XML documents**, where the processing program can process the tags as they are encountered. This is also known as **event-based processing**.

A well-formed XML document can be schemaless; that is, it can have any tag names for the elements within the document. In this case, there is no predefined set of elements (tag names) that a program processing the document knows to expect. This gives the document creator the freedom to specify new elements, but limits the possibilities for automatically interpreting the meaning or semantics of the elements within the document.

A stronger criterion is for an XML document to be **valid**. In this case, the document must be well formed, and it must follow a particular schema. That is, the element names used in the start and end tag pairs must follow the structure specified in a separate **XML DTD (Document Type Definition)** file or XML schema file. Figure shows a simple XML DTD file, which specifies the elements (tag names) and their nested structures. Any valid documents conforming to this DTD should follow the specified structure. A special syntax exists for specifying DTD files. First, a name is given to the **root tag** of the document, which is called Projects in the first line in Figure. Then the elements and their nested structure are specified.

```

<!DOCTYPE Projects [
 <!ELEMENT Projects (Project+)
 <!ELEMENT Project (Name, Number, Location, Dept_no?, Workers)
 <!ATTLIST Project
 ProjId ID #REQUIRED>
 >
 <!ELEMENT Name (#PCDATA)>
 <!ELEMENT Number (#PCDATA)>
 <!ELEMENT Location (#PCDATA)>
 <!ELEMENT Dept_no (#PCDATA)>
 <!ELEMENT Workers (Worker*)>
 <!ELEMENT Worker (Ssn, Last_name?, First_name?, Hours)>
 <!ELEMENT Ssn (#PCDATA)>
 <!ELEMENT Last_name (#PCDATA)>
 <!ELEMENT First_name (#PCDATA)>
 <!ELEMENT Hours (#PCDATA)>
]>

```

**Figure.** An XML DTD file called *Projects*.

When specifying elements, the following notation is used:

A \* following the element name means that the element can be repeated zero or more times in the document. This kind of element is known as an *optional multivalued (repeating) element*.

A + following the element name means that the element can be repeated one or more times in the document. This kind of element is a *required multivalued (repeating) element*.

A ? following the element name means that the element can be repeated zero or one times. This kind is an *optional single-valued (nonrepeating) element*.

An element appearing without any of the preceding three symbols must appear exactly once in the document. This kind is a *required single-valued (nonrepeating) element*.

The **type** of the element is specified via parentheses following the element. If the parentheses include names of other elements, these latter elements are the *children* of the element in the tree structure. If the parentheses include the keyword #PCDATA or one of the other data types available in XML DTD, the element is a leaf node. PCDATA stands for *parsed character data*, which is roughly similar to a string data type.

The list of attributes that can appear within an element can also be specified via the keyword !ATTLIST. In Figure 12.3, the Project element has an attribute ProjId. If the type of an attribute is ID, then it can be referenced from another attribute whose type is IDREF within another element. Notice that attributes can also be used to hold the values of simple data elements of type #PCDATA.

Parentheses can be nested when specifying elements.

A bar symbol ( $e_1 | e_2$ ) specifies that either  $e_1$  or  $e_2$  can appear in the document.

We can see that the tree structure in Figure 12.1 and the XML document in Figure 12.3 conform to the XML DTD in Figure 12.4. To require that an XML document be checked for conformance to a DTD, we must specify this in the declaration of the document. For example, we could change the first line in Figure 12.3 to the following:

```

<?xml version="1.0" standalone="no"?>
<!DOCTYPE Projects SYSTEM "proj.dtd">

```

When the value of the `standalone` attribute in an XML document is “no”, the document needs to be checked against a separate DTD document or XML schema document (see below). The DTD file shown in Figure 12.4 should be stored in the same file system as the XML document, and should be given the file name

proj.dtd. Alternatively, we could include the DTD document text at the beginning of the XML document itself to allow the checking.

Although XML DTD is quite adequate for specifying tree structures with required, optional, and repeating elements, and with various types of attributes, it has several limitations. First, the data types in DTD are not very general. Second, DTD has its own special syntax and thus requires specialized processors. It would be advantageous to specify XML schema documents using the syntax rules of XML itself so that the same processors used for XML documents could process XML schema descriptions. Third, all DTD elements are always forced to follow the specified ordering of the document, so unordered elements are not permitted. These draw-backs led to the development of XML schema, a more general but also more complex language for specifying the structure and elements of XML documents.

## **5.14. XML SCHEMA**

- An XML Schema describes the structure of an XML document, just like a DTD.
- The XML Schema language is also referred to as XML Schema Definition (XSD).
- An XML document is called "well-formed" if it contains the correct syntax. A well-formed and valid XML document is one which have been validated against Schema.

### **XML Schema Example**

Create a schema file.

employee.xsd

```
<?xml version="1.0"?>
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.javatpoint.com"
xmlns="http://www.javatpoint.com"
elementFormDefault="qualified">

<xselement name="employee">
 <xsccomplexType>
 <xsssequence>
 <xselement name="firstname" type="xs:string"/>
 <xselement name="lastname" type="xs:string"/>
 <xselement name="email" type="xs:string"/>
 </xsssequence>
 </xsccomplexType>
</xselement>
</xsschema>
```

xml file using XML schema or XSD file.

employee.xml

```
<?xml version="1.0"?>
<employee
 xmlns="http://www.javatpoint.com"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.javatpoint.com employee.xsd">
```

```
<firstname>vimal</firstname>
<lastname>jaiswal</lastname>
<email>vimal@javatpoint.com</email>
</employee>
```

### Description of XML Schema

**<xs:element name="employee">** :

It defines the element name employee.

**<xs:complexType>** :

It defines that the element 'employee' is complex type.

**<xs:sequence>** :

It defines that the complex type is a sequence of elements.

**<xs:element name="firstname" type="xs:string"/>** :

It defines that the element 'firstname' is of string/text type.

**<xs:element name="lastname" type="xs:string"/>** :

It defines that the element 'lastname' is of string/text type.

**<xs:element name="email" type="xs:string"/>** :

It defines that the element 'email' is of string/text type.

### XML Schema Data types

There are two types of data types in XML schema.

1. simpleType
2. complexType

#### **1) simpleType**

The simpleType allows you to have text-based elements. It contains less attributes, child elements, and cannot be left empty.

#### **2) complexType**

The complexType allows you to hold multiple attributes and elements. It can contain additional sub elements and can be left empty.

### DTD vs XSD

There are many differences between DTD (Document Type Definition) and XSD (XML Schema Definition). In short, DTD provides less control on XML structure whereas XSD (XML schema) provides more control. The important differences are given below:

No.	DTD	XSD
1)	<b>DTD stands for Document Type Definition.</b>	XSD stands for XML Schema Definition.
2)	DTDs are derived from SGML syntax.	XSDs are written in XML.
3)	<b>DTD doesn't support datatypes.</b>	XSD <b>supports datatypes</b> for elements and attributes.
4)	<b>DTD doesn't support namespace.</b>	XSD <b>supports namespace</b> .

5)	DTD <b>doesn't define order</b> for child elements.	XSD <b>defines order</b> for child elements.
6)	DTD is <b>not extensible</b> .	XSD is <b>extensible</b> .
7)	DTD is <b>not simple to learn</b> .	XSD is <b>simple to learn</b> because you don't need to learn new language.
8)	DTD provides <b>less control</b> on XML structure.	XSD provides <b>more control</b> on XML structure.

### 5.15. XQuery: SPECIFYING QUERIES IN XML

- XQuery permits the specification of more general queries on one or more XML documents. The typical form of a query in XQuery is known as a **FLWR expression**, which stands for the four main clauses of XQuery and has the following form:

```
FOR <variable bindings to individual nodes (elements)>
LET <variable bindings to collections of nodes (elements)>
WHERE <qualifier conditions>
RETURN <query result specification>
```

- There can be zero or more instances of the FOR clause, as well as of the LET clause in a single XQuery. The WHERE clause is optional, but can appear at most once, and the RETURN clause must appear exactly once. Let us illustrate these clauses with the following simple example of an XQuery.

```
LET $d := doc(www.company.com/info.xml)
FOR $x IN $d/company/project[projectNumber = 5]/projectWorker, $y IN
$d/company/employee
WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>
```

- Variables are prefixed with the \$ sign. In the above example, \$d, \$x, and \$y are variables.
- The LET clause assigns a variable to a particular expression for the rest of the query. In this example, \$d is assigned to the document file name. It is possible to have a query that refers to multiple documents by assigning multiple variables in this way.
- The FOR clause assigns a variable to range over each of the individual items in a sequence. In our example, the sequences are specified by path expressions. The \$x variable ranges over elements that satisfy the path expression \$d/company/project[projectNumber = 5]/projectWorker. The \$y variable ranges over elements that satisfy the path expression \$d/company/employee. Hence, \$x ranges over projectWorker elements, whereas \$y ranges over employee elements.
- The WHERE clause specifies additional conditions on the selection of items. In this example, the first condition selects only those projectWorker elements that satisfy the condition (hours gt 20.0). The second condition specifies a join condition that combines an employee with a projectWorker only if they have the same ssn value.
- Finally, the RETURN clause specifies which elements or attributes should be retrieved from the items that satisfy the query conditions. In this example, I will return a sequence of elements each containing <firstName, lastName, hours> for employees who work more than 20 hours per week on project number 5.
- Figure includes some additional examples of queries in XQuery that can be specified on an XML instance documents that follow the XML schema document in Figure 12.5. The first query retrieves the first and last names of employees who earn more than \$70,000. The variable \$x is bound to each employeeName

element that is a child of an employee element, but only for employee elements that satisfy the qualifier that their employeeSalary value is greater than \$70,000. The result retrieves the firstName and lastName child elements of the selected employeeName elements. The second query is an alternative way of retrieving the same elements retrieved by the first query.

- The third query illustrates how a join operation can be performed by using more than one variable. Here, the \$x variable is bound to each projectWorker element that is a child of project number 5, whereas the \$y variable is bound to each employee element. The join condition matches ssn values in order to retrieve the employee names. Notice that this is an alternative way of specifying the same query in our earlier example, but without the LET clause.
- XQuery has very powerful constructs to specify complex queries. In particular, it can specify universal and existential quantifiers in the conditions of a query, aggregate functions, ordering of query results, selection based on position in a sequence, and even conditional branching. Hence, in some ways, it qualifies as a fullfledged programming language.
- This concludes our brief introduction to XQuery. The interested reader is referred to [www.w3.org](http://www.w3.org), which contains documents describing the latest standards related to XML and XQuery.

```

1. FOR $x IN
 doc("http://www.company.com/info.xml")
 //employee [employeeSalary > 70000]/employeeName
 RETURN <res> $x/firstName, $x/lastName </res>

2. FOR $x IN
 doc("http://www.company.com/info.xml")/company/employee
 WHERE $x/employeeSalary > 70000
 RETURN <res> $x/employeeName/firstName, $x/employeeName/lastName </res>

3. FOR $x IN
 doc("http://www.company.com/info.xml")/company/project[projectNumber = 5]/projectWorker,
 $y IN doc("http://www.company.com/info.xml")/company/employee
 WHERE $x/hours > 20.0 AND $y/ssn = $x/ssn
 RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>

```

**Figure 12.7**

Some examples of XQuery queries on XML documents that follow the XML schema file *company* in Figure 12.5.

## **5.16. INFORMATION RETRIEVAL (IR) CONCEPTS**

- Information retrieval** is the process of retrieving documents from a collection in response to a query (or a search request) by a user.

### **Introduction to Information Retrieval**

Consider a relation (or table) called HOUSES with the attributes:

HOUSES(Lot#, Address, Square\_footage, Listed\_price)

- This is an example of **structured data**.
- We can compare this relation with home-buying contract documents, which are examples of **unstructured data**. These types of documents can vary from city to city, and even county to county, within a given state in the United States. Typically, a contract document in a particular state will have a standard list of clauses described in paragraphs within sections of the document, with some predetermined (fixed) text and some variable areas whose content is to be supplied by the specific buyer and seller.
- Other variable information would include interest rate for financing, down-payment amount, closing dates, and so on. The documents could also possibly include some pictures taken during a home inspection. The information content in such documents can be considered **unstructured data** that can be

stored in a variety of possible arrangements and formats. By **unstructured information**, we generally mean information that does not have a well-defined formal model and corresponding formal language for representation and reasoning, but rather is based on understanding of natural language.

- With the advent of the World Wide Web (or Web, for short), the volume of unstructured information stored in messages and documents that contain textual and multimedia information has exploded. These documents are stored in a variety of standard formats, including HTML, XML, and several audio and video formatting standards. Information retrieval deals with the problems of storing, indexing, and retrieving (searching) such information to satisfy the needs of users. The problems that IR deals with are exacerbated by the fact that the number of Web pages and the number of social interaction events is already in the billions, and is growing at a phenomenal rate. All forms of unstructured data described above are being added at the rates of millions per day, expanding the searchable space on the Web at rapidly increasing rates.
- Information retrieval** is “the discipline that deals with the structure, analysis, organization, storage, searching, and retrieval of information” .
- We can enhance the definition slightly to say that it applies in the context of unstructured documents to satisfy a user’s information needs. This field has existed even longer than the database field, and was originally concerned with retrieval of cataloged information in libraries based on titles, authors, topics, and keywords. In academic programs, the field of IR has long been a part of Library and Information Science programs. Information in the context of IR does not require machine-understandable structures, such as in relational database systems. Examples of such information include written texts, abstracts, documents, books, Web pages, e-mails, instant messages, and collections from digital libraries. Therefore, all loosely represented (unstructured) or semistructured information is also part of the IR discipline.
- RDBMS vendors are providing modules to support many of these data types, as well as XML data, in the newer versions of their products, sometimes referred to as *extended RDBMSs*, or *object-relational database management systems*. The challenge of dealing with unstructured data is largely an information retrieval problem, although database researchers have been applying data-base indexing and search techniques to some of these problems.
- IR systems go beyond database systems in that they do not limit the user to a specific query language, nor do they expect the user to know the structure (schema) or content of a particular database. IR systems use a user’s information need expressed as a **free-form search request** (sometimes called a **keyword search query**, or just **query**) for interpretation by the system. Whereas the IR field historically dealt with cataloging, processing, and accessing text in the form of documents for decades, in today’s world the use of Web search engines is becoming the dominant way to find information. The traditional problems of text indexing and making collections of documents searchable have been transformed by making the Web itself into a quickly accessible repository of human knowledge.

An IR system can be characterized at different levels:

- 1) Types of Users.** The user may be an *expert user* (for example, a curator or a librarian), who is searching for specific information that is clear in his/her mind and forms relevant queries for the task, or a *layperson user* with a generic information need. The latter cannot create highly relevant queries for search (for example, students trying to find information about a new topic, researchers trying to assimilate different points of view about a historical issue, a scientist verifying a claim by another scientist, or a person trying to shop for clothing).
- 2) Types of Data.** Search systems can be tailored to specific types of data. For example, the problem of retrieving information about a specific topic may be handled more efficiently by customized search systems that are built to collect and retrieve only information related to that specific topic. The information repository could be hierarchically organized based on a concept or topic hierarchy. These

topical *domain-specific* or *vertical IR systems* are not as large as or as diverse as the generic World Wide Web, which contains information on all kinds of topics. Given that these domain-specific collections exist and may have been acquired through a specific process, they can be exploited much more efficiently by a specialized system.

- 3) **Types of Information Need.** In the context of Web search, users' information needs may be defined as navigational, informational, or transactional.<sup>3</sup> **Navigational search** refers to finding a particular piece of information (such as the Georgia Tech University Website) that a user needs quickly. The purpose of **informational search** is to find current information about a topic (such as research activities in the college of computing at Georgia Tech—this is the classic IR system task). The goal of **transactional search** is to reach a site where further interaction happens (such as joining a social network, product shopping, online reservations, accessing databases, and so on).

#### Levels of Scale:

- This overabundance of information sources in effect creates a high noise-to-signal ratio in IR systems. Especially on the Web, where billions of pages are indexed, IR interfaces are built with efficient scalable algorithms for distributed searching, indexing, caching, merging, and fault tolerance. IR search engines can be limited in level to more specific collections of documents.
- **Enterprise search systems** offer IR solutions for searching different entities in an enterprise's **intranet**, which consists of the network of computers within that enterprise. The searchable entities include e-mails, corporate documents, manuals, charts, and presentations, as well as reports related to people, meetings, and projects. They still typically deal with hundreds of millions of entities in large global enterprises.
- On a smaller scale, there are personal information systems such as those on desktops and laptops, called **desktop search engines** (for example, Google Desktop), for retrieving files, folders, and different kinds of entities stored on the computer. There are peer-to-peer systems, such as BitTorrent, which allows sharing of music in the form of audio files, as well as specialized search engines for audio, such as Lycos and Yahoo! audio search.

#### Databases and IR Systems: A Comparison:

- Within the computer science discipline, databases and IR systems are closely related fields. Databases deal with structured information retrieval through well-defined formal languages for representation and manipulation based on the theoretically founded data models. Efficient algorithms have been developed for operators that allow rapid execution of complex queries. IR, on the other hand, deals with unstructured search with possibly vague query or search semantics and without a well-defined logical schematic representation. Some of the key differences between databases and IR systems are listed in Table.
- Whereas databases have fixed schemas defined in some data model such as the relational model, an IR system has no fixed data model; it views data or documents according to some scheme, such as the vector space model, to aid in query processing. Databases using the relational model employ SQL for queries and transactions. The queries are mapped into relational algebra operations and search algorithms and return a new relation (table) as the query result, providing an exact answer to the query for the current state of the database. In IR systems, there is no fixed language for defining the structure (schema) of the document or for operating on the document—queries tend to be a set of query terms (keywords) or a free-form natural language phrase. An IR query result is a list of document ids, or some pieces of text or multimedia objects (images, videos, and so on), or a list of links to Web pages.
- The result of a database query is an exact answer; if no matching records (tuples) are found in the relation, the result is empty (null). On the other hand, the answer to a user request in an IR query represents the IR system's best attempt at retrieving the information most relevant to that query.
- Whereas database systems maintain a large amount of metadata and allow their use in query optimization, the operations in IR systems rely on the data values themselves and their occurrence frequencies. Complex

statistical analysis is sometimes performed to determine the *relevance* of each document or parts of a document to the user request.

**Table.** A Comparison of Databases and IR Systems

Databases	IR Systems
■ Structured data	■ Unstructured data
■ Schema driven	■ No fixed schema; various data models (e.g., vector space model)
■ Relational (or object, hierarchical, and network) model is predominant	■ Free-form query models
■ Structured query model	■ Rich data operations
■ Rich metadata operations	■ Search request returns list or pointers to documents
■ Query returns data	■ Results are based on approximate matching and measures of effectiveness (may be imprecise and ranked)
■ Results are based on exact matching (always correct)	

- A **search engine** is a practical application of information retrieval to large-scale document collections. With significant advances in computers and communications technologies, people today have interactive access to enormous amounts of user-generated distributed content on the Web. This has spurred the rapid growth in search engine technology, where search engines are trying to discover different kinds of real-time content found on the Web.
- The part of a search engine responsible for discovering, analyzing, and indexing these new documents is known as a **crawler**. Other types of search engines exist for specific domains of knowledge.
- For example, the biomedical literature search database was started in the 1970s and is now supported by the PubMed search engine, which gives access to over 20 million abstracts.
- While continuous progress is being made to tailor search results to the needs of an end user, the challenge remains in providing high-quality, pertinent, and timely information that is precisely aligned to the information needs of individual users.

### **Modes of Interaction in IR Systems**

- Information retrieval is the process of retrieving documents from a collection in response to a query (or a search request) by a user. Typically the collection is made up of documents containing unstructured data. Other kinds of documents include images, audio recordings, video strips, and maps. Data may be scattered non uniformly in these documents with no definitive structure.
- A **query** is a set of **terms** (also referred to as **keywords**) used by the searcher to specify an information need (for example, the terms ‘databases’ and ‘operating systems’ may be regarded as a query to a computer science bibliographic database). An informational request or a search query may also be a natural language phrase or a question (for example, “What is the currency of China?” or “Find Italian restaurants in Sarasota, Florida.”).

There are two main modes of interaction with IR systems

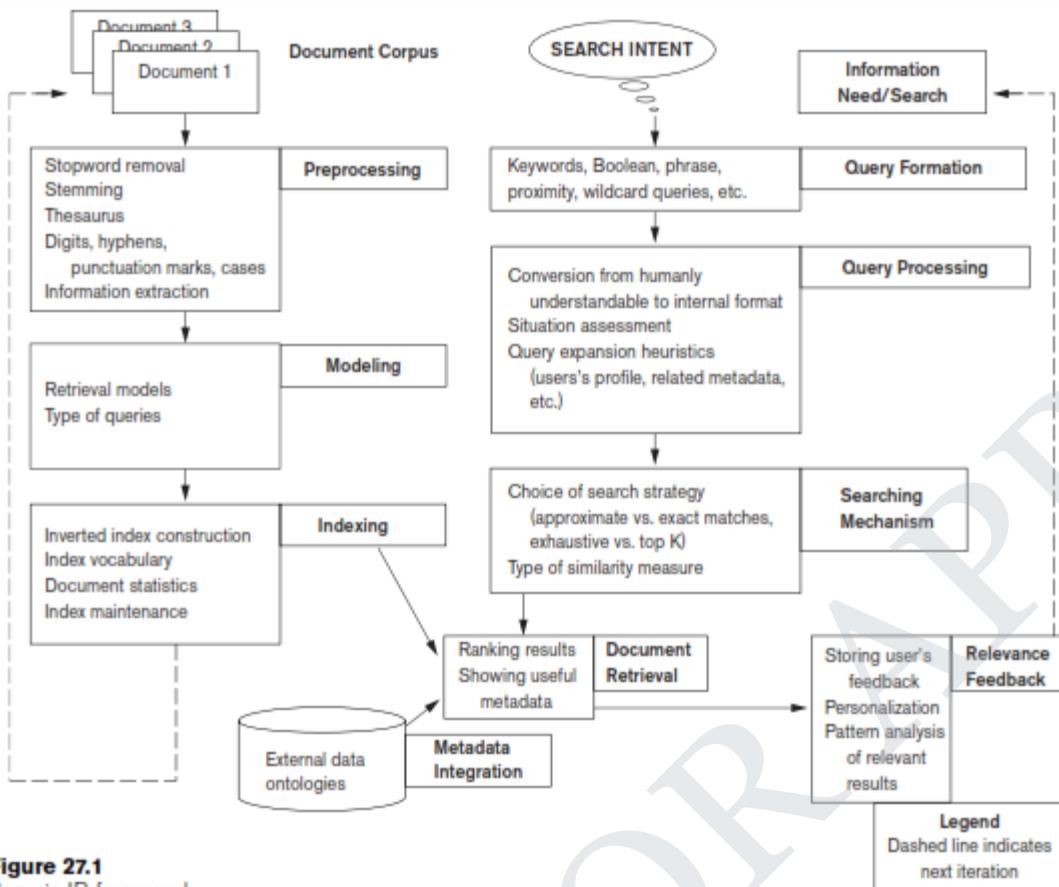
- 1) retrieval
- 2) browsing

- **Retrieval** is concerned with the extraction of relevant information from a repository of documents through an IR query
- **Browsing** signifies the activity of a user visiting or navigating through similar or related documents based on the user’s assessment of relevance. During browsing, a user’s information need may not be defined *a priori* and is flexible.

- Consider the following browsing scenario: A user specifies ‘Atlanta’ as a keyword. The information retrieval system retrieves links to relevant result documents containing various aspects of Atlanta for the user. The user comes across the term ‘Georgia Tech’ in one of the returned documents, and uses some access technique (such as clicking on the phrase ‘Georgia Tech’ in a document, which has a built-in link) and visits documents about Georgia Tech in the same or a different Website (repository). There the user finds an entry for ‘Athletics’ that leads the user to information about various athletic programs at Georgia Tech. Eventually, the user ends his search at the Fall schedule for the Yellow Jackets foot-ball team, which he finds to be of great interest. This user activity is known as browsing. **Hyperlinks** are used to interconnect Web pages and are mainly used for browsing. **Anchor texts** are text phrases within documents used to label hyperlinks and are very relevant to browsing.
- Web search** combines both aspects—browsing and retrieval—and is one of the main applications of information retrieval today. Web pages are analogous to documents. Web search engines maintain an indexed repository of Web pages, usually using the technique of inverted indexing. They retrieve the most relevant Web pages for the user in response to the user’s search request with a possible ranking in descending order of relevance. The **rank of a Webpage** in a retrieved set is the measure of its relevance to the query that generated the result set.

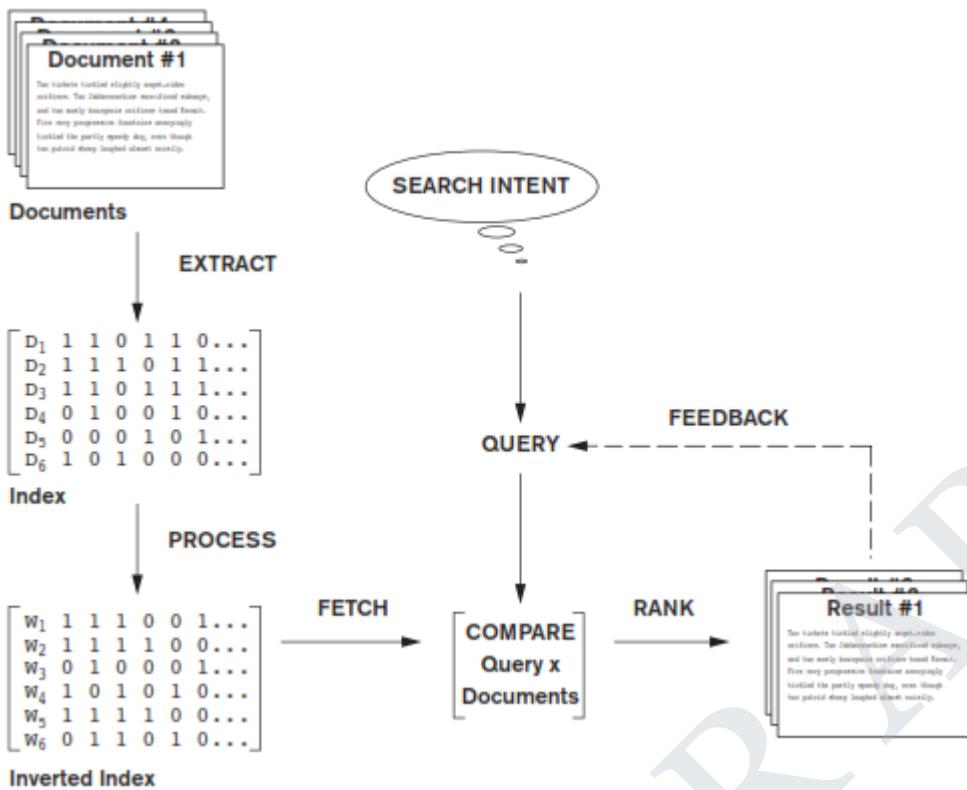
### Generic IR Pipeline

- Documents are made up of unstructured natural language text composed of character strings from English and other languages. Common examples of documents include newswire services, corporate manuals and reports, government notices, Web page articles, blogs, tweets, books, and journal papers. There are two main approaches to IR: statistical and semantic.
- In a **statistical approach**, documents are analyzed and broken down into chunks of text (words, phrases, or  $n$ -grams, which are all subsequences of length  $n$  characters in a text or document) and each word or phrase is counted, weighted, and measured for relevance or importance. These words and their properties are then compared with the query terms for potential degree of match to produce a ranked list of resulting documents that contain the words. Statistical approaches are further classified based on the method employed. The three main statistical approaches are Boolean, vector space, and probabilistic.
- Semantic approaches** to IR use knowledge-based techniques of retrieval that broadly rely on the syntactic, lexical, sentential, discourse-based, and pragmatic levels of knowledge understanding. In practice, semantic approaches also apply some form of statistical analysis to improve the retrieval process.
- Figure shows the various stages involved in an IR processing system. The steps shown on the left in Figure are typically offline processes, which prepare a set of documents for efficient retrieval; these are document preprocessing, document modeling, and indexing. The steps involved in query formation, query processing, searching mechanism, document retrieval, and relevance feedback are shown on the right in Figure. In each box, we highlight the important concepts and issues. The rest of this chapter describes some of the concepts involved in the various tasks within the IR process shown in Figure.



**Figure 27.1**  
Generic IR framework.

- Figure shows a simplified IR processing pipeline. In order to perform retrieval on documents, the documents are first represented in a form suitable for retrieval. The significant terms and their properties are extracted from the documents and are represented in a document index where the words/terms and their properties are stored in a matrix that contains these terms and the references to the documents that contain them.
- This index is then converted into an inverted index of a word/term vs. document matrix. Given the query words, the documents containing these words—and the document properties, such as date of creation, author, and type of document—are fetched from the inverted index and compared with the query. This comparison results in a ranked list shown to the user.
- The user can then provide feedback on the results that triggers implicit or explicit query expansion to fetch results that are more relevant for the user. Most IR systems allow for an interactive search where the query and the results are successively refined.



**Figure** Simplified IR process pipeline.

### **5.17. RETRIEVAL MODELS:**

Three main statistical models—Boolean, vector space, and probabilistic—and the semantic model.

#### **1) Boolean Model**

- In this model, documents are represented as a set of *terms*. Queries are formulated as a combination of terms using the standard Boolean logic set-theoretic operators such as AND, OR and NOT. Retrieval and relevance are considered as binary concepts in this model, so the retrieved elements are an “exact match” retrieval of relevant documents. There is no notion of ranking of resulting documents.
- All retrieved documents are considered equally important—a major simplification that does not consider frequencies of document terms or their proximity to other terms compared against the query terms.
- Boolean retrieval models lack sophisticated ranking algorithms and are among the earliest and simplest information retrieval models. These models make it easy to associate metadata information and write queries that match the contents of the documents as well as other properties of documents, such as date of creation, author, and type of document.

#### **2) Vector Space Model**

- The vector space model provides a framework in which term weighting, ranking of retrieved documents, and relevance feedback are possible. Documents are represented as *features* and *weights* of term features in an *n*-dimensional vector space of terms.

- **Features** are a subset of the terms in a *set of documents* that are deemed most relevant to an IR search for this particular set of documents. The process of selecting these important terms (features) and their properties as a sparse (limited) list out of the very large number of available terms (the vocabulary can contain hundreds of thousands of terms) is independent of the model specification. The query is also specified as a terms vector (vector of features), and this is compared to the document vectors for similarity/relevance assessment.
- The similarity assessment function that compares two vectors is not inherent to the model—different similarity functions can be used. However, the cosine of the angle between the query and document vector is a commonly used function for similarity assessment. As the angle between the vectors decreases, the cosine of the angle approaches one, meaning that the similarity of the query with a document vector increases. Terms (features) are weighted proportional to their frequency counts to reflect the importance of terms in the calculation of relevance measure. This is different from the Boolean model, which does not take into account the frequency of words in the document for relevance match.
- In the vector model, the *document term weight*  $w_{ij}$  (for term  $i$  in document  $j$ ) is represented based on some variation of the TF (term frequency) or TF-IDF (term frequency-inverse document frequency) scheme (as we will describe below). **TF-IDF** is a statistical weight measure that is used to evaluate the importance of a document word in a collection of documents. The following formula is typically used:

$$\text{cosine}(d_j, q) = \frac{\langle d_j \times q \rangle}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{|V|} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{|V|} w_{ij}^2} \times \sqrt{\sum_{i=1}^{|V|} w_{iq}^2}}$$

In the formula given above, we use the following symbols:

$d_j$  is the document vector.

$q$  is the query vector.

$w_{ij}$  is the weight of term  $i$  in document  $j$ .

$w_{iq}$  is the weight of term  $i$  in query vector  $q$ .

$|V|$  is the number of dimensions in the vector that is the total number of important keywords (or features).

TF-IDF uses the product of normalized frequency of a term  $i$  ( $TF_{ij}$ ) in document  $D_j$  and the inverse document frequency of the term  $i$  ( $IDF_i$ ) to weight a term in a document. The idea is that terms that capture the essence of a document occur frequently in the document (that is, their TF is high), but if such a term were to be a good term that discriminates the document from others, it must occur in only a few documents in the general population (that is, its IDF should be high as well).

IDF values can be easily computed for a fixed collection of documents. In case of Web search engines, taking a representative sample of documents approximates IDF computation. The following formulas can be used:

$$TF_{ij} = f_{ij} / \sum_{i=1 \text{ to } |V|} f_{ij}$$

$$IDF_i = \log(N/n_i)$$

In these formulas, the meaning of the symbols is:

$TF_{ij}$  is the normalized term frequency of term  $i$  in document  $D_j$ .

$f_{ij}$  is the number of occurrences of term  $i$  in document  $D_j$ .

$IDF_i$  is the inverse document frequency weight for term  $i$ .

$N$  is the number of documents in the collection.

$n_i$  is the number of documents in which term  $i$  occurs.

Note that if a term  $i$  occurs in all documents, then  $n_i = N$  and hence  $IDF_i = \log(1)$  becomes zero, nullifying its importance and creating a situation where division by zero can occur. The weight of term  $i$  in document  $j$ ,  $w_{ij}$  is computed based on its TF-IDF value in some techniques. To prevent division by zero, it is common to add a 1 to the denominator in the formulae such as the cosine formula above.

Sometimes, the relevance of the document with respect to a query ( $\text{rel}(D_j, Q)$ ) is directly measured as the sum of the TF-IDF values of the terms in the Query  $Q$ :

$$\text{rel}(D_j, Q) = \sum_{i \in Q} TF_{ij} \times IDF_i$$

The normalization factor (similar to the denominator of the cosine formula) is incorporated into the TF-IDF formula itself, thereby measuring relevance of a document to the query by the computation of the dot product of the query and document vectors.

The Rocchio algorithm is a well-known relevance feedback algorithm based on the vector space model that modifies the initial query vector and its weights in response to user-identified relevant documents. It expands the original query vector  $q$  to a new vector  $q_e$  as follows:

$$q_e = \alpha q + \frac{\beta}{|D_r|} \sum_{d_r \in D_r} d_r - \frac{\gamma}{|D_{ir}|} \sum_{d_{ir} \in D_{ir}} d_{ir},$$

Here,  $D_r$  and  $D_{ir}$  are relevant and non-relevant document sets and  $\alpha$ ,  $\beta$ , and  $\gamma$  are parameters of the equation. The values of these parameters determine how the feed-back affects the original query, and these may be determined after a number of trial-and-error experiments.

### 3) Probabilistic Model

The similarity measures in the vector space model are somewhat ad hoc. For example, the model assumes that those documents closer to the query in cosine space are more relevant to the query vector. In the probabilistic model, a more concrete and definitive approach is taken: ranking documents by their estimated probability of relevance with respect to the query and the document. This is the basis of the *Probability Ranking Principle* developed by Robertson:<sup>11</sup>

In the probabilistic framework, the IR system has to decide whether the documents belong to the **relevant set** or the **nonrelevant set** for a query. To make this decision, it is assumed that a predefined relevant set and nonrelevant set exist for the query, and the task is to calculate the probability that the document belongs to the relevant set and compare that with the probability that the document belongs to the nonrelevant set.

Given the document representation  $D$  of a document, estimating the relevance  $R$  and nonrelevance  $NR$  of that document involves computation of conditional probability  $P(R|D)$  and  $P(NR|D)$ . These conditional probabilities can be calculated using Bayes' Rule:

$$P(R|D) = P(D|R) \times P(R)/P(D)$$

$$P(NR|D) = P(D|NR) \times P(NR)/P(D)$$

A document  $D$  is classified as relevant if  $P(R|D) > P(NR|D)$ . Discarding the constant  $P(D)$ , this is equivalent to saying that a document is relevant if:

$$P(D|R) \times P(R) > P(D|NR) \times P(NR)$$

The likelihood ratio  $P(D|R)/P(D|NR)$  is used as a score to determine the likelihood of the document with representation  $D$  belonging to the relevant set.

The *term independence* or *Naïve Bayes* assumption is used to estimate  $P(D|R)$  using computation of  $P(t_i|R)$  for term  $t_i$ . The likelihood ratios  $P(D|R)/P(D|NR)$  of documents are used as a proxy for ranking based on the assumption that highly ranked documents will have a high likelihood of belonging to the relevant set.<sup>13</sup>

With some reasonable assumptions and estimates about the probabilistic model along with extensions for incorporating query term weights and document term weights in the model, a probabilistic ranking algorithm called **BM25** (Best Match 25) is quite popular. This weighting scheme has evolved from several versions of the **Okapi** system.

The Okapi weight for Document  $d_j$  and query  $q$  is computed by the formula below.

Additional notations are as follows:

$t_i$  is a term.

$f_{ij}$  is the raw frequency count of term  $t_i$  in document  $d_j$ .

$f_{iq}$  is the raw frequency count of term  $t_i$  in query  $q$ .

$N$  is the total number of documents in the collection.

$df_i$  is the number of documents that contain the term  $t_i$ .

$dl_j$  is the document length (in bytes) of  $d_j$ .

$avdl$  is the average document length of the collection.

The Okapi relevance score of a document  $d_j$  for a query  $q$  is given by the equation below, where  $k_1$  (between 1.0–2.0),  $b$  (usually 0.75), and  $k_2$  (between 1–1000) are parameters:

$$\text{okapi}(d_j, q) = \sum_{t_i \in q, d_j} \ln \frac{N - df_i + 0.5}{df_i + 0.5} \times \frac{(k_1 + 1)f_{ij}}{k_1 \left( 1 - b + b \frac{dl_j}{avdl} \right) + f_{ij}} \times \frac{(k_2 + 1)f_{iq}}{k_2 + f_{iq}},$$

#### 4) Semantic Model

However sophisticated the above statistical models become, they can miss many relevant documents because those models do not capture the complete meaning or information need conveyed by a user's query. In semantic models, the process of matching documents to a given query is based on concept level and semantic matching instead of index term (keyword) matching. This allows retrieval of relevant documents that share meaningful associations with other documents in the query result, even when these associations are not inherently observed or statistically captured.

Semantic approaches include different levels of analysis, such as morphological, syntactic, and semantic analysis, to retrieve documents more effectively. In **morphological analysis**, roots and affixes are analyzed to determine the parts of speech (nouns, verbs, adjectives, and so on) of the words. Following morphological analysis, **syntactic analysis** follows to parse and analyze complete phrases in documents. Finally, the semantic methods have to resolve word ambiguities and/or generate relevant synonyms based on the **semantic relationships** between levels of structural entities in documents (words, paragraphs, pages, or entire documents).

The development of a sophisticated semantic system requires complex knowledge bases of semantic information as well as retrieval heuristics. These systems often require techniques from artificial intelligence and expert systems. Knowledge bases like Cyc and WordNet have been developed for use in *knowledge-based IR systems* based on semantic models. The Cyc knowledge base, for example, is a representation of a vast quantity of commonsense knowledge about assertions (over 2.5 million facts and rules) interrelating more than 155,000 concepts for reasoning about the objects and events of everyday life. WordNet is an extensive thesaurus (over 115,000 concepts) that is very popular and is used by many systems and is under continuous development.

## **5.18. TYPES OF QUERIES IN IR SYSTEMS**

Different keywords are associated with the document set during the process of indexing. These keywords generally consist of words, phrases, and other characterizations of documents such as date created, author names, and type of document. They are used by an IR system to build an inverted index, which is then consulted during the search. The queries formulated by users are compared to the set of index keywords. Most IR systems also allow the use of Boolean and other operators to build a complex query. The query language with these operators enriches the expressiveness of a user's information need.

### **1) Keyword Queries:**

Keyword-based queries are the simplest and most commonly used forms of IR queries: the user just enters keyword combinations to retrieve documents. The query keyword terms are implicitly connected by a logical AND operator. A query such as 'database concepts' retrieves documents that contain both the words 'database' and 'concepts' at the top of the retrieved results. In addition, most systems also retrieve documents that contain only 'database' or only 'concepts' in their text. Some systems remove most commonly occurring words (such as *a, the, of*, and so on, called **stopwords**) as a preprocessing step before sending the filtered query keywords to the IR engine. Most IR systems do not pay attention to the ordering of these words in the query. All retrieval models provide support for keyword queries.

### **2) Boolean Queries:**

Some IR systems allow using the AND, OR, NOT, ( ), + , and – Boolean operators in combinations of keyword formulations. AND requires that both terms be found. OR lets either term be found. NOT means any record containing the second term will be excluded. '( )' means the Boolean operators can be nested using parentheses. '+' is equivalent to AND, requiring the term; the '+' should be placed directly in front of the search term. ‘–’ is equivalent to AND NOT and means to exclude the term; the ‘–’ should be placed directly in front of the search term not wanted. Complex Boolean queries can be built out of these operators and their combinations, and they are evaluated according to the classical rules of Boolean algebra. No ranking is possible, because a document either satisfies such a query (is “relevant”) or does not satisfy it (is “nonrelevant”). A document is retrieved for a Boolean query if the query is logically true as an exact match in the document. Users generally do not use combinations of these complex Boolean operators, and IR systems support a restricted version of these set operators. Boolean retrieval models can directly support different Boolean operator implementations for these kinds of queries.

### **3) Phrase Queries**

When documents are represented using an inverted keyword index for searching, the relative order of the terms in the document is lost. In order to perform exact phrase retrieval, these phrases should be encoded in the inverted index or implemented differently (with relative positions of word occurrences in documents). A phrase query consists of a sequence of words that makes up a phrase. The phrase is generally enclosed within double quotes. Each retrieved document must contain at least one instance of the exact phrase. Phrase searching is a more restricted and specific version of proximity searching. For example, a phrase searching query could be 'conceptual database design'. If phrases are indexed by the retrieval model, any retrieval model can be used for these query types. A phrase thesaurus may also be used in semantic models for fast dictionary searching for phrases.

### **4) Proximity Queries:**

Proximity search refers to a search that accounts for how close within a record multiple terms should be to each other. The most commonly used proximity search option is a phrase search that requires terms to be in the exact order. Other proximity operators can specify how close terms should be to each other. Some will also

specify the order of the search terms. Each search engine can define proximity operators differently, and the search engines use various operator names such as NEAR, ADJ(adjacent), or AFTER. In some cases, a sequence of single words is given, together with a maximum allowed distance between them. Vector space models that also maintain information about positions and offsets of tokens (words) have robust implementations for this query type. However, providing support for complex proximity operators becomes computationally expensive because it requires the time-consuming preprocessing of documents, and is thus suitable for smaller document collections rather than for the Web.

#### 5) Wildcard Queries:

Wildcard searching is generally meant to support regular expressions and pattern matching-based searching in text. In IR systems, certain kinds of wildcard search support may be implemented—usually words with any trailing characters (for example, ‘data\*’ would retrieve *data*, *database*, *datapoint*, *dataset*, and so on). Providing support for wildcard searches in IR systems involves preprocessing overhead and is not considered worth the cost by many Web search engines today. Retrieval models do not directly provide support for this query type.

#### 6) Natural Language Queries

There are a few natural language search engines that aim to understand the structure and meaning of queries written in natural language text, generally as a question or narrative. This is an active area of research that employs techniques like shallow semantic parsing of text, or query reformulations based on natural language understanding. The system tries to formulate answers for such queries from retrieved results. Some search systems are starting to provide natural language interfaces to provide answers to specific types of questions, such as definition and factoid questions, which ask for definitions of technical terms or common facts that can be retrieved from specialized databases. Such questions are usually easier to answer because there are strong linguistic patterns giving clues to specific types of sentences—for example, ‘defined as’ or ‘refers to’. Semantic models can provide support for this query type.

UNIT-1PART-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
- γ. 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
- iii) Embedded SQL

- 3) Assume the following table :

degree(degcode,name,subject)

candidate(seat no, degcode,semesrer,month,year,result)

Marks(seatno,degcode,semester,month,year,papcode,marks)

Degcode-degree code.Name-name of the degree(MSC.MCOM)

SUBJECT\_subject of courses Eg. Phy , pap code –paper code eg.A1

Solve the following queries using SQL

- i) Write a SELECT statement to display all the degree codes are there in the candidate table but not present in degree table in the order of degcode.
- ii) Write a SELECT statement to display the name of all the candidates who have got less than 40 marks in exactly 2 subjects .
- iii) Write a SELECT statement to display the name, subject and number of the candidate for all degrees in which there are less than 5 candidates.
- iv) Write a SELECT statement to display the names of all the candidates who have got highest total marks in MSc., (Maths).

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

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

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

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

- 8) consider the relation schema given in figure 1.design and draw an ER diagram that capture the information of this schema.

Employee(empno,name,office,age)

Books(isbn,tittle ,authors,publisher)

Loan(empno,isbn,date)

write the following queries in relational algebra and SQL.

- i)find the name of employees who have borrowed a book published by McGraw-Hill.

- ii) find the name of employees who have borrowed all books published by McGraw-Hill.

- 9) Write the DDL,DML,DCL commands for the students database.Which contains  
Student details :name,id,DOB,branch,DOJ.  
Course details:Course name,Course id,Stud.id,Faculty name,id,marks.
- 10) Differentiate between foreign key constraints and referential integrity constraints with suitable examples?  
11) Justify the need of embedded SQL .consider the relation student(studentno,name,mark and grade).Write embedded dynamic SQL statements in C language to retrieve all the student's records whose marks more than 90.  
12) Explain about functions and procedures in SQL.  
13) Discuss about triggers.

### **CS8492 / DATABASE MANAGEMENT SYSTEMS**

#### **UNIT-2**

#### **PART -A**

1. Why 4NF in Normal Form is more desirable than BCNF? Nov/ Dec 2014

4NF is an improvement over BCNF since it eliminates another form of undesirable structure MVD by taking two projections.

R.A  $\rightarrow$  R.B

A relation R is in fourth normal form if whenever there exists a MVD in relation R.A  $\rightarrow\!\!\!\rightarrow$  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  $\rightarrow$  Y, which specifies Y is functionally dependent on X.

3. State the anomalies of 1NF. Nov/ Dec 2015

INSERT. Certain student with SID 5 got admission in a different campus (say) Karachi cannot be added until the student registers for a course.

DELETE. If student graduates and his/her corresponding record is deleted, then all information about that student is lost.

UPDATE. If student migrates from Islamabad campus to Lahore campus (say) SID = 1, then six rows would have to be updated with this new information

4. Explain entity relationship model. May/ June 2016

The entity relationship model is a collection of basic objects called entities and relationship among those objects. An entity is a thing or object in the real world that is distinguishable from other objects.

5. What is a weak entity? Give example. Nov/ Dec 2016

An entity set may not have sufficient attributes to form a primary key, and its primary key compromises of its partial key and primary key of its parent entity, then it is said to be Weak Entity set

6. What are the desirable properties of decomposition? Nov/ Dec 2017

- a. Lossless Join: this property ensures that any instance of the original relation can be identified from corresponding instances in the smaller relation.
- b. Dependency Preservation: this property ensures that a constraint on the original relation can be maintained by simply enforcing some constraint on each of the smaller relation.

7. What is meant by normalization of data?

— It is a process of analyzing the given relation schemas based on their Functional Dependencies (FDs) and primary key to achieve the properties

- i) Minimizing redundancy
- ii) Minimizing insertion, deletion and updating anomalies

## 8. Distinguish key and super key. Nov Dec 2017

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

## 9. Define instance and schema?

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

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

## 10. Define the terms 1) Physical schema 2) logical schema.

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

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

## 11. What are attributes? Give examples.

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

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

## 12. What is relationship? Give examples

A relationship is an association among several entities.

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

## 13. . Define the term Relationship set.

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

## 14. Define null values

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

15. Consider the following relation :

R (A, B, C, D, E)

The primary key of the relation is AB. The following functional dependencies hold :

$A \rightarrow C$

$B \rightarrow D$

$AB \rightarrow E$ .

Is the above relation in second normal form?

16. Consider the following relation :

R(A, B, C, D)

The primary key of the relation is A. The following functional dependencies hold :

$A \rightarrow BC$

$B \rightarrow D$

Is the above relation in third normal form?

17. What is the need for normalization?

Normalization is the process of removing redundant data from your tables in order to improve storage efficiency, data integrity and scalability. This improvement is balanced against an increase in complexity and potential performance losses from the joining of the normalized tables at query-time. There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a table). Both of these are worthy goals as they reduce the amount of space a [database](#) consumes and ensure that data is logically stored.

18. Give an example of a relation schema R and a set of dependencies such that R is in BCNF, but not in 4NF.

19. Why are certain functional dependencies called as trivial functional dependencies?

20. A relation  $R=\{A,B,C,D\}$  has FD's  $F=\{ A \rightarrow D , D \rightarrow C , C \rightarrow AB \}$  is R is in 3NF?

## Part-B

1. Distinguish between lossless-join decomposition and dependency preserving decomposition.
2. Discuss the correspondence between the ER model construct and the relational model constructs. Show how each ER model construct can be mapped to the relation model. discuss the option for mapping EER model construct.
3. Consider the relation schema given in figure. Design and draw an ER diagram that capture the information of this schema.

Employee(empno,name,office,age)

Books(isbn,title ,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, date 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<sup>st</sup>, 2<sup>nd</sup>, and 3rd, BCNF, 4<sup>th</sup>, 5<sup>th</sup> and domain-key) with suitable examples. **May/ June 2015**
8. Explain non loss decomposition and functional dependencies with suitable example.
9. Explain the boyce-code normal form with an example. Also state how it differs from that of 3NF.
10. Discuss join dependencies and fifth normal form, and explain why 5NF?
11. Construct an ER diagram for car insurance company that has a set of customers each of whom owns one or more cars. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars, and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the premium was made. **(7) Nov/ Dec 2016**
12. Describe about the multi\_valued dependencies and fourth normal form with suitable example.
13. Explain boyce codd normal form and fourth norml forms with suitable example.
14. Explain the principles of
  - (i) Loss less join decomposition
  - (ii) Join dependencies
  - (iii) Fifth normal form.

### CS8492 / DATABASE MANAGEMENT SYSTEMS UNIT-III

#### PART-A

##### **1. What is meant by concurrency control? Nov/ Dec 2015**

In a multiprogramming environment, multiple transactions can be executed simultaneously. The system must control the interaction among the concurrent transactions. This control is achieved through one of concurrency control schemes. The concurrency control schemes are based on the serializability property.

##### **2. Give an example of two phase commit protocol. Nov/ Dec 2015**

```
lock-X(A); // Growing Phase
read(A);
A=A-50;
Write(A);
lock-X(B); // Growing Phase
read(B);
B=B+50;
Wtite(B);
unlock(A); // Shrinking Phase
unlock(B); // Shrinking Phase
```

##### **3. What are the properties of transaction? April/ May 2016, Nov/ Dec 2014, April/ May 2015** Collection of operations that form a single logical unit of work are called transaction.

##### ACID Properties

###### **1) Atomicity.**

Either all operations of the transaction are properly reflected in the database or none are.

###### **2) Consistency.**

Execution of a transaction in isolation preserves the consistency of the database.

###### **3) Isolation.**

Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

#### **4) Durability.**

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

#### **4. Differentiate Strict Two Phase Locking Protocol and Rigorous Two Phase Locking Protocol. April/ May 2016**

##### **Strict Two Phase Locking Protocol**

All exclusive locks are held by transaction T and are released when T commits and not before.

##### **Rigorous Two Phase Locking Protocol**

All locks both exclusive and shared locks held by transaction T are released when T commits and not before.

#### **5. What is Serializability (Serializable Schedule)? How is it tested? Nov/ Dec 2014, Nov/ Dec 2016, April/ May 2017**

Serializability is a schedule that has the same effect on the database as a serial schedule. It is tested using two techniques: View Serializability and Conflict Serializability.

#### **6. List the four conditions for deadlock. Nov/ Dec 2016**

- i. mutual exclusion
- ii. hold and wait
- iii. no pre-emption
- iv. circular wait

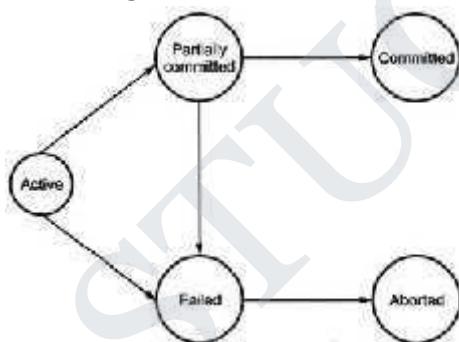
#### **7. What type of locking is needed for insert and delete operations? April/ May 2017**

Shared lock is obtained when Data item can only be read. But data item can be both read as well as written when exclusive lock is obtained. So exclusive lock is needed for insert and delete operations

#### **8. Brief about cascading rollback.**

The Phenomenon in which a single transaction failure leads to a series of transaction roll backs, is called cascading rollback.

#### **9. With neat diagram show the states of a transaction.**



#### **10. What is transaction-management component?**

Ensuring atomicity is the responsibility of the database system itself specifically; it is handled by a component called the transaction-management component.

#### **11. When two operations in schedule are said to be conflict?**

- i) Two operation belong to different transaction
- ii) Two operation access the same item x
- iii) At least one of the operation is write-item(x)

#### **12. Define cascading rollback**

A **cascading rollback** occurs in database systems when a transaction (T1) causes a failure and a **rollback** must be performed. Other transactions dependent on T1's actions must also be rolled back 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 started.

**16. Define timestamp based protocol.**

Timestamp based protocol ensures serializability. It selects an ordering among transactions in advance using time stamps.

**17. If deadlock is avoided by deadlock avoidance scheme, is starvation still possible? Explain your answer.**

A transaction may become the victim of deadlock-prevention rollback arbitrarily many times, thus creating a potential starvation situation.

**18. When are two scheduled conflict equivalent?**

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent.

**19. Why is it necessary to have control of concurrent execution of transactions? How is it made possible?**

The system must control the interaction among the concurrent transactions. This control is achieved through one of the concurrency control schemes. The concurrency control are based on the serializability property.

**20. What are the two operations that access data in transaction?**

Read(x)- transfer data item x from database.

Write(x)- transfer data item x from the local buffer.

21. List the two commonly used Concurrency Control techniques.

1) Lock based protocols

2) Time stamp based protocol

22. What are the pitfalls of lock-based protocols?

23. List the SQL statements used for transaction control.

24. List down the SQL facilities for concurrency.

**Part B**

1. What is concurrency? Explain it in terms of locking mechanisms and two phase commit protocol. **Now/ Dec 2014**
2. Write short notes on Transaction concept and schedules. **Now/ Dec 2014**
3. Write short notes on Deadlock. **Now/ Dec 2014, (16) Now/ Dec 2015, (6) Now/ Dec 2016**
4. What is concurrency control? How is it implemented in DBMS? Illustrate with a suitable example. **Now/ Dec 2015**
5. Discuss View Serializability and conflict Serializability. **Now/ Dec 2015**
6. Explain briefly about two phase commit. **May/ June 2016, (8) May/ June 2015, (6) Now/ Dec 2016**
7. Explain about locking protocols. **May/ June 2016**
8. Consider the following schedules. The actions are listed in the order they are scheduled and prefixed with the transaction name. **May/ June 2015**

S1: T1: R(X), T2: R(X), T1: W(Y), T2: W(Y), T1: R(Y), T2: R(Y)

S2: T3: W(X), T1: R(X), T1: W(Y), T2: R(X), T2: W(Z), T3: R(Z) For each of the schedules answer the following questions:

- (i) What is the precedence graph for the schedule?  
(ii) Is the schedule conflict serializable? If so, what are all the conflict equivalent serial schedule?  
(iii) Is the schedule view serializable? If so, what are all the view equivalent serial schedule?
9. Discuss the violations caused by each of the following: Dirty read, non-repeatable read and phantoms with suitable example. *April/ May 2017*
10. Consider the following two transactions:
- T1: read(A);  
Read(B);  
If A=0 then B=B+1;  
Write(B);
- T2: read(B);  
Read(A);  
If B=0 then A=A+1;  
Write(A);
- Add lock and unlock instructions to transactions T1 and T2 so that they observe the two phase locking protocol.  
Can the execution of these transactions result in a deadlock? *Nov/ Dec 2016*
11. Consider the following extension to the tree locking protocol, which allows both shared and exclusive locks:
- A transaction can be either a read only transaction in which case it can request only shared locks or an update transaction, in which case it can request only exclusive locks.
  - Each transaction must follow the rules of the tree protocol. Read-only transactions may lock any data item first, whereas update transaction must lock the root first.
- Show that the protocol ensures serializability and deadlock freedom *Nov/ Dec 2016*
12. Elaborate the SQL facilities for concurrency and recovery.  
13. Discuss about transaction recovery.  
14. Explain various isolation levels in concurrency.

**CS8492 / DATABASE MANAGEMENT SYSTEMS**  
**Unit-4**

**Part-A****1. Differentiate Static and Dynamic Hashing. Nov/ Dec 2014, May/ June 2015, Nov/ Dec 2015**

S.No	Static Hashing	Dynamic Hashing
1	Number of buckets is fixed	Number of buckets is not fixed
2	Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows. If database shrinks, again space will be wasted	Good for database that grows and shrinks in size
3	One option is periodic re-organization of the file with a new hash function, but it is very expensive	Allows the hash function to be modified dynamically

**2. Give an example of a join that is not a simple equi join for which partitioned parallelism can be used. Nov/ Dec 2015**

$r, s \rightarrow$  relations

$A, B, C \rightarrow$  attributes

$(r.A=s.B) \wedge (r.a < s.c)$

Here we have extra conditions which can be checked after the join. Hence partitioned parallelism is used.

**3. What is meant by garbage collection? May/ June 2016**

It is a process of destroying objects that are no longer referenced and freeing the resources those objects used.

**4. Define software and hardware RAID system. May/ June 2016**

RAID has an arrangement of several independent disks that are organized to improve reliability through parity schemes and error correcting codes and at the same time increase performance through data striping.

**5. List out the mechanisms to avoid collisions during hashing. Nov/ Dec 2016****a. Open Hashing (Separate chaining)**

Open Hashing, is a technique in which the data is not directly stored at the hash key index ( $k$ ) of the Hash table. Rather the data at the key index ( $k$ ) in the hash table is a pointer to the head of the data structure where the data is actually stored.

**b. Closed hashing (open Addressing)**

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found.

**6. What are the disadvantages of B tree over B+ tree? Nov/ Dec 2016**

- 1) In B+-Trees, all records are stored at the leaf level while keys and child pointers are stored in interior nodes (non-leaf nodes). This maximizes the branching factor of the internal nodes.
- 2) Because B+ trees don't have data associated with interior nodes, more keys can fit on a page of memory. Therefore, it will require fewer cache misses in order to access data that is on a leaf node.
- 3) The leaf nodes of B+ trees are linked, so doing a full scan of all objects in a tree requires just one linear pass through all the leaf nodes. A B tree, on the other hand, would require a traversal of every level in the tree. This full-tree traversal will likely involve more cache misses than the linear traversal of B+ leaves.

**7. Define replication transparency. May/ June 2017**

A **transparency** is some aspect of the **distributed system** that is hidden from the user. A **transparency** is provided by including some set of mechanisms in the **distributed system** at a layer below the interface where the **transparency** is required. **Replication transparency** is the term used to describe the fact that the user should be unaware that data is **replicated**.

on.

**8. What are two approaches to store a relation in the distributed database? May '04**

- a. Replications ii. Fragmentation

**9. What are the advantages of distributed database? Dec'04, May'08**

- i. Sharing data
- ii. Autonomy
- iii. Availability

**10. List out the reasons for the development of distributed database. May'06**

- i. In the centralized system, data is stored on a single computer. If that computer fails, complete system fails.
- ii. In client-server system also the data is stored on server. If server fails, the complete system fails.
- iii. Thus, there was need of a system in which data is distributed and replicated among several computers.

**11. Give the measures of quality of a disk.**

1. Capacity
2. Access time
3. Seek time
4. Data transfer rate
5. Reliability
6. Rotational latency time

**12. What are the types of storage devices?**

Primary storage, Secondary storage, Tertiary storage

**13. Define access time and seek time.**

**Access time** is the time from when a read or write request is issued to when data transfer begins. The time for repositioning the arm is called the seek time and it increases with the distance that the arm is called the **seek time**.

**14. Define rotational latency time.**

The time spent waiting for the sector to be accessed to appear under the head is called the rotational latency time.

**15. What is the use of a slotted-page structure and what is the information present in the header?**

The slotted-page structure is used for organizing records within a single block.

The header contains the following information.

- i. The number of record entries in the header.
- ii. The end of free space.

An array whose entries contain the location and size of each record.

**16. What are the advantages of distributed system?**

- 1) Sharing data
- 2) Autonomy
- 3) Availability

17. Which are the factors to be considered for the evaluation of indexing and hashing techniques?

18. What are the advantages and disadvantages of indexed sequential file?

**Part-A**

1. What is a B+ tree index file in DBMS?
  2. Examine the need for query Optimization.
  3. Explain —Query Optimization with your own database.
  4. Point out the methods for implementing JOINs.
  5. Define software and hardware RAID systems.
  6. Illustrate the need for RAID.
  7. Distinguish between fixed length records and variable length records?
  8. When is it preferable to use a dense index rather than a sparse index? Explain your answer.
  9. List the different Hashing techniques.
  10. Give the procedure to reduce the occurrences of bucket overflows in a hash file organization?
  11. What are ordered indices with example?
  12. Contrast sparse index and dense index
  13. Outline the steps involved in query processing.
  14. Point out the disadvantages of B Tree over B+ Tree
  15. Differentiate between Static and Dynamic Hashing
  16. List out the mechanisms to avoid collision during hashing.
  17. What are select operations?
  18. Assess why we need to go for cost estimation in query optimization.
  19. What is hash function? Give example.
20. Prepare the factors to be considered for the evaluation of indexing and hashing techniques?

**PART-B**

1. Explain about RAID system. How does it improve performance and reliability.
2. Discuss the level 3 and level 4 of RAID.
- 3.(i) Describe the index schemas used in databases.(07)  
(ii) Since indices speed query processing, why might they not be kept on several search keys? List as many reasons as possible.(06)
4. Describe the different types of file organization? Explain using a sketch of each of them with their advantages and disadvantages.(13)

- 5.(i) Describe the ordered indices with example.(10)  
(ii) Describe the different methods of implementing variable length records. (03)

6. Give a detailed description about Query Processing and Optimization. Explain the cost estimation of Query Optimization.

- 7.(i) Discuss briefly about B+ tree index file with example. (07)  
(ii) How does a B-tree differ from a B+ - tree? why is a B+-tree usually preferred as an access structure to a data file?(06)

8. (i) Illustrate indexing techniques with suitable examples (07)

- (ii) Write notes on Hashing.(06)

9. Illustrate the Join order optimization and Heuristic optimization algorithms.(13)

10. What is meant by semantic query optimization? How does it differ from other query optimization technique? Give example. (13)

11. Examine the algorithms for SELECT and JOIN

12. Examine the catalog information for cost estimation for selection and sorting operation in database. (13)

13. Describe about B tree index file with example.(13)

14. Explain the distinction between static and dynamic hashing. Discuss the relative merits of each technique in database applications. (13)

15. Develop a B+ tree to insert the following

5,3,4,9,7,15,14,21,22,23. (13)

16. Construct B tree and B<sup>+</sup> tree to insert the following key values(the order of the tree is three) 32,11,15,13,7,22,15,44,67,4.( 15)

17. The following key values are organized in an extendable hashing technique.  
1 3 5 8 9 12 17 28 Show the extendable hash structure for this file if the hash function is  $h(x)=x \bmod 8$  and buckets can hold three records.

**Unit-5**  
**Part-A**

**1. What is crawling and indexing the Web? Nov/ Dec 2014**

**Web crawling** is the process by which we gather pages from the web, in order to index them and support a search engine. The crawler begins with one or more URLs that constitute a seed set. It picks a URL from this seed set, then fetches the web page at that URL. **Indexing** is the process of adding webpages into the search

**2. What is Relevance Ranking? Nov/ Dec 2014**

**Relevancy ranking** is the process of sorting the document results so that those documents which are most likely to be relevant to your query are shown at the top.

**3. List the types of privileges used in database access control. Nov/ Dec 2015**

- System privileges:** these privileges are related to the access of the database
- Object privileges:** these privileges are focused on a particular database object

**4. Define a distributed database management System. Nov/ Dec 2016**

A DDBMS (distributed database management system) is a centralized application that manages a distributed database as if it were all stored on the same computer. The DDBMS synchronizes all the data periodically, and in cases where multiple users must access the same data, ensures that updates and deletes performed on the data at one location will be automatically reflected in the data stored elsewhere.

**5. How does the concept of an object in the object oriented model differ from the concept of an entity in the entity relationship model? Nov/ Dec 2016**

An **entity** is something that exists in itself, actually or potentially, concretely or abstractly, physically or not. It needs not be of material existence.

In Object Oriented Model, an **object** is a location in memory having a value and possibly referenced by an identifier.

An entity is an abstract concept that's typically represented by a table in a database schema. The term object usually refers to in-memory data structures.

A database object is represented as database, schema, table, column, primary key, and foreign key while a database Entity is a concept or object of importance about which data must be captured.

**6. Explain the three phase commit protocol.**

It is an extension of two phase commit protocol for avoiding blocking problem of 2PC protocol under certain assumptions:

1. No network partition occurs in the system
2. Not more than k sites fail

**7. Write the difference between XML Schema and XML DTD**

- i. XML schemas are written in XML while DTD are derived from SGML syntax.
- ii. XML schemas define data types for elements and attributes while DTD doesn't support data types.
- iii. XML schemas allow support for namespaces while DTD does not.
- iv. XML schemas define number and order of child elements, while DTD does not.
- v. XML schemas can be manipulated on your own with XML DOM but it is not possible in case of DTD.
- vi. Using XML schema user need not to learn a new language but working with DTD is difficult for a user.

vii. XML schema provides secure data communication i.e sender can describe the data in a way that receiver will understand, but in case of DTD data can be misunderstood by the receiver.

viii. XML schemas are extensible while DTD is not extensible.

**Part-A**

1. Compare information retrieval Vs DBMS.
2. Give the architecture models in distributed database.
3. Show how are transaction performed in Object oriented database?
4. List the Operations performed in transaction?
5. Define Information Retrieval system. Prepare how it is differs from database system.
6. Define distributed database management system.
7. Demonstrate the meaning of homogenous and heterogeneous DDBMS
8. What are ODL and OQL.
9. List out the IR models.
10. Tell how spatial databases are more helpful than active database?
11. Differentiate XML schema and DTD.
12. Discuss Relevance Ranking.
13. State the function of XML schema.
14. List the features of object relational.
15. How does the concept of an object in the object oriented model differ from the concept of an entity in the entity relationship model?
16. Can we have more than one constructor in a class? If yes, explain the need for such a situation.
17. List Explain the need of object oriented database.
18. Give the general syntax of XML file.

**PART-B**

1. Explain the structure of XML databases. (13)
2. Describe the important models of information retrieval. (13)

3.

Describe about Distributed Databases and their characteristics,

advantages and disadvantages.(13)

4.Explain the necessary characteristics a system must satisfy to be considered as an object oriented database management system. (13)

5.Differentiate between Document Type Definition and XML schema. (13)

6.i) Discuss about Distributed Transactions (07)

ii) Show the challenges in object relational database.(6)

7.i) Compare and contrast between object oriented and XML databases. (7)

ii) Give XML representation of bank management system and also explain about DTD and XML schema (6)

8.Discuss briefly about object database concepts. (13)

9.Illustrate the concepts for information retrieval. (13)

10.Illustrate the hierarchical data model in XML. (13)

11.Point out the types of queries in IR systems. (13)

12.Describe in detail about Object Model of ODMG. (13)

13.i) Explain the features of object relational. (7)

ii) Examine the process of Querying XML data (6)

14. Give the DTD or XML Schema for an XML representation of the following nested-relational schema : (15)

Emp=(ename,ChildrenSet setoff(Children),SkillsSet setoff (Skills))

Children=(name,Birthday)

Birthday=(day,month,year)

Skills=(type,ExamsSet setoff(Exams))

Exams=(year,city)

Explain with diagrammatic illustration the architecture of a distributed database management system. (15)

# CS8492 Database Management Systems

## UNIT I

### RELATIONAL DATABASES

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

Q. No.	Questions	CO	Blo m's Level
1.	<p><b>Define database management system and its applications.</b> <u>Nov/Dec 2008, 2014</u>          Database management system (DBMS) is a collection of interrelated data and a set of programs to access those data.          Applications:</p> <ul style="list-style-type: none"> <li>▪ Banking</li> <li>▪ Airlines</li> <li>▪ Universities</li> <li>▪ Credit card transactions</li> <li>▪ Tele communication</li> <li>▪ Finance</li> <li>▪ Sales</li> <li>▪ Manufacturing</li> <li>▪ Human resources</li> </ul>	C212.1	BTL1
2.	<p><b>What are the advantages of using a DBMS?</b> <u>May / Dec 2007</u> <b>What is the purpose of database management system ?</b> <u>Nov/Dec 2014</u></p> <ol style="list-style-type: none"> <li>a. The advantages of using a DBMS are</li> <li>a) Controlling redundancy</li> <li>b) Restricting unauthorized access</li> <li>c) Providing multiple user interfaces</li> <li>d) Enforcing integrity constraints.</li> <li>e) Providing backup and recovery</li> </ol>	C212.1	BTL 1
3.	<p><b>Compare database systems with file systems.</b> <u>NOV/DEC 2006. Apr/ May 2015.</u>  <b>What are the disadvantages of file processing system?</b> <u>May/ June 2016</u></p> <ol style="list-style-type: none"> <li>a. Data redundancy and inconsistency</li> <li>b. Difficulty in accessing data</li> <li>c. Atomicity of updates</li> <li>d. Concurrent access by multiple users</li> <li>e. Security problems</li> </ol>	C212.1	BTL 4

4.	<b>List the features of a database.</b> <ul style="list-style-type: none"> <li>• It is a persistent (stored) collection of related data.</li> <li>• The data is input (stored) only once.</li> <li>• The data is organized (in some fashion).</li> <li>• The data is accessible and can be queried (effectively and efficiently).</li> </ul>	C212.1	BTL 1
5.	<b>Define a database</b> Specifying the data types, structures, and constraints of the data to be stored using a <i>Data Definition Language</i>	C212.1	BTL 1
6.	<b>Define a data model.</b> A data model is a collection of concepts that can be used to describe the structure of a database. The model provides the necessary means to achieve the abstraction.	C212.1	BTL 1
7.	<b>What are the categories of data models.</b> <b>High level/conceptual data models</b> –provide concepts close to the way users perceive the data. <b>Physical data models</b> –provide concepts that describe the details of how data is stored in the computer. These concepts are generally meant for the specialist, and not the end user. <b>Representational data models</b> –provide concepts that may be understood by the end user but not far removed from the way data is organized.	C212.1	BTL 5
8.	<b>Define high level/conceptual data model.</b> <b>Entity</b> –represents a real world object or concept <b>Attribute</b> - represents property of interest that describes an entity, such as name or salary. <b>Relationships</b> –among two or more entities, represents an association among two or more entities.	C212.1	BTL 1
9.	<b>Define representational data models.</b> <b>Representational data models</b> are used most frequently in commercial DBMSs. They include relational data models, and legacy models such as network and hierarchical models.	C212.1	BTL 1
10.	<b>Define physical data model.</b> <b>Physical data models</b> describe how data is stored in files by representing record formats, record orderings and access paths.	C212.1	BTL1
11.	<b>What is object data model.</b> <b>Object data models</b> –a group of higher level implementation data models closer to conceptual data models	C212.1	BTL 2
12.	<b>What is internal level schema.</b> <b>Object data models</b> –a group of higher level implementation data models closer to conceptual data models	C212.1	BTL 2
13.	<b>What is the conceptual level schema.</b> <b>The conceptual level</b> –has a conceptual schema which describes the structure of the database for users. It hides the details of the physical storage structures, and	C212.1	BTL 2

	concentrates on describing entities, data types, relationships, user operations and constraints. Usually a representational data model is used to describe the conceptual schema.		
14.	<p><b>What is an external or view level schema.</b></p> <p><b>The External or View level</b> –includes external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. Represented using the representational data model.</p>	C212.1	BTL 2
15.	<p><b>List the components of DBMS.</b></p> <p>The major components of database management system are:</p> <ul style="list-style-type: none"> <li>• Software</li> <li>• Hardware</li> <li>• Data</li> <li>• Procedures</li> <li>• Database Access Language</li> <li>• Users</li> </ul>	C212.1	BTL 1
16.	<p><b>What is relational model.</b></p> <p>The relational model represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.</p>	C212.1	BTL 2
17.	<p><b>List some of the relational model concepts.</b></p> <ol style="list-style-type: none"> <li>1. <b>Attribute:</b> Each column in a Table. Attributes are the properties which define a relation. e.g., Student_Rollno, NAME, etc.</li> <li>2. <b>Tables</b> – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.</li> <li>3. <b>Tuple</b> – It is nothing but a single row of a table, which contains a single record.</li> <li>4. <b>Relation Schema:</b> A relation schema represents the name of the relation with its attributes.</li> <li>5. <b>Degree:</b> The total number of attributes which in the relation is called the degree of the relation.</li> <li>6. <b>Cardinality:</b> Total number of rows present in the Table.</li> <li>7. <b>Column:</b> The column represents the set of values for a specific attribute.</li> <li>8. <b>Relation instance</b> – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.</li> <li>9. <b>Relation key</b> - Every row has one, two or multiple attributes, which is called relation key.</li> <li>10. <b>Attribute domain</b> – Every attribute has some pre-defined value and scope which is known as attribute domain</li> </ol>	C212.1	BTL 1
18.	<p><b>List some relational integrity constraints.</b></p> <ol style="list-style-type: none"> <li>1. Domain Constraints</li> <li>2. Key constraints</li> <li>3. Referential integrity constraints</li> </ol>	C212.1	BTL1
19.	<p><b>Define domain constraints.</b></p> <p>Domain constraints can be violated if an attribute value is not appearing in the corresponding domain or it is not of the appropriate data type.</p>	C212.1	BTL2

	<p>Domain constraints specify that within each tuple, and the value of each attribute must be unique. This is specified as data types which include standard data types integers, real numbers, characters, Booleans, variable length strings, etc.</p> <p><b>Example:</b></p> <pre>Create DOMAIN CustomerName CHECK (value not NULL)</pre> <p>The example shown demonstrates creating a domain constraint such that CustomerName is not NULL</p>																										
20.	<p><b>Define key constraints.</b></p> <p>An attribute that can uniquely identify a tuple in a relation is called the key of the table. The value of the attribute for different tuples in the relation has to be unique.</p> <p><b>Example:</b></p> <p>In the given table, CustomerID is a key attribute of Customer Table. It is most likely to have a single key for one customer, CustomerID =1 is only for the CustomerName =" Google".</p> <table border="1"> <thead> <tr> <th>CustomerID</th> <th>CustomerName</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Google</td> </tr> <tr> <td>2</td> <td>Amazon</td> </tr> <tr> <td>3</td> <td>Apple</td> </tr> </tbody> </table>	CustomerID	CustomerName	1	Google	2	Amazon	3	Apple	C212.1	BTL2																
CustomerID	CustomerName																										
1	Google																										
2	Amazon																										
3	Apple																										
21.	<p><b>Define referential integrity constraints.</b></p> <p>Referential integrity constraints is base on the concept of Foreign Keys. A foreign key is an important attribute of a relation which should be referred to in other relationships. Referential integrity constraint state happens where relation refers to a key attribute of a different or same relation. However, that key element must exist in the table.</p> <p><b>Example:</b></p> <table border="1"> <caption>Customer</caption> <thead> <tr> <th>CustomerID</th> <th>CustomerName</th> <th>Status</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Google</td> <td>Active</td> </tr> <tr> <td>2</td> <td>Amazon</td> <td>Active</td> </tr> <tr> <td>3</td> <td>Apple</td> <td>Inactive</td> </tr> </tbody> </table> <table border="1"> <caption>Billing</caption> <thead> <tr> <th>InvoiceNo</th> <th>CustomerID</th> <th>Amount</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>\$100</td> </tr> <tr> <td>2</td> <td>1</td> <td>\$200</td> </tr> <tr> <td>3</td> <td>2</td> <td>\$150</td> </tr> </tbody> </table> <p>In the above example, we have 2 relations, Customer and Billing.</p> <p>Tuple for CustomerID =1 is referenced twice in the relation Billing. So we know CustomerName=Google has billing amount \$300</p>	CustomerID	CustomerName	Status	1	Google	Active	2	Amazon	Active	3	Apple	Inactive	InvoiceNo	CustomerID	Amount	1	1	\$100	2	1	\$200	3	2	\$150	C212.1	BTL3
CustomerID	CustomerName	Status																									
1	Google	Active																									
2	Amazon	Active																									
3	Apple	Inactive																									
InvoiceNo	CustomerID	Amount																									
1	1	\$100																									
2	1	\$200																									
3	2	\$150																									
22.	<p><b>List the operations which can be done on the relational model.</b></p> <p>The operations are, Insert, update, delete and select.</p>	C212.1	BTL1																								

	<ul style="list-style-type: none"> <li>Insert is used to insert data into the relation</li> <li>Delete is used to delete tuples from the table.</li> <li>Modify allows you to change the values of some attributes in existing tuples.</li> <li>Select allows you to choose a specific range of data.</li> </ul>				
23.	<p><b>What are the advantages of relational model.</b></p> <ul style="list-style-type: none"> <li><b>Simplicity:</b> A relational data model is simpler than the hierarchical and network model.</li> <li><b>Structural Independence:</b> The relational database is only concerned with data and not with a structure. This can improve the performance of the model.</li> <li><b>Easy to use:</b> The relational model is easy as tables consisting of rows and columns is quite natural and simple to understand</li> <li><b>Query capability:</b> It makes possible for a high-level query language like SQL to avoid complex database navigation.</li> <li><b>Data independence:</b> The structure of a database can be changed without having to change any application.</li> <li><b>Scalable:</b> Regarding a number of records, or rows, and the number of fields, a database should be enlarged to enhance its usability.</li> </ul>	C212.1	BTL1		
24.	<p><b>What are the disadvantages of relational model.</b></p> <ul style="list-style-type: none"> <li>Few relational databases have limits on field lengths which can't be exceeded.</li> <li>Relational databases can sometimes become complex as the amount of data grows, and the relations between pieces of data become more complicated.</li> <li>Complex relational database systems may lead to isolated databases where the information cannot be shared from one system to another.</li> </ul>	C212.1	BTL1		
25.	<p><b>Define relational algebra.</b></p> <ul style="list-style-type: none"> <li>Intermediate language used within DBMS.</li> <li>Procedural language</li> </ul>	C212.1	BTL1		
26.	<p><b>List the relational algebraic operations.</b></p> <ul style="list-style-type: none"> <li><i>Basic operators:</i> select, project, union, set difference, Cartesian product</li> <li><i>Derived operators:</i> set intersection, division, join</li> </ul>	C212.1	BTL1		
27.	<p><b>Define select operation in relational algebra.</b> Produce table containing subset of rows of argument table satisfying condition  <math>\sigma_{\text{condition}} \text{ relation}</math></p> <p>Example:</p> <table style="width: 100%; text-align: center;"> <tr> <td style="width: 40%;">Person</td> <td style="width: 60%;"><math>\sigma_{\text{Hobby}=\text{'stamps'}}(\text{Person})</math></td> </tr> </table>	Person	$\sigma_{\text{Hobby}=\text{'stamps'}}(\text{Person})$	C212.1	BTL1
Person	$\sigma_{\text{Hobby}=\text{'stamps'}}(\text{Person})$				

	<table border="1"> <thead> <tr> <th><i>Id</i></th><th><i>Name</i></th><th><i>Address</i></th><th><i>Hobby</i></th></tr> </thead> <tbody> <tr> <td>1123</td><td>John</td><td>123 Main</td><td>stamps</td></tr> <tr> <td>1123</td><td>John</td><td>123 Main</td><td>coins</td></tr> <tr> <td>5556</td><td>Mary</td><td>7 Lake Dr</td><td>hiking</td></tr> <tr> <td>9876</td><td>Bart</td><td>5 Pine St</td><td>stamps</td></tr> </tbody> </table>	<i>Id</i>	<i>Name</i>	<i>Address</i>	<i>Hobby</i>	1123	John	123 Main	stamps	1123	John	123 Main	coins	5556	Mary	7 Lake Dr	hiking	9876	Bart	5 Pine St	stamps	<table border="1"> <thead> <tr> <th><i>Id</i></th><th><i>Name</i></th><th><i>Address</i></th><th><i>Hobby</i></th></tr> </thead> <tbody> <tr> <td>1123</td><td>John</td><td>123 Main</td><td>stamps</td></tr> <tr> <td>9876</td><td>Bart</td><td>5 Pine St</td><td>stamps</td></tr> </tbody> </table>	<i>Id</i>	<i>Name</i>	<i>Address</i>	<i>Hobby</i>	1123	John	123 Main	stamps	9876	Bart	5 Pine St	stamps				
<i>Id</i>	<i>Name</i>	<i>Address</i>	<i>Hobby</i>																																			
1123	John	123 Main	stamps																																			
1123	John	123 Main	coins																																			
5556	Mary	7 Lake Dr	hiking																																			
9876	Bart	5 Pine St	stamps																																			
<i>Id</i>	<i>Name</i>	<i>Address</i>	<i>Hobby</i>																																			
1123	John	123 Main	stamps																																			
9876	Bart	5 Pine St	stamps																																			
28.	<p><b>Define project operator in relational algebra.</b></p> <p>Produces table containing subset of columns of argument table</p> <ul style="list-style-type: none"> <li>Example:</li> </ul> $\Pi_{\text{attribute list}}(\text{relation})$ <p>Example:</p> <table> <thead> <tr> <th colspan="4"><b>Person</b></th> <th><math>\Pi_{\text{Name,Hobby}}(\text{Person})</math></th> </tr> <tr> <th><i>Id</i></th> <th><i>Name</i></th> <th><i>Address</i></th> <th><i>Hobby</i></th> <th><i>Name</i></th> <th><i>Hobby</i></th> </tr> </thead> <tbody> <tr> <td>1123</td> <td>John</td> <td>123 Main</td> <td>stamps</td> <td>John</td> <td>stamps</td> </tr> <tr> <td>1123</td> <td>John</td> <td>123 Main</td> <td>coins</td> <td>John</td> <td>coins</td> </tr> <tr> <td>5556</td> <td>Mary</td> <td>7 Lake Dr</td> <td>hiking</td> <td>Mary</td> <td>hiking</td> </tr> <tr> <td>9876</td> <td>Bart</td> <td>5 Pine St</td> <td>stamps</td> <td>Bart</td> <td>stamps</td> </tr> </tbody> </table>	<b>Person</b>				$\Pi_{\text{Name,Hobby}}(\text{Person})$	<i>Id</i>	<i>Name</i>	<i>Address</i>	<i>Hobby</i>	<i>Name</i>	<i>Hobby</i>	1123	John	123 Main	stamps	John	stamps	1123	John	123 Main	coins	John	coins	5556	Mary	7 Lake Dr	hiking	Mary	hiking	9876	Bart	5 Pine St	stamps	Bart	stamps	C212.1	BTL1
<b>Person</b>				$\Pi_{\text{Name,Hobby}}(\text{Person})$																																		
<i>Id</i>	<i>Name</i>	<i>Address</i>	<i>Hobby</i>	<i>Name</i>	<i>Hobby</i>																																	
1123	John	123 Main	stamps	John	stamps																																	
1123	John	123 Main	coins	John	coins																																	
5556	Mary	7 Lake Dr	hiking	Mary	hiking																																	
9876	Bart	5 Pine St	stamps	Bart	stamps																																	
29.	<p><b>Define Cartesian product in relational algebra.</b></p> <ul style="list-style-type: none"> <li>If <math>R</math> and <math>S</math> are two relations, <math>R \times S</math> is the set of all concatenated tuples <math>\langle x, y \rangle</math>, where <math>x</math> is a tuple in <math>R</math> and <math>y</math> is a tuple in <math>S</math> <ul style="list-style-type: none"> <li>(<math>R</math> and <math>S</math> need not be union compatible)</li> </ul> </li> <li><math>R \times S</math> is <u>expensive to compute</u>:</li> <ul style="list-style-type: none"> <li>Factor of two in the size of each row</li> <li>Quadratic in the number of rows</li> </ul> </ul> <p>Example:</p> <table> <thead> <tr> <th><math>R</math></th> <th><math>S</math></th> <th><math>R \times S</math></th> </tr> </thead> <tbody> <tr> <td> <math display="block">\begin{array}{cc} a &amp; b \\ x_1 &amp; x_2 \\ x_3 &amp; x_4 \end{array}</math> </td> <td> <math display="block">\begin{array}{cc} c &amp; d \\ y_1 &amp; y_2 \\ y_3 &amp; y_4 \end{array}</math> </td> <td> <math display="block">\begin{array}{cccc} a &amp; b &amp; c &amp; d \\ x_1 &amp; x_2 &amp; y_1 &amp; y_2 \\ x_1 &amp; x_2 &amp; y_3 &amp; y_4 \\ x_3 &amp; x_4 &amp; y_1 &amp; y_2 \\ x_3 &amp; x_4 &amp; y_3 &amp; y_4 \end{array}</math> </td> </tr> </tbody> </table>	$R$	$S$	$R \times S$	$\begin{array}{cc} a & b \\ x_1 & x_2 \\ x_3 & x_4 \end{array}$	$\begin{array}{cc} c & d \\ y_1 & y_2 \\ y_3 & y_4 \end{array}$	$\begin{array}{cccc} a & b & c & d \\ x_1 & x_2 & y_1 & y_2 \\ x_1 & x_2 & y_3 & y_4 \\ x_3 & x_4 & y_1 & y_2 \\ x_3 & x_4 & y_3 & y_4 \end{array}$	C212.1	BTL1																													
$R$	$S$	$R \times S$																																				
$\begin{array}{cc} a & b \\ x_1 & x_2 \\ x_3 & x_4 \end{array}$	$\begin{array}{cc} c & d \\ y_1 & y_2 \\ y_3 & y_4 \end{array}$	$\begin{array}{cccc} a & b & c & d \\ x_1 & x_2 & y_1 & y_2 \\ x_1 & x_2 & y_3 & y_4 \\ x_3 & x_4 & y_1 & y_2 \\ x_3 & x_4 & y_3 & y_4 \end{array}$																																				

30.	<b>What is a table in relational database.</b> A table is set of data elements that has a horizontal dimension (rows) and a vertical dimension (columns) in a relational database system. A table has a specified number of columns but can have any number of rows. Rows stored in a table are structurally equivalent to records from flat files. Columns are often referred as attributes or fields.	C212.1	BTL1
31.	<b>Define an identifier.</b> An identifier is an attribute that is used either as a primary key or as a foreign key. The integer datatype is used for identifiers. In cases where the number of records exceed the allowed values by the integer datatype then a big integer datatype is used.	C212.1	BTL1
32.	<b>Define a primary key.</b> A column in a table whose values uniquely identify the rows in the table. A primary key value cannot be NULL to matching columns in other tables\	C212.1	BTL1
33.	<b>What is a Foreign key?</b> A column in a table that does not uniquely identify rows in that table, but is used as a link to matching columns in other tables.	C212.1	BTL1
34.	<b>What is Relationship?</b> A relationship is an association between two tables. For example the relationship between the table "hotel" and "customer" maps the customers to the hotels they have used.	C212.1	BTL1
35.	<b>What is an index in relational database?</b> An index is a data structure which enables a query to run at a sublinear-time. Instead of having to go through all records one by one to identify those which match its criteria the query uses the index to filter out those which don't and focus on those who do.	C212.1	BTL1
36.	<b>Define view.</b> A view is a virtual or logical table composed of the result set of a pre-compiled query. Unlike ordinary tables in a relational database, a view is not part of the physical schema: it is a dynamic, virtual table computed or collated from data in the database. Changing the data in a view alters the data stored in the database	C212.1	BTL1
37.	<b>What is a Query</b> A query is a request to retrieve data from a database with the SQL SELECT instruction or to manipulate data stored in tables.	C212.1	BTL1

38.	<b>Define SQL</b> Structured Query Language (SQL), pronounced "sequel", is a language that provides an interface to relational database systems. It was developed by IBM in the 1970s for use in System R. SQL is a de facto standard, as well as an ISO and ANSI standard.	C212.1	BTL1
39.	<b>List any 5 responsibilities of the DB Manager <u>May June 2007</u></b> a) Interaction with the file manager b) Translation of DML commands into low level file system commands c) Storing, retrieving and updating data in the database d) Data dictionary e) Indices	C212.1	BTL1
40.	<b>What is Data Dictionary <u>May 2003, 2004, Nov 2006 &amp; Nov 2007</u>.</b> <b>Data dictionary</b> : which stores meta data about the structure of the database, in particular schema of the database.	C212.1	BTL1
41.	<b>What is Data independence &amp; what are the levels. <u>APRIL MAY 2008, MAY /JUNE 2012, Nov 2017</u></b> The ability to modify a schema definition in one level without affecting a schema definition in the next higher level called data independence. <b>Different levels .</b> Physical level Logical level View level	C212.1	BTL1
42.	<b>Define instance and schemas. <u>April/May 2012</u></b> - Database changes over time as information is inserted and deleted. - The collection of information stored in the database at a particular moment called an <b>instance of the database</b> . - The overall design of the database is called the <b>database schema</b> .	C212.1	BTL1
43.	<b>What are the types of attributes. <u>April/May 2007</u></b> o Simple o Composite o Single-valued o Multi-valued o Derived	C212.1	BTL1
44.	<b>What is mapping cardinalities <u>April/May 2009</u></b> Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.	C212.1	BTL1
45.	What are the various database languages in SQL? <b>April/May 2018</b> <b>Data Definition Language (DDL)</b> Commands that define a database, including creating, altering, and dropping	C212.1	BTL1

	<p>tables and establishing constraints</p> <p><b>Data Manipulation Language (DML)</b> Commands that maintain and query a database</p> <p><b>Data Control Language (DCL)</b> Commands that control a database, including administering privileges and committing data</p> <pre> graph LR     subgraph " "         direction TB         DDL[DDL Define the database: CREATE tables, indexes, views Establish foreign keys Drop or truncate tables]         DML[DML Load the database: INSERT data UPDATE the database Manipulate the database: SELECT]         DCL[DCL Control the database: GRANT, ADD, REVOKE]     end     PD[Physical Design]     I[Implementation]     M[Maintenance]      DDL --&gt; PD     DDL --&gt; I     DDL --&gt; M     DML --&gt; I     DML --&gt; M     DCL --&gt; I     DCL --&gt; M </pre>																											
46.	<p>List the data types in SQL.</p> <table border="1"> <tbody> <tr> <td rowspan="3">Numeric types</td> <td>integer</td> <td>integer, int, smallint, long</td> </tr> <tr> <td>floating point</td> <td>float, real, double precision</td> </tr> <tr> <td>formatted</td> <td>decimal(i,j), dec(i,j)</td> </tr> <tr> <td rowspan="2">Character-string types</td> <td>fixed length</td> <td>char(n), character(n)</td> </tr> <tr> <td>varying length</td> <td>varchar(n), char varying(n), character varying(n)</td> </tr> <tr> <td rowspan="2">Bit-string types</td> <td>fixed length</td> <td>bit(n)</td> </tr> <tr> <td>varying length</td> <td>bit varying(n)</td> </tr> <tr> <td>Date and time types</td> <td></td> <td>date, time, datetime, timestamp, time with time zone, interval</td> </tr> <tr> <td rowspan="2">Large types</td> <td>character</td> <td>long varchar(n), clob, text</td> </tr> <tr> <td>binary</td> <td>blob</td> </tr> </tbody> </table>	Numeric types	integer	integer, int, smallint, long	floating point	float, real, double precision	formatted	decimal(i,j), dec(i,j)	Character-string types	fixed length	char(n), character(n)	varying length	varchar(n), char varying(n), character varying(n)	Bit-string types	fixed length	bit(n)	varying length	bit varying(n)	Date and time types		date, time, datetime, timestamp, time with time zone, interval	Large types	character	long varchar(n), clob, text	binary	blob	C212.1	BTL1
Numeric types	integer		integer, int, smallint, long																									
	floating point		float, real, double precision																									
	formatted	decimal(i,j), dec(i,j)																										
Character-string types	fixed length	char(n), character(n)																										
	varying length	varchar(n), char varying(n), character varying(n)																										
Bit-string types	fixed length	bit(n)																										
	varying length	bit varying(n)																										
Date and time types		date, time, datetime, timestamp, time with time zone, interval																										
Large types	character	long varchar(n), clob, text																										
	binary	blob																										
47.	<p><b>What are the function of DBA? April/May 2009</b></p> <ul style="list-style-type: none"> <li>- Schema definition.</li> <li>- Storage structure and access –method definition.</li> <li>- Schema and physical-organization modification</li> <li>- Granting of authorization for data access:</li> <li>- Routine maintenance</li> </ul>	C212.1	BTL1																									
48.	<p><b>What is Relational Model . April/May 2009</b></p> <ul style="list-style-type: none"> <li>- The relation is the only construct required to represent the associations</li> </ul>	C212.1	BTL1																									

	among the attributes of an entity as well as the relationship among different entities. - A relation may be visualized as a named table. Each column of the table corresponds to an attribute of the relation and is named.		
49.	<b>Define Data Model. April/May 2009</b> It is a collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints	C212.1	BTL1
50.	What the problems caused by the redundancy? The problems caused by data redundancy are <ul style="list-style-type: none"> <li>• The first is that storing values multiple times will lead to waste spaces.</li> <li>• The second problem is that when a field value changes, multiple occurrences need to be updated. For example, if customer-1 moves, it needs to change the values of Street, City, State and Zip in multiple records.</li> <li>• The third problem occurs when forget to change the values in any of the records. The database would then have inconsistent data.</li> </ul>	C212.1	BTL1
51.	What is data definition language? <b>Data Definition Language:</b> A <b>data definition language</b> or <b>data description language (DDL)</b> is a syntax for defining the data structures using imperative verbs, especially database schemas.	C212.1	BTL1
52.	What is the syntax for creating a table in SQL? <b>Syntax:</b> <b>CREATE TABLE [table name] ([column definitions]) [table parameters]</b> <b>Example:</b> The command to create a table named employees with a few sample columns would be: <pre>CREATE TABLE employees (     id INTEGER PRIMARY KEY,     first_name VARCHAR(50) NULL,     last_name VARCHAR(75) NOT NULL,     fname VARCHAR(50) NOT NULL,     dateofbirth DATE NULL );</pre>	C212.1	BTL2
53.	What is the syntax for removing or deleting a table in SQL? <b>Drop</b> - To destroy an existing database, table, index, or view. <b>Syntax:</b> DROP objecttype objectname. <b>Example:</b> The command to drop a table named employees would be: <pre>DROP employees;</pre> The Drop statement would remove the entire table (employees) from the database.	C212.1	BTL2
54.	How to modify or alter an existing table in SQL. <b>Alter</b> - To modify an existing database object. An ALTER statement in SQL changes the properties of an object inside of a relational database management system (RDBMS). <b>Syntax:</b>	C212.1	BTL2

	<p>ALTER objecttype objectname parameters.</p> <p><b>Example:</b> The command to add (then remove) a column named bubbles for an existing table named sink would be:          ALTER TABLE sink ADD bubbles INTEGER;          ALTER TABLE sink DROP COLUMN bubbles;</p>		
55.	<p>How to rename an existing table in SQL?</p> <p><b>Syntax:</b> Rename - to rename the table.</p> <p><b>Example</b> RENAME TABLE old_name TO new_name;</p>	C212.1	BTL2
56.	<p>Define Data Manipulation Language.</p> <p>A data manipulation language (DML) is a set of commands permitting users to manipulate data in a database. This manipulation involves inserting data into database tables, retrieving existing data, deleting data from existing tables and modifying existing data.</p>	C212.1	BTL1
57.	<p>List the DML commands in SQL.</p> <p><b>SELECT:</b> This command is used to retrieve rows from a table.</p> <p><b>The select syntax:</b> SELECT [column name(s)] from [table name] where [conditions].</p> <p><b>UPDATE:</b> This command modifies data of one or more records.</p> <p><b>The update command syntax:</b> UPDATE table name SET column name = value where [condition].</p> <p><b>INSERT:</b> This command adds one or more records to a database table.</p> <p><b>The insert command syntax:</b> INSERT INTO table name [column(s)] VALUES [value(s)].</p> <p><b>DELETE:</b> This command removes one or more records from a table according to specified conditions.</p> <p><b>Delete command syntax:</b> DELETE FROM table name where [condition].</p>	C212.1	BTL1
58.	<p>List the aggregation functions in SQL.</p> <ul style="list-style-type: none"> <li>• COUNT returns the number of rows in a specified column.</li> <li>• SUM returns the sum of the values in a specified column.</li> <li>• AVG returns the average of the values in a specified column.</li> <li>• MIN returns the smallest value in a specified column.</li> <li>• MAX returns the largest value in a specified column.</li> </ul> <p><b>Examples:</b> Property (PropertyNo, Street, City, postcode, Type, OwnerNo, Rooms, Rent)</p> <p><b>Query:</b> How many properties cost more than 350 per month to rent?</p> <pre>SELECT COUNT(*) AS count FROM property WHERE rent &gt; 350;</pre>	C212.1	BTL1
59.	<p>What is subquery.</p> <p><b>Subquery</b></p> <ul style="list-style-type: none"> <li>• A complete SELECT statement can be embedded (subselect) within another SELECT statement.</li> <li>• A subselect can be used in the WHERE and HAVING</li> </ul>	C212.1	BTL1

	<p>clauses of the outer SELECT statement (nested query).</p> <ul style="list-style-type: none"> <li>• A subquery can be used immediately following a relational operator.</li> <li>• Subquery always enclosed in parentheses.</li> </ul> <p><b>Type of subquery:</b></p> <ul style="list-style-type: none"> <li>• A scalar subquery returns a single column and a single row (singlevalue).</li> <li>• A row subquery returns multiple columns, but a single row.</li> <li>• A table subquery returns one or more columns and multiple rows.</li> </ul> <p><b>Example:</b> List the staff who work in the branch at <u>_163 Main St‘.</u></p> <pre>SELECT sno, fname, lname, position FROM staff WHERE bno = (SELECT bno FROM branch WHERE street = <u>_163 Main St‘);</u></pre>		
60.	<p><b>How to modify a data in a table in SQL?</b></p> <p><b>Modifying Data in the DB (UPDATE)</b></p> <p><b>Syntax</b></p> <pre>UPDATE table_name SET column_name1 = data_value1 [, column_namei = data_valuei ...] [WHERE search_condition]</pre> <p><b>Description:</b></p> <ul style="list-style-type: none"> <li>• table_name may be either a base table or an updatable view.</li> <li>• The SET clause specifies the names of one or more columns that are updated for all rows in the table.</li> <li>• Only rows that satisfy the search_condition are updated.</li> <li>• data_values must be compatible with the data types for the corresponding columns.</li> </ul>	C212.1	BTL3
61.	<p><b>What is DCL?</b></p> <p><b>The Data Control Language (DCL)</b> component of the SQL language is used to create privileges to allow users access to, and manipulation of, the database.</p> <p>There are two main commands:</p> <p style="padding-left: 20px;"><b>GRANT</b> to grant a privilege to a user</p> <p style="padding-left: 20px;"><b>REVOKE</b> to revoke (remove) a privilege from a user</p>	C212.1	BTL1

	Define GRANT command in SQL. <b>The Syntax for the GRANT command is:</b> GRANT privilege_name ON object_name TO {user_name  PUBLIC  role_name} [WITH GRANT OPTION]; <b>Description:</b> <b>privilege_name</b> is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT. <b>object_name</b> is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE. <b>user_name</b> is the name of the user to whom an access right is being granted. <b>PUBLIC</b> is used to grant access rights to all users. <b>ROLES</b> are a set of privileges grouped together. <b>WITH GRANT OPTION</b> - allows a user to grant access rights to other users.  <b>Example:</b> GRANT SELECT ON employee TO user1; <b>Description:</b> This command grants a SELECT permission on employee table to user1. WITH GRANT option should be used carefully because if the GRANT SELECT privilege is permitted on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.	C212.1	BTL1
62.	<b>Write about REVOKE command in SQL.</b> <b>SQL REVOKE Command:</b> The REVOKE command removes user access rights or privileges to the database objects. <b>The Syntax for the REVOKE command is:</b> REVOKE privilege_name ON object_name FROM {user_name  PUBLIC  role_name} <b>Example:</b> REVOKE SELECT ON employee FROM user1; <b>Description:</b> This command will REVOKE a SELECT privilege on employee table from user1. When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it.	C212.1	BTL1
63.	<b>Define privileges in SQL.</b> <b>Privileges:</b> Privileges defines the access rights provided to a user on a database object. There are two types of privileges. 1) <b>System privileges</b> - This allows the user to CREATE, ALTER, or DROP database objects. 2) <b>Object privileges</b> - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.	C212.1	BTL1
64.	What is system privileges? <b>System privileges</b> - This allows the user to CREATE, ALTER, or DROP database objects. Few CREATE system privileges are listed below:	C212.1	BTL1
65.			

	<table border="1"> <thead> <tr> <th>System Privileges</th><th>Description</th></tr> </thead> <tbody> <tr> <td>CREATE object</td><td>allows users to create the specified object in their own schema.</td></tr> <tr> <td>CREATE ANY object</td><td>allows users to create the specified object in any schema.</td></tr> </tbody> </table> <p>The above rules also apply for ALTER and DROP system privileges.</p>	System Privileges	Description	CREATE object	allows users to create the specified object in their own schema.	CREATE ANY object	allows users to create the specified object in any schema.						
System Privileges	Description												
CREATE object	allows users to create the specified object in their own schema.												
CREATE ANY object	allows users to create the specified object in any schema.												
66.	<p><b>What is object privileges?</b></p> <p><b>Object privileges</b> - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.</p> <table border="1"> <thead> <tr> <th>Object Privileges</th><th>Description</th></tr> </thead> <tbody> <tr> <td>INSERT</td><td>allows users to insert rows into a table.</td></tr> <tr> <td>SELECT</td><td>allows users to select data from a database object.</td></tr> <tr> <td>UPDATE</td><td>allows user to update data in a table.</td></tr> <tr> <td>EXECUTE</td><td>allows user to execute a stored procedure or a function.</td></tr> </tbody> </table>	Object Privileges	Description	INSERT	allows users to insert rows into a table.	SELECT	allows users to select data from a database object.	UPDATE	allows user to update data in a table.	EXECUTE	allows user to execute a stored procedure or a function.	C212.1	BTL1
Object Privileges	Description												
INSERT	allows users to insert rows into a table.												
SELECT	allows users to select data from a database object.												
UPDATE	allows user to update data in a table.												
EXECUTE	allows user to execute a stored procedure or a function.												
67.	<p>Define roles in SQL.</p> <p><b>Roles:</b> Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.</p>	C212.1	BTL1										
68.	<p>List the system privileges granted to role.</p> <table border="1"> <thead> <tr> <th>System Role</th><th>Privileges Granted to the Role</th></tr> </thead> <tbody> <tr> <td>CONNECT</td><td>CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.</td></tr> <tr> <td>RESOURCE</td><td>CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.</td></tr> <tr> <td>DBA</td><td>ALL SYSTEM PRIVILEGES</td></tr> </tbody> </table>	System Role	Privileges Granted to the Role	CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.	RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.	DBA	ALL SYSTEM PRIVILEGES	C212.1	BTL1		
System Role	Privileges Granted to the Role												
CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.												
RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.												
DBA	ALL SYSTEM PRIVILEGES												
69.	<p>Write the syntax for creating a role.</p> <p><b>The Syntax to create a role is:</b></p> <pre>CREATE ROLE role_name [IDENTIFIED BY password];</pre> <p><b>Example:</b> To create a role called "developer" with password as "pwd",  <code>CREATE ROLE developer [IDENTIFIED BY pwd];</code></p>	C212.1	BTL1										

	<p>It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password.</p>		
70.	<p>What is the syntax for dropping a role in SQL?</p> <p><b>The Syntax to drop a role from the database:</b></p> <pre>DROP ROLE role_name;</pre> <p><b>Example:</b> To drop a role called developer, you can write:</p> <pre>DROP ROLE testing;</pre>	C212.1	BTL1
71.	<p>Define TCL.</p> <p><b>TCL</b> - Transactional Control Language. It is used to manage different transactions occurring within a database.</p> <p><b>Examples of TCL commands include:</b></p> <p><b>COMMIT</b> to apply the transaction by saving the database changes.</p> <p><b>ROLLBACK</b> to undo all changes of a transaction.</p> <p><b>SAVEPOINT</b> to divide the transaction into smaller sections. It defines breakpoints for a transaction to allow partial rollbacks.</p>	C212.1	BTL1
72.	<p>Define COMMIT in SQL.</p> <p><b>SYNTAX:</b></p> <pre>COMMIT [WORK] [ COMMENT 'text'   FORCE 'text' [, integer] ]</pre> <p><b>Description:</b></p> <p><b>WORK</b> : is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.</p> <p><b>COMMENT</b> : specifies a comment to be associated with the current transaction. The 'text' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.</p> <p><b>FORCE</b> : manually commits an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the integer to specifically assign the transaction a system change number (SCN). If you omit the integer, the transaction is committed using the current SCN. COMMIT statements using the FORCE clause are not supported in PL/SQL.</p>	C212.1	BTL1
73.	<p>Define ROLLBACK statement in SQL.</p> <p><b>PURPOSE:</b></p> <p>To undo work done in the current transaction. You can also use this command to manually und the work done by an in-doubt distributed transaction.</p> <p><b>SYNTAX:</b></p> <pre>ROLLBACK [WORK][ TO [SAVEPOINT] savepoint   FORCE 'text' ]</pre> <p><b>Description:</b></p> <p><b>WORK</b> : is optional and is provided for ANSI compatibility.</p> <p><b>TO</b> : rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.</p> <p><b>FORCE</b> : manually rolls back an in-doubt distributed transaction. The</p>	C212.1	BTL1

	transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. ROLLBACK statements with the FORCE clause are not supported in PL/SQ		
74.	<p><b>What is embedded SQL?</b></p> <p>Embedded SQL statements are SQL statements written inline with the program source code of the host language. The embedded SQL statements are parsed by an embedded SQL preprocessor and replaced by host-language calls to a code library. The output from the preprocessor is then compiled by the host compiler. This allows programmers to embed SQL statements in programs written in any number of languages such as: C/C++, COBOL and Fortran.</p>	C212.1	BTL1
75.	<p><b>What is static SQL? Dec 2017</b></p> <p><b>Static SQL</b></p> <p>The source form of a static SQL statement is embedded within an application program written in a host language such as COBOL.</p> <p>The statement is prepared before the program is executed and the operational form of the statement persists beyond the execution of the program.</p> <p>Static SQL statements in a source program must be processed before the program is compiled. This processing can be accomplished through the DB2 precompiler or the SQL statement coprocessor.</p> <p>The DB2 precompiler or the coprocessor checks the syntax of the SQL statements, turns them into host language comments, and generates host language statements to invoke DB2.</p> <p>The preparation of an SQL application program includes precompilation, the preparation of its static SQL statements, and compilation of the modified source program.</p>	C212.1	BTL1
76.	<p><b>What is dynamic SQL?</b></p> <p><b>Dynamic SQL:</b></p> <p>Programs that contain embedded dynamic SQL statements must be precompiled like those that contain static SQL, but unlike static SQL, the dynamic statements are constructed and prepared at run time.</p> <p>The source form of a dynamic statement is a character string that is passed to DB2 by the program using the static SQL statement PREPARE or EXECUTE IMMEDIATE.</p>	C212.1	BTL1

**PART-B**

Q. No.	Questions	CO	Bloom's Level

1.	List and describe the features and purpose of database?	C212.1	BTL5
2.	List and explain the limitations of file processing system?	C212.1	BTL5
3.	Compare database vs file processing system?	C212.1	BTL2
4.	What is data model? Explain various data models with example?	C212.1	BTL2
5.	Explain the database system architecture with neat diagram? <b>April/May 2017, Nov/Dec 2017</b>	C212.1	BTL2
6.	List and describe the components of database management system with neat diagram.	C212.1	BTL2
7.	Define relational algebra. Explain various relational algebraic operations with example. <b>Nov/Dec 2016, April/May 2017</b>	C212.1	BTL2
8.	List and explain the properties of a relations with example.	C212.1	BTL2
9.	What are the various data types in SQL? Explain them with example?	C212.1	BTL2
10.	List and explain various database languages with example?	C212.1	BTL2
11.	Explain DML with example?	C212.1	BTL3
12.	Explain DDL with example?	C212.1	BTL3
13.	Explain DCL with example?	C212.1	BTL3
14.	Explain TCL with example?	C212.1	BTL3
15.	Justify the need of embedded SQL. Consider the relation student (Reg No, name, mark and grade). Write embedded dynamic SQL program in C language to retrieve all the students records whose mark is more than 90. <b>Nov/Dec 2016, April/May 2017</b> .	C212.1	BTL5

## UNIT II

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

Q. No.	Questions	CO	Bloom's Level

1.	<b>What is an entity relationship model? May/ June 2016</b> 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.	C212.2	BTL1
2.	<b>Define weak and strong entity sets? April/May 2009, April/May 2018</b> <b>Weak entity set:</b> entity set that do not have key attribute of their own are called weak entity sets. <b>Strong entity set:</b> Entity set that has a primary key is termed a strong entity set.	C212.2	BTL1
3.	<b>Give the limitations of ER model? How do you overcome this? May/ June 2007</b> 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.	C212.2	BTL1
4.	<b>Define Specialization and Aggregation.</b> It is the process of designating sub groupings within an entity set. It is a top down process. <b>Specialization</b> which is represented by triangle. The label ISA stands for “is a and represent, for eg that customer <b>is a</b> person. <b>Aggregation</b> is a special kind of association that specifies a whole/part relationship between the aggregate (whole) and a component part.	C212.2	BTL1
5.	<b>What are three characteristics of a relational database system? Nov/Dec 2008</b> Controlling redundancy Restricting unauthorized access Providing multiple user interfaces	C212.2	BTL1
6.	<b>Give the distinction between primary key, candidate key and super key. Nov/Dec 2006,2009</b> Primary key – is used in a data base to avoid duplication of attributes and also makes a relation to the other database. Candidate key - a key which is in the data base is called as candidate key, it might be any key attribute. Super key – collection of keys of a database is called as super key	C212.2	BTL1
7.	<b>Define functional dependency. Nov/Dec 2010, Apr/ May 2015</b> A functional dependency is a constraint between two sets of attributes from the data base. A functional dependency , denoted by $X \rightarrow Y$ Between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation instance r of R. $R = \{ A_1, A_2, \dots, A_n \}$ . The constraint states that ,for any two tuples t1 and t2 in r such that $t1[X] = t2[X]$ , we must also have $t1[Y] = t2[Y]$ . <b>We can also say that Y is functionally dependent on X.</b>	C212.2	BTL1

8.	<p>Define an entity?</p> <p><b>Entities:</b> Entity -a thing (animate or inanimate) of independent physical or conceptual existence and distinguishable.</p> <p><b>Example:</b> In the University database context, an individual student, faculty member, a class room, a course are entities.</p>	C212.2	BTL1
9.	<p>What is an entity set?</p> <p>Entity Set or Entity Type-Collection of entities all having the same properties.</p> <p><b>Example:</b> Student entity set –collection of all student entities. Course entity set –collection of all course entities.</p>	C212.2	BTL2
10.	<p>What is an attribute?</p> <p><b>Attributes:</b> Attributes - Each entity is described by a set of attributes/properties.</p> <p><b>Example:</b> StudName–name of the student. RollNumber–the roll number of the student. Sex–the gender of the student etc. All entities in an Entity set/type have the same set of attributes.</p>	C212.2	BTL2
11.	<p>What is derived attributes?</p> <p>Derived attributes are those created by a formula or by a summary operation on other attributes</p>	C212.2	BTL2
12.	<p>What is a recursive relationship?</p> <p>Recursive relationships A recursive relationship is an entity is associated with itself.</p> <p><b>Example:</b> An employee may manage many employees and each employee is managed by one employee.</p> <pre> graph TD     subgraph Recursive_Relationship [Recursive Relationship]         direction TB         E[EMPLOYEE] -- manages --&gt; E         E --- L[is managed]         L --- Note[Employee may manage other employees. Each employee is managed by at most one employee.]     end   </pre>	C212.2	BTL2
13.	<p><b>Define cardinality?</b></p> <p><b>Cardinality</b> A business rule indicating the number of times a particular object or</p>	C212.2	BTL1

	activity may occur.		
14.	<p><b>List the properties of a relation?</b></p> <p>All entries in a given column are of the same kind or type  The ordering of columns is immaterial.  No two tuples are exactly the same.  There is only one value for each attribute of a tuple.  The ordering of tuples is immaterial.</p>	C212.2	BTL1
15.	<p><b>Define a key?</b></p> <p>A key is a set of attributes that uniquely identifies an entire tuple, a functional dependency allows us to express constraints that uniquely identify the values of certain attributes.</p> <p>First, keys will typically be used as links, ie. key values will appear in other relations to represent their associated tuples  Second, keys form the basis for constructing indexes to speed up retrieval of tuples from a relation. Small keys will decrease the size of indexes and the time to look up an index.</p>	C212.2	BTL2
16.	<p><b>Define a relational schema?</b></p> <p>A Relational Database Schema comprises</p> <ol style="list-style-type: none"> <li>1. the definition of all domains</li> <li>the definition of all relations, specifying for each its intension (all attribute names), and</li> <li>a primary key</li> </ol>	C212.2	BTL1
17.	<p><b>List the uses of functional dependencies?</b></p> <p>We use functional dependencies to:  test relations to see if they are legal under a given set of functional dependencies.  If a relation <math>r</math> is legal under a set <math>F</math> of functional dependencies, we say that <math>r</math> satisfies <math>F</math>.  specify constraints on the set of legal relations  We say that <math>F</math> holds on <math>R</math> if all legal relations on <math>R</math> satisfy the set of functional dependencies <math>F</math>.</p>	C212.2	BTL1
18.	<p><b>Define normalization. <u>Nov/Dec2009.April/May2010</u></b></p> <p>Normalization of data is a process during which unsatisfactory relation schemas are decomposed by breaking up their attributes into smaller relation shemas that possess desirable properties.</p>	C212.2	BTL2
19.	<b>List the functional dependencies rules?</b>	C212.2	BTL1

	<p><b>1. Reflexivity Rule</b> --- If <math>X</math> is a set of attributes and <math>Y</math> is a subset of <math>X</math>, then <math>X \rightarrow Y</math> holds. each subset of <math>X</math> is functionally dependent on <math>X</math>.</p> <p><b>2. Augmentation Rule</b> --- If <math>X \rightarrow Y</math> holds and <math>W</math> is a set of attributes, and then <math>WX \rightarrow WY</math> holds.</p> <p><b>3. Transitivity Rule</b> --- If <math>X \rightarrow Y</math> and <math>Y \rightarrow Z</math> holds, then <math>X \rightarrow Z</math> holds.</p> <p><b>Derived Theorems from Axioms</b></p> <p><b>4. Union Rule</b> --- If <math>X \rightarrow Y</math> and <math>X \rightarrow Z</math> holds, then <math>X \rightarrow YZ</math> holds.</p> <p><b>5. Decomposition Rule</b> --- If <math>X \rightarrow YZ</math> holds, then so do <math>X \rightarrow Y</math> and <math>X \rightarrow Z</math>.</p> <p><b>6. Pseudo transitivity Rule</b> --- If <math>X \rightarrow Y</math> and <math>WY \rightarrow Z</math> hold then so does <math>WX \rightarrow Z</math>.</p>		
20.	<p>What is normalization?</p> <p><b>Database normalization</b> is the process of removing redundant data from the tables in to improve storage efficiency, data integrity, and scalability.</p> <p>In the relational model, methods exist for quantifying how efficient a database is. These classifications are called <b>normal forms</b> (or NF), and there are algorithms for converting a given database between them.</p> <p>Normalization generally involves splitting existing tables into multiple ones, which must be re-joined or linked each time a query is issued.</p>	C212.2	BTL1
21.	<p><b>Define data Anomalies.</b></p> <p><b>Data Anomalies</b></p> <p>Data anomalies are inconsistencies in the data stored in a database as a result of an operation such as update, insertion, and/or deletion. Such inconsistencies may arise when have a particular record stored in multiple locations and not all of the copies are updated.</p>	C212.2	BTL1
22.	<p><b>Define 1NF.</b></p> <p>Each table has a primary key: minimal set of attributes which can uniquely identify a record</p> <p>The values in each column of a table are atomic (No multi-value attributes allowed). There are no repeating groups: two columns do not store similar information in the same table</p>	C212.2	BTL1
23.	<p><b>Define 2NF.</b></p> <p>where a key has more than one attribute, check that each non-key attribute depends on the whole key and not part of the key</p> <p>for each subset of the key which determines an attribute or group of attributes create a new form. Move the <i>dependant</i> attributes to the new form.</p> <p>Add the part key to new form, making it the primary key.</p> <p>Mark the part key as a foreign key in the original form.</p>	C212.2	BTL1
24.	<p><b>Define 3NF.</b></p> <p>A relation R is in Third Normal Form (3NF) if and only if it is:</p> <p>in Second Normal Form.</p> <p>Every non-key attribute is non-transitively dependent on the primary key.</p>	C212.2	BTL1

	<b>Define BCNF</b> <b>Boyce-Codd Normal Form:</b> <ul style="list-style-type: none"><li>• A relation is in Boyce-Codd normal form (BCNF) if for every FD A <math>\rightarrow</math> B either<ul style="list-style-type: none"><li>○ B is contained in A (the FD is trivial), or A contains a candidate key of the relation,</li><li>• In other words: every determinant in a non-trivial dependency is a (super) key.</li></ul></li></ul> <p>The same as 3NF except in 3NF we only worry about non-key Bs If there is only one candidate key then 3NF and BCNF are the same</p>	C212.2	BTL1
25.	<b>List the decomposition properties. April/May 2017</b> <b>Decomposition Properties</b> <ul style="list-style-type: none"><li>• Lossless: Data should not be lost or created when splitting relations up</li><li>Dependency preservation: It is desirable that FDs are preserved when splitting relations up</li></ul> <p>Normalisation to 3NF is always lossless and dependency preserving Normalisation to BCNF is lossless, but may not preserve all dependencies</p>	C212.2	BTL1
26.	<b>Define MVD.</b> A multivalued dependency is a <i>full constraint</i> between two sets of attributes in a relation. In contrast to the <i>functional dependency</i> , the multivalued dependency requires that certain tuples be present in a relation. Therefore, a multivalued dependency is also referred as a <i>tuple-generating</i> dependency. The multivalued dependency also plays a role in 4NF normalization.	C212.2	BTL1
27.	<b>Define 4NF.</b> <b>4th Normal Form</b> A Boyce Codd normal form relation is in fourth normal form if there is no multi value dependency in the relation or  there are multi value dependency but the attributes, which are multi value dependent on a specific attribute, are dependent between themselves.	C212.2	BTL1
28.	sume the following relation <b>a:pk1, b:pk2, c:pk3</b>  Recall that a relation is in BCNF if all its determinant are candidate keys, in other words each determinant can be used as a primary key. Because relation R has only one determinant ( <b>a, b, c</b> ), which is the composite primary key and since the primary is a candidate key therefore R is in BCNF.		
29.	<b>Define 5NF</b> <b>FIFTH NORMAL FORM</b> R is in 5NF if and only if every join dependency in R is implied by the candidate keys of R 5NF is always achievable.	C212.2	BTL1

	<p>a join dependency, <math>\ast(A, B, \dots, Z)</math>, is implied by the candidate keys, <math>K_1, \dots, K_m</math> of <math>R</math></p> <p>ω the fact that <math>K_1, \dots, K_m</math> are candidate keys for <math>R</math> determine the fact that <math>R</math> has the JD <math>\ast(A, B, \dots, Z)</math></p>		
30.	<p>Show the binary relationship in diagram?</p> <p><b>A. ONE-TO-ONE</b></p> <pre>     graph LR       EMPLOYEE[EMPLOYEE] --- is assigned  WORKSTATION[WORKSTATION]   </pre> <p>Every employee is assigned one workstation; not all workstations are assigned to employees.</p> <p><b>B. ONE-TO-MANY</b></p> <pre>     graph LR       DEPARTMENT[DEPARTMENT] --- is responsible  PROJECT[PROJECT]   </pre> <p>A department may be responsible for many projects but each project is the responsibility of one department.</p> <p><b>C. MANY-TO-MANY</b></p> <pre>     graph LR       EMPLOYEE[EMPLOYEE] &lt;--&gt; is assigned, has assigned  PROJECT[PROJECT]   </pre> <p>Employees may be assigned to many projects; every project has assigned at least one employee.</p>	C212.2	BTL2
31.	<p>List the properties of a database relation.</p> <p>Properties of database relations are:</p> <ul style="list-style-type: none"> <li>• relation name is distinct from all other relations</li> <li>• each cell of relation contains exactly one atomic (single) value</li> <li>• each attribute has a distinct name</li> <li>• values of an attribute are all from the same domain</li> <li>• order of attributes has no significance</li> <li>• each tuple is distinct; there are no duplicate tuples</li> <li>• order of tuples has no significance, theoretically.</li> </ul>	C212.2	BTL1
32.	<p>List the various object-based logical models.</p> <ul style="list-style-type: none"> <li>– Entity-relationship model</li> <li>– Object-oriented model</li> <li>– Semantic model</li> <li>– Functional model</li> </ul>	C212.2	BTL1
33.	<p>Various Record-based logical models</p> <ul style="list-style-type: none"> <li>– Relational model (e.g., SQL/DS, DB2)</li> <li>– Network model</li> <li>– Hierarchical model (e.g., IMS)</li> </ul>	C212.2	BTL1

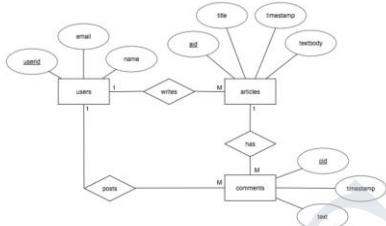
34.	<p>What is a domain in a database.  The term domain is used to refer to a set of values of the same kind or type. For example, the column, "Cname" and "Ccity" in the following figure, both have values of type string (ie. valid values are any string). But they denote different domains, ie."Cname" denotes the domain of customer names while "Ccity" denotes the domain of city names. They are different domains even if they share common values.</p> <table border="1" data-bbox="486 494 1073 705"> <thead> <tr> <th colspan="4">Customer</th> </tr> <tr> <th>C#</th><th>Cname</th><th>Ccity</th><th>Cphone</th></tr> </thead> <tbody> <tr> <td><b>1</b></td><td><b>Codd</b></td><td><b>London</b></td><td><b>2263035</b></td></tr> <tr> <td><b>2</b></td><td><b>Martin</b></td><td><b>Paris</b></td><td><b>5555910</b></td></tr> <tr> <td><b>3</b></td><td><b>Deen</b></td><td><b>London</b></td><td><b>2234391</b></td></tr> </tbody> </table> <p><b>Tuple</b></p>	Customer				C#	Cname	Ccity	Cphone	<b>1</b>	<b>Codd</b>	<b>London</b>	<b>2263035</b>	<b>2</b>	<b>Martin</b>	<b>Paris</b>	<b>5555910</b>	<b>3</b>	<b>Deen</b>	<b>London</b>	<b>2234391</b>	C212.2	BTL1
Customer																							
C#	Cname	Ccity	Cphone																				
<b>1</b>	<b>Codd</b>	<b>London</b>	<b>2263035</b>																				
<b>2</b>	<b>Martin</b>	<b>Paris</b>	<b>5555910</b>																				
<b>3</b>	<b>Deen</b>	<b>London</b>	<b>2234391</b>																				
35.	<p><b>What is candidate Keys?</b>  Candidate keys are those keys which is candidate for primary key of a table. Such keys will full fill all the requirements of primary key which is not null and have unique records is a candidate for primary key. So thus type of key is known as candidate key. Every table must have at least one candidate key but at the same time can have several.</p>	C212.2	BTL1																				
36.	<p><b>Define the Primary Key.</b>  Unique attribute of a table will be considered as a primary key. Primary keys are used to identify tables. There is only one primary key per table.</p>	C212.2	BTL1																				
37.	<p><b>What is Foreign Key? April/May 2018</b>  Foreign key are those keys which is used to define relationship between two tables. When we want to implement relationship between two tables then we use concept of foreign key. It is also known as referential integrity. We can create more than one foreign key per table. foreign key is generally a primary key from one table that appears as a field in another where the first table has a relationship to the second. In other words, if we had a table A with a primary key X that linked to a table B where X was a field in B, then X would be a foreign key in B.</p>	C212.2	BTL1																				
38.	<p><b>Define alternate Key.</b>  If any table have more than one candidate key, then after choosing primary key from those candidate key, rest of candidate keys are known as an alternate key of that table. Suppose we have a table named Employee which has two columns EmpID and EmpMail, both have not null attributes and unique value. If EmpID is considered as a primary key to that table then EmpMail is known as alternate key.</p>	C212.2	BTL1																				
39.	<p><b>Define Composite Key</b>  When a key is created on more than one column then that key is known as composite key. Consider a table "Student" which has two columns Sid and SrefNo and these two columns are used as a primary key for record extraction, then this key is known as composite key.</p>	C212.2	BTL1																				

	<b>Distinguish between key and super key? April/May 2017</b> A <b>Superkey</b> , SK, specifies a uniqueness constraint that no two distinct tuples in any state r of R can have the same value for SK.  40. A <b>Key</b> , K, of R is a Superkey of R with additional property that removing any attribute A, from K leaves set of attributes, that is not a superkey any more.  So the difference between Superkey and key is that removing any attribute from superkey may or may not leave set of attributes that are superkey but removing any attribute from a key will give a set that will no more be a super key.	C212.2	
41.	List the similarities between primary and candidate keys. Both Primary and Candidate keys can uniquely identify records in a table on the database.  Both Primary and Candidate keys have constraints UNIQUE and NOT NULL.  Primary key or Candidate keys can be either single column or combination of multiple columns in a table.	C212.2	BTL1
42.	What are the characteristics of primary key? Primary key is a special kind of index in that, <ul style="list-style-type: none"><li>• there can be only one;</li><li>• it cannot be nullable</li><li>• it must be unique.</li></ul>	C212.2	BTL1
43.	Define super key. A super key is a set or one of more columns (attributes) to uniquely identify rows in a table.	C212.2	BTL1
44.	<b>What is need for normalization. Nov/Dec 2010</b> To ensure that the update anomalies do not occur. Normal forms provide a formal frame work for analyzing relation shemas based on their keys and on the functional dependencies among their attributes.  A series of tests that can be carried out on individual relation schemas so that the relation database can be normalized to any degree. When a test fails , the relation violating that test must be decomposed into relations that individually meet the normalization tests	C212.2	BTL1
45.	<b>Define lossless joins (or) What is non additive property. Nov/Dec 2011</b> Lossless join property or non additive property ensures that no spurious tuples(tuples containing wrong information) are generated when a natural join operation is applied to the relations in the decomposition.	C212.2	BTL1
46.	<b>Explain BCNF with example(or) How to convert a relation into BCNF. Nov/dec 2007, 2008, 2009 2010, 2011,2014</b> Boyce-Codd Normal form: It is stricter than 3NF, meaning that every relation in BCNF is also in 3NF; however a relation in 3NF is not necessarily in BCNF. A relation is in BCNF if and only if every determinant is a candidate key (i.e) a relatioln schema R is in BCNF if whenever a functional dependency X->A holds in R, then X is a superkey of R	C212.2	BTL1
47.	<b>What are pitfalls in relational database design?Nov/Dec2009 2010</b> <ul style="list-style-type: none"><li>○ Repetition of informaiton</li></ul>	C212.2	BTL1

	<ul style="list-style-type: none"> <li>○ Inability to represent certain information</li> <li>○ Loss of information</li> </ul>		
48.	<p><b>State the anomalies of 1NF. Nov/ Dec 2015.</b></p> <p>No repeating groups Redundancy of data</p>	C212.2	BTL1
49.	<p><b>Explain the desirable properties of decomposition.</b></p> <p>Lossless-join decomposition Dependency preservation Repetition of information</p>	C212.2	BTL1
50.	<p><b>Outline the steps involved in query processing. April/May 2018.</b></p> <p>Parsing and translation Optimization Evaluation</p>	C212.2	BTL1
51.	<p><b>Explain trivial dependency? Nov/Dec 2009, 2010</b></p> <p>Functional dependency of the form <math>\alpha \rightarrow\beta</math>. is trivial if <math>\alpha \subseteq \beta</math>. Trivial functional dependencies are satisfied by all the relations.</p>	C212.2	BTL1
52.	<p><b>Why must multivalued dependencies exist in pairs? Nov/DEC 2010, 2011</b></p> <p>Multivalued dependencies: Multivalued dependencies are a result of 1NF which disallowed an attribute in a tuple to have a set of values. If we have two or more multivalued independent attributes in the same relation schema, we get into the problem of having to repeat every value of one of the attributes with every value of the other attributes with every value of the other attribute to keep the relation instances consistent. A multivalued dependency <math>X \multimap Y</math> specified on relation schema R where X and Y are subsets of R specifies the following constraint on any relation r of R.</p>	C212.2	BTL1
53.	<p><b>Comparison of BCNF and 3NF: Nov/Dec 2010, April/may2011</b></p> <p>3NF design is always dependency preserving and lossless. dependency preserving is difficult to achieve in BCNF sometimes. BCNF strictly removes transitive dependency. BCNF relation is in 3NF , but reverse is not possible</p>	C212.2	BTL1
54.	<p><b>Why 4NF in Normal Form is more desirable than BCNF? (Nov/Dec 2014, Dec 2017)</b></p> <p>BCNF (Boyce code normal form) has all functional dependencies A to B are trivial or discriminator should be superkey. To get relation in BCNF, Splitting the relation schema not necessarily preserve all functional dependency, Loss less decomposition and dependency are main points for the normalization sometime, it is not possible to get a BCNF decomposition that is dependency, preserving. While 4NF has very similar definition as BCNF. A relational Schema is in 4NF, if all multivalued dependencies A to B are trivial and determinate A is superkey of schema. If a relational schema is in 4nf, it is already in BCNF. and 4NF decomposition preserve the all functional dependency. so 4NF is preferable than to have BCNF.</p>	C212.2	BTL1

**PART-B**

Q. NO.	QUESTIONS	CO	BLO OM'S LEVE L
1	<b>Explain The Different Types Of Normalization <u>Nov/Dec 2009, 2010, 2011, Nov/ Dec 2014, Apr/May 2015, Nov/Dec 2016</u> Refer the Elmasri Navathe page no. 517</b>	C212.2	BTL5
2	<b>Explain In Detail About Boyce Codd Normal Form And Non Lossless Join Dependency. <u>Nov/Dec 2009, 2010, 2011</u> Refer the Elmasri Navathe page no. 529</b>	C212.2	BTL5
3	<b>Give Detail explanation about Data Model &amp; its types. <u>May/ June 2009, Nov/ Dec 2014</u> Refer the Elmasri Navathe page no. 30</b>	C212.2	BTL5
4	<b>Explain the various components of ER diagram with examples Briefly. Refer the Elmasri Navathe page no. 203</b>	C212.2	BTL5
5	<b>Draw ER Diagram for Difference applications ( Eg. Banking Systems . Nov/ Dec 2014, Nov/Dec 2017, Restaurant menu ordering system Apr/ May 2015, Car Rental company <u>Nov/ Dec 2015), Car Insurance Company (Nov/Dec 2016)</u></b>	C212.2	BTL5
6	<b>Discuss the correspondence between the ER model construct and the relational model constructs. Show how each ER model construct can be mapped to the relational model. Discuss the option for mapping EER model construct. April/May 2017</b>	C212.2	BTL5
7	<b>Various join strategies with Example. Nov/Dec 2016 Nested loop join Block nested loop join Merge join Hash join</b>	C213.2	BTL6
8	<b>Differentiate between foreign key constraints and referential integrity constraints with suitable example. Nov/Dec 2017</b>	C213.2	BTL5
9	<b>Distinguish between lossless-join decomposition and dependency preserving decomposition. Nov/Dec 2017.</b>	C213.2	BTL5
10	<b>Explain FNF, SNF, TNF and DCNF with an example. Nov/Dec 2016</b>	C213.2	BTL1
11	<b>Give detail Explanation about Query processing / Optimization overview with example. <u>Nov/Dec 2014, May/ June 2016, Nov/Dec 2016, Nov/Dec 2017.</u> Elmasri Navathe page no.679</b>	C213.2	BTL1
12		C213.2	BTL5

	<p>A car rental company maintains a database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles.</p> <ul style="list-style-type: none"> <li>• Trucks: cargo capacity.</li> <li>• Sports cars: horsepower, renter age requirement.</li> <li>• Vans: number of passengers.</li> <li>• Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive).</li> </ul> <p>Construct an ER model for the car rental company database. (Nov/Dec 15)</p>		
13	<p>Construct an E-R diagram for a car insurance company whose customers cars each. Each car has associated with it zero to any number of recorded insurance policy covers one or more cars, and has one or more premium associated with it. Each payment is for a particular period of time set of date when the payment was received. (Nov/Dec 16)</p>	C213.2	BTL5
14	<p>For the ER diagram given below explain ER to Relational mapping procedures</p>  <pre>     erDiagram {         class User {             string email;             string name;         }         class Article {             string title;             string timestamp;             string textBody;         }         class Comment {             string id;             string timestamp;             string text;         }          User }o--o{ Article : writes         User }o--o{ Comment : posts         Article }o--o{ Comment : has     }   </pre>	C213.2	BTL1
15	<p>Draw E - R Diagram for the "Restaurant Menu Ordering System", which food items ordering and services within a restaurant. The entire restaura detailed as follows. The Customer is able to view the food items menu, c orders and obtain the final bill through the computer kept in their table. their wireless tablet PC are able to initialize a table for customers, control to assist customers, orders, send orders to food preparation staff (chef) a customer's bill. The food preparation staffs (Chefs), with their touch-dis system, are able to view orders sent to the kitchen by waiters. During pre able to let the waiter know the status of each item, and can send notificat completed. The system should have full accountability and logging facil support supervisor actions to account for exceptional circumstances, suc refunded or walked out on. (Apr/May 17)</p>	C213.2	BTL1

**UNIT III****TRANSACTIONS**

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

Q. No.	Questions	CO	Bloom's Level
1	<b>What is transaction?</b> <u>Nov/Dec 2010 Nov/Dec 2014</u> Collections of operations that form a single logical unit of work are called transactions.	C212.3	BTL1
2	<b>What are the properties of transaction?</b> <u>Nov /Dec 2009, April/May2010 NOV/DEC 2014 , May/June 2016 (or) What are the ACID properties? APRIL/MAY-2011 , Nov/Dec 2017,</u> The properties of transactions are: Atomicity Consistency Isolation Durability	C212.3	BTL1
3	<b>Define lock?</b> <u>Nov2009, Nov2010, Nov2011</u> Lock is the most common used to implement the requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item.	C212.3	BTL1
4	<b>What are the different modes of lock?</b> <u>Nov2009, Nov2011</u> The modes of lock are: Shared /Read Exclusive /Write	C212.3	BTL1
5	<b>Define deadlock?</b> <u>April 2009,April 2010, April2011</u> Neither of the transaction can ever proceed with its normal execution. This situation is called deadlock.	C212.3	BTL1
6	<b>Define the phases of two phase locking protocol</b> <u>April/May 2009</u> Growing phase: a transaction may obtain locks but not release any lock. Shrinking phase: a transaction may release locks but may not obtain any new locks.	C212.3	BTL1
7	<b>What is meant by log-based recovery?</b> <u>April 2009</u> The most widely used structures for recording database modifications is the log. The log is a sequence of log records, recording all the update activities in the database. There are several types of log records.	C212.3	BTL1
8	<b>Define shadow paging.</b> <u>Nov/Dec 2009</u>	C212.3	BTL1

	An alternative to log-based crash recovery technique is shadow paging. This technique needs fewer disk accesses than do the log-based methods.		
9	<b>When is a transaction rolled back?</b> Any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, then the transaction has been rolled back.	C212.3	BTL1
10	<b>What is a recovery scheme? April/May 2009</b> An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure.	C212.3	BTL1
11	<b>What are two pitfalls (problem) of lock-based protocols? APRIL/MAY-2011</b> Deadlock Starvation	C212.3	BTL2
12	<b>What are the two statements regarding transaction?</b> The two statements regarding transaction of the form: → Begin transaction → End transaction	C212.3	BTL1
13	<b>What is recovery management component?</b> Ensuring durability is the responsibility of a software component of the base system called the recovery management component.	C212.3	BTL1
14	<b>When is a transaction rolled back?</b> Any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, then the transaction has been rolled back.	C212.3	BTL3
15	<b>What is a shadow copy scheme?</b> It is simple, but efficient, scheme called the shadow copy schemes. It is based on making 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.	C212.3	BTL1
16	<b>Give the reasons for allowing concurrency? Nov/Dec 2017</b> The reasons for allowing concurrency is if the transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. So concurrent execution reduces the unpredictable delays in running transactions.	C212.3	BTL1
17	<b>What is average response time?</b> The average response time is that the average time for a transaction to be completed after it has been submitted.	C212.3	BTL1
18	<b>What is serializability? Explain its types? NOV/DEC 2014, Nov/Dec 2016, April/May 2018</b> A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of: <ul style="list-style-type: none"><li>• conflict serializability</li><li>• view serializability</li></ul>	C212.3	BTL2

19	<b>Define upgrade and downgrade?</b> It provides a mechanism for conversion from shared lock to exclusive lock is known as upgrade. It provides a mechanism for conversion from exclusive lock to shared lock is known as downgrade.	C212.3	BTL1
20	<b>What is a database graph?</b> The partial ordering implies that the set D may now be viewed as a directed acyclic graph, called a database graph.	C212.3	BTL1
21	<b>What are the two methods for dealing deadlock problem?</b> The two methods for dealing deadlock problem is deadlock detection deadlock recovery.	C212.3	BTL1
22	<b>What is a recovery scheme?</b> An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure.	C212.3	BTL1
23	<b>Define garbage collection. May/June 2016</b> Garbage may be created also as a side effect of crashes. Periodically, it is necessary to find all the garbage pages and to add them to the list of free pages. This process is called garbage collection.	C212.3	BTL1
24	<b>Differentiate strict two phase locking protocol and rigorous two phase locking protocol. May/June 2016</b> In strict two phase locking protocol all exclusive mode locks taken by a transaction is held until that transaction commits. Rigorous two phase locking protocol requires that all locks be held until the transaction commits.	C212.3	BTL1
25	<b>How the time stamps are implemented</b> Use the value of the system clock as the time stamp. That is a transaction's time stamp is equal to the value of the clock when the transaction enters the system. • Use a logical counter that is incremented after a new timestamp has been assigned; that is the time stamp is equal to the value of the counter.	C212.3	BTL1
26	<b>What are the time stamps associated with each data item?</b> W-timestamp (Q) denotes the largest time stamp if any transaction that executed WRITE (Q) successfully. • R-timestamp (Q) denotes the largest time stamp if any transaction that executed READ (Q) successfully.	C212.3	BTL1
27	<b>Define blocks? What are its types?</b> The database system resides permanently on nonvolatile storage, and is partitioned into fixedlength storage units called blocks. <b>Physical blocks</b> The input and output operations are done in block units. The blocks residing on the disk are referred to as physical blocks. <b>Buffer blocks</b> The blocks residing temporarily in main memory are referred to as buffer blocks	C212.3	BTL1
28	<b>What is meant by concurrency control? Nov/Dec 2015</b> Process of managing simultaneous operations on the database without	C212.3	BTL1

	having them interfere with one another. Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.		
29	<b>Give an example of Two phase commit protocol. Nov/Dec 2015</b> In distributed databases, "single phase" commit is what is used. All the necessary information for the transaction to be undone or completed are written to persistent storage (typically called a "transaction log") in an atomic step. The transaction is committed as soon as this information is permanently recorded. This is not possible in a distributed system, as there's no guarantee that the commit record is written on all participating systems. With distributed databases, <u>Two Phase Commit</u> solves this problem.	C212.3	BTL2
30	<b>List the four conditions for deadlock. Nov/Dec 2016</b> 1. <b>mutual exclusion</b> : at least one process must be held in a non-shareable mode. 2. <b>hold and wait</b> : there must be a process holding one resource and waiting for another. 3. <b>No preemption</b> : resources cannot be preempted. 4. <b>circular wait</b> : there must exist a set of processes [p1, p2, ..., pn] such that p1 is waiting for p2, p2 for p3, and so on upto pn.	C212.3	BTL1
31	<b>What is serializable schedule? April/May 2017</b> To process transactions concurrently, the database server must execute some component statements of one transaction, then some from other transactions, before continuing to process further operations from the first. The order in which the component operations of the various transactions are interleaved is called the <b>schedule</b>	C212.3	BTL1
32	What type of locking needed for insert and delete operations? <b>April/May 2017</b> <b>Share/Exclusive (for Read/Write) Locks</b> We should allow several transactions to access the same item A if they all access A' for reading purposes only. However, if a transaction is to write an item A, it must have exclusive access to A. For this purpose, a different type of lock called a multiple-mode lock is used. In this scheme there are shared/exclusive or read/write locks are used.	C212.3	BTL1
33	<b>State the difference between a shared lock and an exclusive lock. April/May 2018.</b> <i>Shared lock:</i> Shared locks are placed on resources whenever a read operation (select) is performed. Multiple shared locks can be simultaneously set on a resource. <i>Exclusive lock:</i> Exclusive locks are placed on resources whenever a write operation (INSERT, UPDATE And DELETE) are performed. Only one exclusive lock can be placed on a resource at a time.	C212.3	BTL1

	i.e. the first user who acquires an exclusive lock will continue to have the sole ownership of the resource, and no other user can acquire an exclusive lock on that resource		
34	<b>What is query execution plan? April/May 2017</b> The Query Execution Plans describe the steps and the order used to access or modify data in the database.	C212.3	BTL1
35	<b>What are the costs involved in query execution? April/May 2017</b> i. Access cost to secondary storage. ii. Disk storage cost. iii. Computation cost iv. Memory usage cost and v. Communication cost	C212.3	BTL1
36	<b>Define atomicity?</b> Either all operations of the transaction are reflected properly in the database or none are.	C212.3	BTL1
37	<b>Define Durability?</b> A transaction completes successfully then changes it has made to the database persist even if there are system failure.	C212.3	BTL1
38	<b>What is transaction-management component?</b> Ensuring atomicity is the responsibility of the database system itself specifically, it is handled by a component called the transaction-management component.	C212.3	BTL1
39	<b>What are the two operation for accessing data in transaction?</b> Read(x)- transfer data item x from database. Write(x)- transfer data item x from the local buffer.	C212.3	BTL1
40	<b>What do you mean by read only transaction?</b> The data base operation in a transaction do not update the database but only retrieve data, the transaction is called a read-only transaction.	C212.3	BTL1
41	<b>What are the steps followed in Executing read(x) command in transaction?</b> 1. Find the address of the disk block that contains item x. 2. Copy that disk block into a buffer in main memory. 3. Copy item x from the buffer to the program variable named x.	C212.3	BTL1
42	<b>What are the steps followed in executing write(x) command in transaction?</b> 1. find the address of disk block that contain item x. 2. Copy that disk block into buffer in main memory. 3. Copy item x from the program variable named x into its correct location in the buffer 4. Store the update block from the buffer back to disk.	C212.3	BTL1
43	<b>List out the transaction states?</b> i) active ii) Partially Committed iii) Failed iv) Abort v) committed.	C212.3	BTL1
44	<b>What are the need for concurrency?</b> i) Improved throughput and resource utilization ii) Reduced waiting time.	C212.3	BTL1

45	<b>Define Schedule.</b> The chromosomal order in which instructions are executed in the system.	C212.3	BTL1
46	<b>When the schedule is called serial?</b> Each serial schedule consists of a sequence of instruction from various transaction where the instruction belonging to one single transaction appear together in that schedule.	C212.3	BTL1
47	<b>When two operation in schedule are said to be conflict?</b> 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)	C212.3	BTL1
48	<b>Define cascading rollback?</b> An uncommitted transaction has to be roll back because it read an item from a transaction that failure.	C212.3	BTL1
49	<b>When the schedule is said to cascade less?</b> A schedule is said to be cascadeless or avoid cascading roll back if every transaction in the schedule reads only items that were written by committed transactions.	C212.3	BTL1
50	<b>Define lock table?</b> System maintain record for the items that are currently locked in lock table that which could be organized as a hash file.	C212.3	BTL1
51	<b>What you mean by lock conversion?</b> A transaction that already holds a lock on item x is allowed under certain condition to vонcery the Locke from one locked state to another.	C212.3	

**PART-B**

Q. NO.	QUESTIONS	CO	BLOO M'S LEVE L
1	<b>Describe Log Based Recovery <u>May/june 2008</u></b> <i>Refer Elmasri Navathe page no.612</i>	C212.3	BTL4
2	<b>What is Serializability? Explain Its Types? <u>April2009, 2011 Nov/Dec 2014 , April/May 2018</u></b>	C212.3	BTL2
3	<b>Write Short Notes On Transaction Concept &amp; Transaction State? <u>April 2011 Nov/Dec 2014, April/May 2018</u></b> <i>Refer Elmasri Navathe page no.559, 562</i>	C212.3	BTL3
4	<b>Briefly Describe Concurrency control execution? Illustrate with a suitable example. <u>Nov/ Dec 2015, April/May 2018</u></b> <i>Refer Elmasri Navathe page no.583</i>	C212.3	BTL1
5	<b>Explain in terms of Locking mechanism and Two Phase Commit Protocol <u>Nov/Dec 2014, April / May 2015, May/June 2016, Nov/Dec 2017</u></b>	C212.3	BTL2

	<i>Refer the Elmasri Navathe page no 584</i>		
6	<b>Explain about locking protocols.</b> <u>May/June 2016</u> <i>Refer the Elmasri Navathe page no 584</i>	C212.3	BTL2
7	<b>Discuss about conflict and view serializability.</b> <u>April/ May 2015. Nov/ Dec 2015</u> <i>Refer the Elmasri Navathe page no 570</i>	C212.3	BTL5
8	<b>Explain Deadlock in detail with an example. .</b> <u>Nov/Dec 2014 , Nov/ Dec 2015, Nov/Dec 2017</u> <i>Refer the Elmasri Navathe page no 591</i>	C212.3	BTL2
9	<b>Briefly describe two phase locking in concurrency control techniques.</b> <u>(Nov/Dec 2014)</u> <i>Refer the Elmasri Navathe page no 584</i>	C212.3	BTL1
10	<b>Explain the concepts of concurrent execution in Transaction processing system.</b> <u>(Nov/Dec 2014)</u> <i>Refer Elmasri Navathe page no.583</i>	C212.3	BTL2
11	<b>Give detail explanation about ACID Properties</b> <u>Nov/Dec 2009,April 2011</u> <i>Refer Elmasri Navathe page no.562</i>	C212.3	BTL4
12	<b>State and explain the lock based concurrency control with suitable examples?</b> <u>Nov/Dec 2017</u>	C212.3	BTL4
13	<b>Discuss the violations caused by each of the following: dirty read, non-repeatable read and phantoms with suitable example.</b> <u>April/May 2017.</u>	C212.3	BTL5
14	<b>Explain why timestamp-based concurrency control allows schedules that are not recoverable. Describe how it can be modified through buffering to disallow such schedules.</b> <u>April/May 2017.</u>	C212.3	BTL1
15	<b>How can you implement atomicity in transactions? Explain.</b>	C212.3	BTL6

**UNIT IV*****IMPLEMENTATION TECHNIQUES***

RAID – File Organization – Organization of Records in Files – Indexing and Hashing –Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for SELECT and JOIN operations – Query optimization using Heuristics and Cost Estimation

**PART – A**

Q. No.	Question	CO	Bloom's Level
1	<b>What is B-Tree?</b> <input type="checkbox"/> A B-tree eliminates the redundant storage of search-key values. <input type="checkbox"/> It allows search key values to appear only once.	C213.4	BTL1

2	<b>What is a B+-Tree index?</b> A B+-Tree index takes the form of a balanced tree in which every path from the root of the root of the tree to a leaf of the tree is of the same length	C213.4	BTL4
3	<b>What is a hash index?</b> A hash index organizes the search keys, with their associated pointers, into a hash file structure	C213.4	BTL1
4	<b>Define seek time.</b> The time for repositioning the arm is called the seek time and it increases with the distance that the arm is called the seek time.	C213.4	BTL1
5	<b>Define rotational latency time.</b> The time spent waiting for the sector to be accessed to appear under the head is called the rotational latency time.	C213.4	BTL1
6	<b>What is called mirroring?</b> The simplest approach to introducing redundancy is to duplicate every disk. This technique is called mirroring or shadowing.	C213.4	BTL1
7	<b>What are the two main goals of parallelism?</b> <input type="checkbox"/> Load –balance multiple small accesses, so that the throughput of such accesses increases. <input type="checkbox"/> Parallelize large accesses so that the response time of large accesses is reduced	C213.4	BTL1
8	<b>What is an index?</b> An index is a structure that helps to locate desired records of a relation quickly, without examining all records	C213.4	BTL1
9	<b>What are the factors to be taken into account when choosing a RAID level?</b> <input type="checkbox"/> Monetary cost of extra disk storage requirements. <input type="checkbox"/> Performance requirements in terms of number of I/O operations <input type="checkbox"/> Performance when a disk has failed and Performances during rebuild.	C213.4	BTL1
10	<b>What are the types of storage devices?</b> Primary storage, Secondary storage, Tertiary storage, Volatile storage, Nonvolatile storage	C213.4	BTL1
		C213.4	BTL1

11	<b>What is called remapping of bad sectors?</b> If the controller detects that a sector is damaged when the disk is initially formatted, or when an attempt is made to write the sector, it can logically map the sector to a different physical location.		
12	<b>Define software and hardware RAID systems? (May/June 16)</b> RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called software RAID systems and the systems with special hardware support are called hardware RAID systems.	C213.4	BTL1
13	<b>Define hot swapping?</b> Hot swapping permits the removal of faulty disks and replaces it by new ones without turning power off. Hot swapping reduces the mean time to repair.	C213.4	BTL1
14	<b>What are the ways in which the variable-length records arise in database systems?</b> 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.	C213.4	BTL1
15	<b>What are the two types of blocks in the fixed –length representation? Define them.</b> <input type="checkbox"/> Anchor block: Contains the first record of a chain. <input type="checkbox"/> Overflow block: Contains the records other than those that are the first record of a chain.	C213.4	BTL1
16	<b>What is hashing file organization?</b> In the 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.	C213.4	BTL2
17	<b>What are called index-sequential files?</b> The files that are ordered sequentially with a primary index on the search key are called index-sequential files.	C213.4	BTL2
18	<b>Define Primary index and Secondary Index</b> It is in a sequentially ordered file, the index whose search key specifies the sequential order of the file. Also called clustering index. The search key of a primary index is usually but not necessarily the primary key. It is an index whose search key specifies an order different from the sequential order of the file. Also called non clustering index.	C213.4	BTL1

19	<p><b>Give an example of a join that is not a simple equi-join for which partitioned parallelism can be used. (Nov/Dec 15)</b></p> <p><b>r JOIN (r.A = :s, B) AND (r.A &lt; :s, C)</b></p>	C213.4	BTL2
20	<p><b>Differentiate static and dynamic hashing. (Apr/May 15) (Nov/Dec 14,15)</b></p> <p><b>Static Hashing</b> In static hashing, when a search-key value is provided, the hash function always computes the same address. The number of buckets provided remains unchanged at all times i.e. fixed</p> <p>Space and overhead is more As file grows performance decreases</p> <p><b>Dynamic Hashing</b> Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand i.e. no. of buckets not fixed.</p> <p>Minimum space and less overhead Performance do not degrade as file grows</p>	C213.4	BTL1
21	<p><b>List out the mechanisms to avoid collision during hashing.(Nov/Dec 16)</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> In overflow chaining, the overflow buckets of a given bucket are chained together in a linked list.</li> <li><input type="checkbox"/> Above scheme is called closed hashing. An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.</li> </ul>	C213.4	BTL1
22	<p><b>What are the disadvantages of B-Tree over B+ Tree? (Nov/Dec 16)</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Only small fraction of all search-key values are found early</li> <li><input type="checkbox"/> Non-leaf nodes are larger. Thus, B-Trees typically have greater depth than corresponding B+-Tree</li> <li><input type="checkbox"/> Insertion and deletion more complicated than in B+-Trees</li> <li><input type="checkbox"/> Implementation is harder than B+-Trees.</li> <li><input type="checkbox"/></li> </ul>	C213.4	BTL1
23	<p><b>What is called query processing?</b> Query processing refers to the range of activities involved in extracting data from a database.</p>	C213.4	BTL1
24	<p><b>What is called a query evaluation plan?</b> A sequence of primitive operations that can be used to evaluate be</p>	C213.4	BTL1

	query is a query evaluation plan or a query execution plan.		
25	<b>Explain “Query optimization”?(May/June 16)</b> Query optimization refers to the process of finding the lowest cost method of evaluating a given query.	C213.4	BTL1
26	<b>State the need for Query Optimization. (Apr/May 15)</b> The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.	C213.4	BTL1
27	<b>What is meant by software and hardware RAID systems? May/June 2016</b> RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called software RAID systems and the systems with special hardware support are called hardware RAID systems.	C213.4	BTL1
28	<b>What is the use of RAID? April/May2009, Nov/Dec2010</b> A variety of disk-organization techniques, collectively called redundant arrays of independent disks are used to improve the performance and reliability	C213.4	BTL1
29	<b>What is known as heap file organization? Nov/Dec 2009</b> In the 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. There is a single file for each relation.	C213.4	BTL1
30	<b>What is known as sequential file organization? April/May2009</b> In the sequential file organization, the records are stored in sequential order, according to the value of a “search key” of each record.	C213.4	BTL2
31	<b>What are the types of indices?</b> Ordered indices  Hash indices	C213.4	BTL1
32	<b>What are the advantages and disadvantages of indexed sequential file? APRIL/MAY- 2011</b> The advantage of ordering records in a sequential file according to a key is that you can then search the file more quickly. If you know the key value that you want, you can use one of the relatively fast searches. The disadvantage is that when you insert, you need to rewrite at least everything after the insertion point, which makes inserts very expensive unless they are done at the end of the file. An indexed file approach keeps a (hopefully) small part of each row, and some kind of "pointer" to the row's location within the data file. This allows a search to use the index, which is ordered by the index and (again hopefully) much smaller and therefore much	C213.4	BTL1

	faster than scanning the entire data file for the indexed data.		
33	<p><b>Compare sequential access devices versus random access devices with an example</b></p> <p>sequential access devices random access devices Must be accessed from the beginning It is possible to read data from any location Eg:- tape storage Eg:-disk storage Access to data is much slower Access to data is faster Cheaper than disk Expensive when compared with disk</p>	C213.4	BTL1
34	<p><b>Explain how reliability can be improved through redundancy?</b></p> <p>The simplest approach to introducing redundancy is to duplicate every disk. This technique is called mirroring or shadowing. A logical disk then consists of two physical disks, and write is carried out on both the disk. If one of the disks fails the data can be read from the other. Data will be lost if the second disk fails before the first failed disk is repaired.</p>	C213.4	BTL1
35	<p><b>What is database tuning? APRIL/MAY-2011</b></p> <p>Database tuning describes a group of activities used to optimize and homogenize the performance of a database. It usually overlaps with query tuning, but refers to design of the database files, selection of the database management system (DBMS), operating system and CPU the DBMS runs on.</p>	C213.4	BTL1
36	<p><b>What are a block and a block number?</b></p> <p>A block is a contiguous sequence of sectors from a single track of one platter. Each request specifies the address on the disk to be referenced. That address is in the form of a block number.</p>	C213.4	BTL1
37	<p><b>What are the benefits of RAID</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Data loss can be very dangerous for an organization</li> <li><input type="checkbox"/> RAID technology prevents data loss due to disk failure</li> <li><input type="checkbox"/> RAID technology can be implemented in hardware or software</li> <li><input type="checkbox"/> Servers make use of RAID Technology</li> </ul>	C213.4	BTL1
38	<p><b>Define Bit-Interleaved Parity</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> A single parity bit is enough for error correction, not just detection, since we know which disk has failed</li> </ul>	C213.4	BTL1

	<ul style="list-style-type: none"> <li>– When writing data, corresponding parity bits must also be computed and written to a parity bit disk</li> <li>– To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)</li> <li><input type="checkbox"/> I/O operation addresses all the drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications</li> </ul>		
39	<p><b>What is Block-Interleaved Parity</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> When writing data block, corresponding block of parity bits must also be computed and written to parity disk</li> <li><input type="checkbox"/> To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks</li> </ul>	C213.4	BTL1
40	<p><b>What are the Types of Ordered Indices</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Dense index</li> <li><input type="checkbox"/> Sparse index</li> </ul>	C213.4	BTL1
41	<p><b>What is Multilevel Index</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> If primary index does not fit in memory, access becomes expensive.</li> <li><input type="checkbox"/> To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.</li> <li>– outer index – a sparse index of primary index</li> <li>– inner index – the primary index file</li> <li><input type="checkbox"/> If even outer index is too large to fit in main memory, yet another level of index can be created, and so on</li> </ul>	C213.4	BTL1
42	<p><b>Define Primary and Secondary Indices</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Secondary indices have to be dense.</li> <li><input type="checkbox"/> Indices offer substantial benefits when searching for records.</li> <li><input type="checkbox"/> When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.</li> <li><input type="checkbox"/> Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive</li> <li>– each record access may fetch a new block from disk</li> </ul>	C213.4	BTL1
43	<p><b>Draw the structure of B+-Tree Node .</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> <b>Typical node</b></li> <li>– <math>K_i</math> are the search-key values</li> <li>– <math>P_i</math> are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).</li> <li><input type="checkbox"/> The search-keys in a node are ordered</li> </ul> <p><math>K_1 &lt; K_2 &lt; K_3 &lt; \dots &lt; K_{n-1}</math></p>	C213.4	BTL1
44	<p><b>What are the Algorithms for Executing Relational Query Operations</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> An RDBMS must include one or more alternative algorithms that implement each relational algebra operation (SELECT, JOIN,...) and, in many cases, that implement each combination of these operations.</li> </ul>	C213.4	BTL1

	<ul style="list-style-type: none"> <li><input type="checkbox"/> Each algorithm may apply only to particular storage structures and access paths (such index,...).</li> <li><input type="checkbox"/> Only execution strategies that can be implemented by the RDBMS algorithms and that apply to the particular query and particular database design can be considered by the query optimization module.</li> </ul>		
45	<p><b>What are the Advantages and disadvantages of extendable hashing?</b></p> <p><b>Advantages of extendable hashing:</b></p> <ul style="list-style-type: none"> <li>– Hash performance does not degrade with growth of file</li> <li>– Minimal space overhead</li> </ul> <p><b>Disadvantages of extendable hashing</b></p> <ul style="list-style-type: none"> <li>– Extra level of indirection to find desired record</li> </ul> <p>Bucket address table may itself become very big.</p>	C213.4	BTL1
46	<p><b>What are the Cost functions for SELECT Operation</b></p> <p><b>Linear Search:</b></p> <ul style="list-style-type: none"> <li>– <math>[n\text{Blocks}(R)/2]</math>, if the record is found.</li> <li>– <math>[n\text{Blocks}(R)]</math>, if no record satisfied the condition.</li> </ul> <p><b>Binary Search :</b></p> <ul style="list-style-type: none"> <li>o <math>[\log_2(n\text{Blocks}(R))]</math>, if equality condition is on key attribute, because <math>\text{SCA}(R) = 1</math> in this case.</li> <li>o<math>[\log_2(n\text{Blocks}(R))] + [\text{SCA}(R)/b\text{Factor}(R)] - 1</math>, otherwise.</li> </ul>	C213.4	BTL1
47	<p><b>Write the Cost functions for JOIN Operation</b></p> <p>Join operation is the most time consuming operation to process.</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> An estimate for the size (number of tuples) of the file that results after the JOIN operation is required to develop reasonably accurate cost functions for JOIN operations.</li> <li><input type="checkbox"/> The JOIN operations define the relation containing tuples that satisfy a specific predicate F from the Cartesian product of two relations R and S.</li> </ul>	C213.4	BTL1
48	<p><b>Define Cost Estimation in Query Optimization</b></p> <p>The main aim of query optimization is to choose the most efficient way of implementing the relational algebra operations at the lowest possible cost.</p>	C213.4	BTL1
49	<p><b>What is External Sorting ?</b></p> <p>It refers to sorting algorithms that are suitable for large files of records on disk that do not fit entirely in main memory, such as most database files..</p>	C213.4	BTL1
50	<p><b>What are Cost Components of Query Execution ?</b></p> <p>The cost of executing the query includes the following components:</p> <ul style="list-style-type: none"> <li>– Access cost to secondary storage.</li> <li>– Storage cost.</li> <li>– Computation cost.</li> <li>– Memory uses cost.</li> <li>– Communication cost.</li> </ul>	C213.4	BTL1

**PART – B**

Q. NO.	QUESTIONS	CO	BLOOM'S LEVEL
1	Describe File Organization. <u>April/May 2009, 2011</u>	C213.4	BTL5
2	Define RAID and Briefly Explain RAID techniques. ( <i>Nov/Dec 14, 15, 16</i> ) ( <i>Apr/May 15, 16</i> )	C213.4	BTL5
3	Explain Secondary storage devices.	C213.4	BTL2
4	Explain about static and dynamic hashing with an example	C213.4	BTL5
5	Explain about Multidimensional and parallel with an example	C213.4	BTL3
6	Explain about ordered indices with an example	C213.4	BTL5
7	Explain about B <sub>+</sub> trees indexing concepts with an example ( <i>Nov/Dec 14</i> ) ( <i>May/June 16</i> )	C213.4	BTL5
8	Explain about B trees indexing concepts with an example ( <i>Nov/Dec 14</i> )	C213.4	BTL5
9	Illustrate indexing and hashing techniques with suitable examples. ( <i>Nov/Dec 15</i> )	C213.4	BTL1
10	Explain about Query optimization with neat Diagram. ( <i>Nov/Dec 14, 16</i> )	C213.4	BTL1
11	Give a detailed description about Query processing and Optimization. Explain the cost estimation of Query Optimization ( <i>Nov/Dec 14</i> ), <u>Nov/ Dec 2016</u>	C213.4	BTL1
12	Discuss about join order optimization and heuristic optimization algorithm. ( <i>Apr/May 15</i> )	C213.4	BTL1
13	Briefly explain about Query Processing review( <i>May/June 16</i> )	C213.4	BTL2

14	<b>Write Short Notes On Index Structure Of Files? <u>Nov/Dec 2014</u></b>	C213.4	BTL1
15	<b>Give detail Explanation about Hashing &amp; Types of Hashing. <u>Nov/Dec 2015</u></b>	C213.4	BTL1

## UNIT V ADVANCED TOPICS

Distributed Databases: Architecture, Data Storage, Transaction Processing – Object-based Databases: Object Database Concepts, Object-Relational features, ODMG Object Model, ODL, OQL – XML Databases: XML Hierarchical Model, DTD, XML Schema, XQuery – Information Retrieval: IR Concepts, Retrieval Models, Queries in IR systems.

Q. No.	Questions	CO	Bloom's Level
1	<b>What is homogeneous distributed database and heterogeneous distributed database</b> A homogeneous distributed database has identical software and hardware running all databases instances, and may appear through a single interface as if it were a single database. A heterogeneous distributed database may have different hardware, operating systems, database management systems, and even data models for different databases	C213.5	BTL1
2	<b>Define Distributed Database Systems. (<u>Nov/Dec 16</u>)</b> Database spread over multiple machines (also referred to as sites or nodes). Network interconnects the machines. Database shared by users on multiple machines is called Distributed Database Systems	C213.5	BTL1
3	<b>What are the types of Distributed Database</b> <input type="checkbox"/> Homogeneous Distributed DB Heterogeneous Distributed DB	C213.5	BTL2
4	<b>Define fragmentation in Distributed Database</b> The system partitions the relation into several fragments and stores each fragment at different sites Two approaches : Horizontal Fragmentation, Vertical Fragmentation	C213.5	BTL4
5		C213.5	BTL2

	<b>Define Database replication.</b> Database replication can be used on many database management systems, usually with a master/slave relationship between the original and the copies. The master logs the updates, which then ripple through to the slaves. The slave outputs a message stating that it has received the update successfully, thus allowing the sending of subsequent updates.		
6	<b>What is the advantage of OODB?</b> An integrated repository of information that is shared by multiple users, multiple products, multiple applications on multiple platforms.	C213.5	BTL2
7	<b>What is Object database System?</b> An object database is a database management system in which information is represented in the form of objects as used in object-oriented programming. Object-relational databases are a hybrid of both approaches.	C213.5	BTL1
8	<b>What are the advantages of OODB?</b> An integrated repository of information that is shared by multiple users, multiple products, multiple applications on multiple platforms. It also solves the following problems: 1. The semantic gap: The real world and the Conceptual model is very similar. 2. Impedance mismatch: Programming languages and database systems must be interfaced to solve application problems. But the language style, data structures, of a programming language (such as C) and the DBMS (such as Oracle) are different. The OODB supports general purpose programming in the OODB framework. 3. New application requirements: Especially in OA, CAD, CAM, CASE, object-orientation is the most natural and most convenient.	C213.5	BTL1
9	<b>How do you define types in object relational feature in oracle?</b> Oracle allows us to define types similar to the types of SQL. The syntax is CREATE TYPE t AS OBJECT ( list of attributes and methods );	C213.5	BTL1
10	<b>Define ODMG Object model?</b> The ODMG object model is the data model upon which the object definition language (ODL) and object query language (OQL) are based.	C213.5	BTL1
11	<b>Define ODL.</b> ODL language is used to create object specifications: <input type="checkbox"/> classes and interfaces - Using the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming language, such as C++, SMALLTALK, and JAVA	C213.5	BTL1

12	<b>Define Information Retrieval.</b> It is an activity of obtaining information resources relevant to an information need from a collection of information resources	C213.5	BTL1
13	<b>Define Relevance Ranking. (Nov/Dec 14)</b> A system in which the search engine tries to determine the theme of a site that a link is coming from.	C213.5	BTL1
14	<b>Can we have more than one constructor in a class? If yes, explain the need for such a situation. (Nov/Dec 15)</b> Yes, default constructor and constructor with parameter	C213.5	BTL1
15	<b>Define XML Database.</b> An XML database is a data persistence software system that allows data to be stored in XML format. These data can then be queried, exported and serialized into the desired format. XML databases are usually associated with document-oriented databases.	C213.5	BTL4
16	<b>Define OQL with syntax.</b> <input type="checkbox"/> Entry point to the database: needed for each query which can <input type="checkbox"/> be any named <i>persistent object</i> : <pre>class Person ( extent persons key ssn) { ... ... }  class Faculty extends Person ENTRY POINTS ( extent faculy ) { ... ... }  class Department ( extent departmet key dname){ ... ... }</pre>	C213.5	BTL1
17	<b>Define Crawling and indexing the web. (Nov/Dec 14)</b> Web Crawling is the process of search engines combing through web pages in order to properly index them. These “web crawlers” systematically crawl pages and look at the keywords contained on the page, the kind of content, all the links on the page, and then returns that information to the search engine’s server for indexing. Then they follow all the hyperlinks on the website to get to other websites. When a search engine user enters a query, the search engine will go to its index and return the most relevant search results based on the keywords in the search term. Web crawling is an automated process and provides quick, up to date data.	C213.5	BTL1

18	<p><b>How does the concept of an object in the object-oriented model differ from the concept of an entity in the entity-relationship model? (Nov/Dec 16)</b></p> <p>An entity is simply a collection of variables or data items. An object is an encapsulation of data as well as the methods (code) to operate on the data. The data members of an object are directly visible only to its methods. The outside world can gain access to the object's data only by passing pre-defined messages to it and these messages are implemented by the methods.</p>	C213.5	BTL1
19	<p><b>Is XML Hierarchical?</b></p> <p>XML documents have a hierarchical structure and can conceptually be interpreted as a tree structure, called an XML tree. XML documents must contain a root element (one that is the parent of all other elements). All elements in an XML document can contain sub elements, text and attributes.</p>	C213.5	BTL1
20	<p><b>What is DTD?</b></p> <p>A document type definition (DTD) contains a set of rules that can be used to validate an XML file. After you have created a DTD, you can edit it manually, adding declarations that define elements, attributes, entities, and notations, and how they can be used for any XML files that reference the DTD file.</p>	C213.5	BTL1
21	<p><b>What is the use of XML Schema?</b></p> <p>XML Schema is commonly known as XML Schema Definition (XSD). It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces.</p>	C213.5	BTL4
22	<p><b>What is Xpath and Xquery?</b></p> <p>XPath can be used to navigate through elements and attributes in an XML document. XPath is a syntax for defining parts of an XML document. XPath uses path expressions to navigate in XML documents. XPath contains a library of standard functions. XPath is a major element in XSLT and in XQuery.</p>	C213.5	BTL2
23	<p><b>Define Keyword Queries.</b></p> <p>Keyword-based queries are the simplest and most commonly used forms of IR queries: the user just enters keyword combinations to retrieve documents.</p>	C213.5	BTL1
24	<p><b>What are the Types of Queries in IR Systems</b></p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Keyword Queries. Boolean Queries</li> <li><input type="checkbox"/> Phrase Queries</li> <li><input type="checkbox"/> Proximity Queries</li> </ul>	C213.5	BTL1

	<input type="checkbox"/> Wildcard Queries <input type="checkbox"/> Natural Language Queries		
25	<p><b>State the steps to create DTD.</b></p> <p>Create a new DTD, complete the following steps:</p> <ol style="list-style-type: none"> <li>1. Create a project to contain the DTD if needed.</li> <li>2. In the workbench, click File &gt; New &gt; Other and select XML &gt; DTD. Click Next.</li> <li>3. Select the project or folder that will contain the DTD.</li> <li>4. In the File name field, type the name of the DTD, for example MyDTD.dtd. The name of your DTD file must end with the extension .dtd</li> <li>5. Click Next.</li> <li>6. Optional: You can use a DTD template as the basis for your new DTD file. To do so, click the Use DTD Template check box, and select the template you want to use.</li> <li>7. Click Finish.</li> </ol>	C213.5	BTL1
26	<p><b>What is the difference between Information Retrieval and DBMS. ?</b></p> <p>S.No Information Retrieval DBMS 1 Imprecise semantics Precise semantics 2 Keyword search SQL 3 Unstructured data format Structured data 4 Reads mostly. Adds document occasionally. Expects reasonable number of updates. 5 Displays page through top k results. Generates full answer.</p>	C213.5	BTL1
27	<p><b>What are Goals of Distributed Database system ?</b></p> <p><b>Reliability:</b> In distributed database system, if one system fails down or stops working for some time another system can complete the task.</p> <p><b>Availability:</b> In distributed database system reliability can be achieved even if sever fails down. Another system is available to serve the client request. <b>Performance:</b> Performance can be achieved by distributing database over different locations. So the databases are available to every location which is easy to maintain.</p>	C213.5	BTL2
28	<p><b>Distributed DBMS Architectures</b></p> <p>DDBMS architectures are generally developed depending on three parameters –</p> <p><b>Distribution</b> – It states the physical distribution of data across the different sites.</p> <p><b>Autonomy</b> – It indicates the distribution of control of the database system and the degree to which each constituent DBMS can operate independently.</p> <p><b>Heterogeneity</b> – It refers to the uniformity or dissimilarity of the data models, system components and databases.</p>	C213.5	BTL1
29	<p><b>What are Architectural Models of DDBMS.</b></p> <p>Some of the common architectural models are –</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Client - Server Architecture for DDBMS</li> <li><input type="checkbox"/> Peer - to - Peer Architecture for DDBMS</li> </ul>	C213.5	BTL2

	<input type="checkbox"/> Multi - DBMS Architecture		
30	<p><b>What is Client - Server Architecture for DDBMS</b>  This is a two-level architecture where the functionality is divided into servers and clients. The server functions primarily encompass data management, query processing, optimization and transaction management. Client functions include mainly user interface. However, they have some functions like consistency checking and transaction management.</p> <p>The two different client - server architecture are –</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Single Server Multiple Client</li> <li><input type="checkbox"/> Multiple Server Multiple Client (</li> </ul>	C213.5	BTL2
31	<p><b>Write short notes on Peer- to-Peer Architecture for DDBMS</b>  In these systems, each peer acts both as a client and a server for imparting database services. The peers share their resource with other peers and co-ordinate their activities.</p> <p>This architecture generally has four levels of schemas –</p> <p><b>Global Conceptual Schema</b> – Depicts the global logical view of data.</p> <p><b>Local Conceptual Schema</b> – Depicts logical data organization at each site.</p> <p><b>Local Internal Schema</b> – Depicts physical data organization at each site.</p> <p><b>External Schema</b> – Depicts user view of data</p>	C213.5	BTL2
32	<p><b>Write short notes on Multi - DBMS Architectures</b>  This is an integrated database system formed by a collection of two or more autonomous database systems.</p> <p>Multi-DBMS can be expressed through six levels of schemas –</p> <p><b>Multi-database View Level</b> – Depicts multiple user views comprising of subsets of the integrated distributed database.</p> <p><b>Multi-database Conceptual Level</b> – Depicts integrated multi-database that comprises of global logical multi-database structure definitions.</p> <p><b>Multi-database Internal Level</b> – Depicts the data distribution across different sites and multi-database to local data mapping.</p> <p><b>Local database View Level</b> – Depicts public view of local data.</p> <p><b>Local database Conceptual Level</b> – Depicts local data organization at each site.</p> <p><b>Local database Internal Level</b> – Depicts physical data organization at each site.</p> <p>There are two design alternatives for multi-DBMS –</p> <ul style="list-style-type: none"> <li><input type="checkbox"/> Model with multi-database conceptual level.</li> <li><input type="checkbox"/> Model without multi-database conceptual level.</li> </ul>	C213.5	BTL1
33	<p><b>Define Replication and Fragmentation.</b></p> <p><b>Replication.</b> The system maintains several identical replicas of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r.</p> <p><b>Fragmentation.</b> The system partitions the relation into several fragments, and stores each fragment at a different site.</p>	C213.5	BTL1
34	<p><b>What are the advantages and disadvantages to replication.</b></p> <p><b>Availability</b> If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r, despite the failure of one site.</p> <p><b>Increased parallelism.</b> In the case where the majority of accesses to the relation r</p>	C213.5	BTL1

	<p>result in only the reading of the relation, then several sites can process queries involving r in parallel. The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites. <b>Increased overhead on update.</b> The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever r is updated, the update must be propagated to all sites containing replicas. The result is increased overhead. For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account agrees in all sites.</p>		
35	<p><b>Define Transparency</b> The user of a distributed database system should not be required to know where the data are physically located nor how the data can be accessed at the specific local site. This characteristic, called <b>data transparency</b>, can take several forms:</p>	C213.5	BTL1
36	<p><b>What are the DISTRIBUTED TRANSACTIONS</b> There are two types of transaction that we need to consider.  <input type="checkbox"/> <b>Local transactions</b> are those that access and update data in only one local database;  <input type="checkbox"/> <b>Global transactions</b> are those that access and update data in several local databases</p>	C213.5	BTL1
37	<p><b>Write the System Failure Modes</b>  <input type="checkbox"/> Failure of a site.  <input type="checkbox"/> Loss of messages.  <input type="checkbox"/> Failure of a communication link.  <input type="checkbox"/> Network partition</p>	C213.5	BTL1
38	<p><b>Define Complex Data Types ..</b> Traditional database applications have conceptually simple datatypes. The basic data items are records that are fairly small and whose fields are atomic.</p>	C213.5	BTL1
39	<p><b>What is Structured Type?</b> Structured types allow composite attributes of E-R designs to be represented directly. For instance, we can define the following structured type to represent a composite attribute name with component attribute <i>firstname</i> and <i>lastname</i>:</p>	C213.5	BTL1
40	<p><b>Write the Object-Identity and Reference Types in SQL</b> Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL we can define a type <i>Department</i> with a field <i>name</i> and a field <i>head</i> that is a reference to the type <i>Person</i>, and a table <i>departments</i> of type <i>Department</i>, as follows:  <i>create type Department (</i>  <i>name varchar(20),</i>  <i>head ref(Person) scope people);</i>  <i>create table departments of Department;</i> Here, the reference is restricted to tuples of the table <i>people</i>.</p>	C213.5	BTL1
41	<p><b>What are the Object-relational Features ?</b> Object-relational database systems are basically extensions of existing relational database systems. Changes are clearly required at many levels of the database system. However, to minimize changes to the storage-system code (relation storage, indices, etc.), the complex datatypes supported by object-relational</p>	C213.5	BTL1

	systems can be translated to the simpler type system of relational databases. Sub tables can be stored in an efficient manner		
42	<b>Define Objects and Literals .</b> Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an <b>object identifier</b> and a <b>state</b> (or current value), whereas a literal has a value (state) but <b>no object identifier</b> . In either case, the value can have a complex <b>structure</b> . The object state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, but it does not change.	C213.5	BTL1
43	<b>What re the aspects of an object?</b> <b>An object has five aspects:</b> identifier, name, lifetime, structure, and creation.	C213.5	BTL1
44	<b>What rae the types of literals?</b> <b>There are three types of literals:</b> atomic, structured, and collection.	C213.5	BTL1
45	What are the notation of ODMG?? The notation of ODMG uses three concepts: <i>interface, literal, and class</i>	C213.5	BTL1
46	<b>Define ODL: OBJECT DEFINITION LANGUAGE</b> Object Definition Language (ODL) is the specification language defining the interface to object types conforming to the ODMG Object Model. Often abbreviated by the acronym ODL. This language's purpose is to define the structure of an Entity-relationship diagram.	C213.5	BTL1
47	<b>How to declare the class and element.?</b> <b>Class Declarations</b> <input type="checkbox"/> interface < name > {elements = attributes, relationships, methods }  <b>Element Declarations</b> <input type="checkbox"/> attribute < type > < name > ; <input type="checkbox"/> relationship < rangetype > < name > ;	C213.5	BTL1
48	<b>Write the Similarities between E/R and ODL</b> <input type="checkbox"/> both support all multiplicities of relationships <input type="checkbox"/> both support inheritance	C213.5	BTL1
49	<b>Define XML SCHEMA</b> XML Schema defines a number of built-in types such as string, integer, decimal date, and boolean. In addition, it allows user-defined types; these may be simple types with added restrictions, or complex types constructed using constructors such as complex Type and sequence.	C213.5	BTL1
50	<b>What are the RETRIEVAL MODELS</b> There are the three main statistical models—Boolean, vector space, and probabilistic—and the semantic model.	C213.5	BTL1

**PART – B**

Q. NO.	QUESTIONS	CO	BLOOM'S LEVEL
1	Explain about Object Oriented Databases and XM Databases.	C213.5	BTL5
2	Explain in detail (i) Information Retrieval (iii) Transaction processing <i>(Nov/Dec 14)</i>	C213.5	BTL5
3	Write short notes on Distributed Transactions. <i>(Nov/Dec 14)</i>	C213.5	BTL6
4	Explain in detail the Client - Server Architecture for DDBMS	C213.5	BTL5
5	Suppose an Object Oriented database had an object A, which references object B, which in turn references object C. Assume all objects are on disk initially? Suppose a program first dereferences A, then dereferences B by following the reference from A, and then finally dereferences C. Show the objects that are represented in memory after each dereference, along with their state. <i>(Nov/Dec 15)</i>	C213.5	BTL3
6	Suppose that you have been hired as a consultant to choose a database system for your client's application. For each of the following applications, state what type of database system (relational, persistent programming language-based OODB, object relational; do not specify a commercial product) you would recommend. Justify your recommendation. (i)A computer-aided design system for a manufacturer of airplanes. (ii)A system to track contributions made to candidates for public office. (iii)An information system to support the making of movies. <i>(Nov/Dec 16)</i>	C213.5	BTL3
7	Give the DTD for an XML representation of the following nested-relational schema $Emp = (ename, ChildrenSet \text{ setof}(Children), SkillsSet \text{ setof}(Skills))$ $Children = (name, Birthday)$ $Birthday = (day, month, year)$ $Skills = (type, ExamsSet \text{ setof}(Exams))$ . $Exams = (year, city)$ <i>(Nov/Dec 16)</i>	C213.5	BTL3
8	Explain XML Schema with an example.	C213.5	BTL5
9	Explain various queries in IR Systems with an example.	C213.5	BTL5

10	Explain ODL and OQL with an example.	C213.5	BTL5
11	Explain ODMG – Object Model in detail	C213.5	BTL5
12	Explain XML Databases.	C213.5	BTL5
13	Explain the object relational database features.	C213.5	BTL5
14	Explain Information Retrieval process in detail	C213.5	BTL5
15	Give the DTD or XML Schema for an XML representation of the nested relational schema . <u>Nov/ Dec 2016</u>	C213.5	BTL5

**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**

**INTRODUCTION****DATABASE**

Database is collection of data which is related by some aspect. Data is collection of facts and figures which can be processed to produce information. Mostly data represents recordable facts. Data aids in producing information which is based on facts. A database management system stores data, in such a way which is easier to retrieve, manipulate and helps to produce information.

So a database is a collection of related data that we can use for

- Defining - specifying types of data
- Constructing - storing & populating
- Manipulating - querying, updating, reporting

**DISADVANTAGES OF FILE SYSTEM OVER DB**

In the early days, File-Processing system is used to store records. It uses various files for storing the records.

Drawbacks of using file systems to store data:

- Data redundancy and inconsistency  
-Multiple file formats, duplication of information in different files
- Difficulty in accessing data  
Need to write a new program to carry out each new task
- Data isolation — multiple files and formats
- Integrity problems  
- Hard to add new constraints or change existing ones
- Atomicity problem  
-Failures may leave database in an inconsistent state with partial updates carried  
Out. E.g. transfer of funds from one account to another should either complete or not happen at all
- Concurrent access anomalies  
- Concurrent accessed needed for performance
- Security problems

Database systems offer solutions to all the above problems

**PURPOSE OF DATABASE SYSTEM**

The typical 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. A file processing system has a number of major disadvantages.

- Data redundancy and inconsistency
- Difficulty in accessing data
- Data isolation – multiple files and formats
- Integrity problems
- Atomicity of updates
- Concurrent access by multiple users
- Security problems

**1.Data redundancy and inconsistency:**

In file processing, every user group maintains its own files for handling its data processing applications.

**Example:**

Consider the UNIVERSITY database. Here, two groups of users might be the course registration personnel and the accounting office. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Storing the same data multiple times is called data redundancy. This redundancy leads to several problems.

• Need to perform a single logical update multiple times.

• Storage space is wasted.

• Files that represent the same data may become inconsistent.

Data inconsistency is the various copies of the same data may no longer Agree. **Example:**

One user group may enter a student's birth date erroneously as JAN-19-1984, whereas the other user groups may enter the correct value of JAN-29-1984.

**2. Difficulty in accessing data**

File processing environments do not allow needed data to be retrieved in a convenient and efficient manner.

**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. **Example:**

The balance of certain types of bank accounts may never fall below a prescribed amount. Developers enforce these constraints in the system by addition appropriate code in the various application programs

**5. Atomicity problems**

Atomic means the transaction must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file processing system.

**Example:**

Consider a program to transfer \$50 from account A to account B. If a system failure occurs during the execution of the program, it is possible that the \$50 was removed from account A but was not credited to account B, resulting in an inconsistent database state.

**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. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to

provide because data may be accessed by many different application programs that have not been coordinated previously.

**Example:** When several reservation clerks try to assign a seat on an airline flight, the system should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger.

**7. Security problems**

Enforcing security constraints to the file processing system is difficult.

**APPLICATION OF DATABASE****Database Applications**

- Banking: all transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions
- Telecommunication: Call History, Billing
- Credit card transactions: Purchase details, Statements

**VIEWS OF DATA**

It refers that how database is actually stored in database, what data and structure of data used by database for data. So describe all this database provides user with views and these are

- **Data abstraction**
- **Instances and schemas**

**Data abstraction**

As a data in database are stored with very complex data structure so when user come and want to access any data, he will not be able to access data if he has go through this data structure. So to simplify the interaction of user and database, DBMS hides some information which is not of user interest, a this is called data abstraction:- **So developer hides complexity from user and store abstract view of data.**

Data abstraction has three level of abstractions

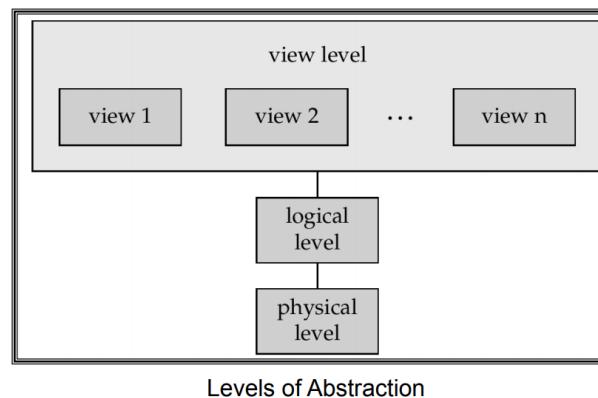
- **level / internal level**
- **Logical level / conceptual level**
- **view level / external level**

**Physical level:-** this is the lowest level of data abstraction which describe How data is actual stored in database. This level basically describe the data structure and access path /indexing use for accessing file.

**Logical level:-** The next level of abstraction describe what data are stored in the database and what are the relationship existed among those of data.

**View level:-** In this level user only interact with database and the complexity remain unview . user see data and there may be many views of one data like chart and graph.

## View of Data

**DATA MODELS IN DBMS**

A **Data Model** is a logical structure of Database. It describes the design of database to reflect entities, attributes, relationship among data, constraints etc.

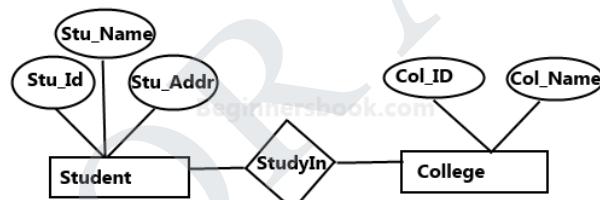
**Types of Data Models:**

**Object based logical Models** – Describe data at the conceptual and view levels.

## 1. E-R Model

An **entity–relationship model (ER model)** is a systematic way of describing and defining a business process. An ER model is typically implemented as a database. The main components of E-R model are: entity set and relationship set.

A sample E-R Diagram:



Sample E-R Diagram

## 2. Object oriented Model

An object data model is a data model based on object-oriented programming, associating methods (procedures) with objects that can benefit from class hierarchies. Thus, “objects” are levels of abstraction that include attributes and behavior

**Record based logical Models** – Like Object based model, they also describe data at the conceptual and view levels. These models specify logical structure of database with records, fields and attributes.

## 1. Relational Model

In relational model, the data and relationships are represented by collection of inter-related tables. Each table is a group of column and rows, where column represents attribute of an entity and rows represents records.

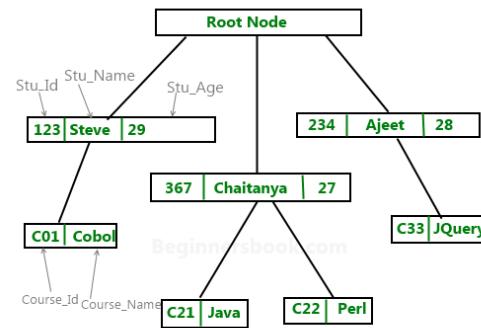
Sample relationship Model: Student table with 3 columns and three records.

Stu_Id	Stu_Name	Stu_Age
111	Ashish	23
123	Saurav	22
169	Lester	24

## 2. Hierarchical Model

In hierarchical model, data is organized into a tree like structure with each record is having one parent record and many children. The main drawback of this model is that, it can have only one to many relationships between nodes.

Sample Hierarchical Model Diagram:



3. **Network Model** – Network Model is same as hierarchical model except that it has graph-like structure rather than a tree-based structure. Unlike hierarchical model, this model allows each record to have more than one parent record.

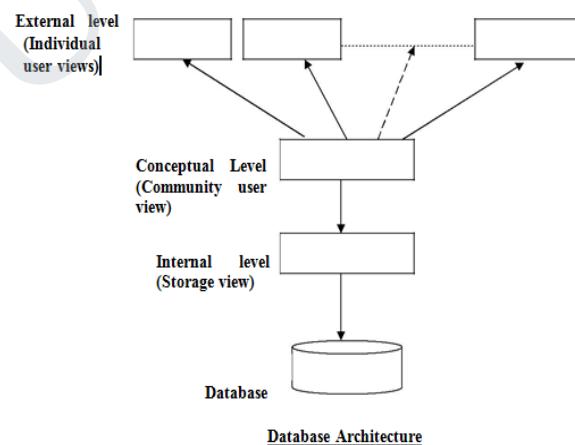
Physical Data Models – These models describe data at the lowest level of abstraction.

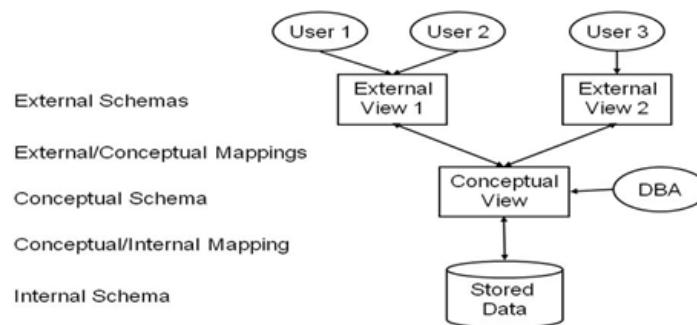
### Three Schema Architecture

The goal of the three schema architecture is to separate the user applications and the physical database. The schemas can be defined at the following levels:

1. **The internal level** – has an internal schema which describes the physical storage structure of the database. Uses a physical data model and describes the complete details of data storage and access paths for the database.
2. **The conceptual level** – has a conceptual schema which describes the structure of the database for users. It hides the details of the physical storage structures, and concentrates on describing entities, data types, relationships, user operations and constraints. Usually a representational data model is used to describe the conceptual schema.
3. **The External or View level** – includes external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. Represented using the representational data model.

The three schema architecture is used to visualize the schema levels in a database. The three schemas are only descriptions of data, the data only actually exists is at the physical level.





## COMPONENTS OF DBMS

### Database Users

Users are differentiated by the way they expect to interact with the system

- Application programmers
- Sophisticated users
- Naïve users
- Database Administrator
- Specialized users etc.,

### Application programmers:

Professionals who write application programs and using these application programs they interact with the database system

### Sophisticated users :

These user interact with the database system without writing programs, But they submit queries to retrieve the information

### Specialized users:

Who write specialized database applications to interact with the database system.

### Naïve users:

Interacts with the database system by invoking some application programs that have been written previously by application programmers

Eg : people accessing database over the web

### Database Administrator:

Coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.

- Schema definition
- Access method definition
- Schema and physical organization modification
- Granting user authority to access the database
- Monitoring performance

### Storage Manager

The Storage Manager include these following components/modules

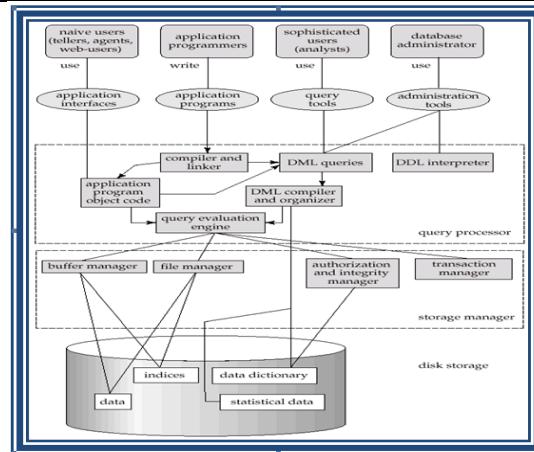
- Authorization Manager
- Transaction Manager
- File Manager
- Buffer Manager
- ❖ Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- ❖ **The storage manager is responsible to the following tasks:**
  - interaction with the file manager
  - efficient storing, retrieving and updating of data

#### Authorization Manager

- Checks whether the user is an authorized person or not
- Test the satisfaction of integrity constraints

#### Transaction Manager

Responsible for concurrent transaction execution It ensures that the database remains in a consistent state despite of the system failure



## EVOLUTION OF RDBMS

Before the acceptance of Codd's Relational Model, database management systems was just an ad hoc collection of data designed to solve a particular type of problem, later extended to solve more basic purposes. This led to complex systems, which were difficult to understand, install, maintain and use. These database systems were plagued with the following problems:

- They required large budgets and staffs of people with special skills that were in short supply.
- Database administrators' staff and application developers required prior preparation to access these database systems.
- End-user access to the data was rarely provided.
- These database systems did not support the implementation of business logic as a DBMS responsibility.

Hence, the objective of developing a relational model was to address each and every one of the shortcomings that plagued those systems that existed at the end of the 1960s decade, and make DBMS products more widely appealing to all kinds of users.

The existing relational database management systems offer powerful, yet simple solutions for a wide variety of commercial and scientific application problems. Almost every industry uses relational systems to store, update and retrieve data for operational, transaction, as well as decision support systems.

## RELATIONAL DATABASE

A relational database is a database system in which the database is organized and accessed according to the relationships between data items without the need for any consideration of physical orientation and relationship. Relationships between data items are expressed by means of **tables**.

It is a tool, which can help you store, manage and disseminate information of various kinds. It is a collection of objects, tables, queries, forms, reports, and macros, all stored in a computer program all of which are inter-related.

It is a method of structuring data in the form of records, so that relations between different entities and attributes can be used for data access and transformation.

## RELATIONAL DATABASE MANAGEMENT SYSTEM

A Relational Database Management System (RDBMS) is a system, which allows us to perceive data as tables (and nothing but tables), and **operators** necessary to manipulate that data are at the user's disposal.

### Features of an RDBMS

The features of a relational database are as follows:

- The ability to create multiple relations (tables) and enter data into them
- An interactive query language
- Retrieval of information stored in more than one table
- Provides a *Catalog* or *Dictionary*, which itself consists of tables ( called *system tables* )

## Basic Relational Database Terminology

### Catalog:

A catalog consists of all the information of the various schemas (external, conceptual and internal) and also all of the corresponding mappings (external/conceptual, conceptual/internal).

It contains detailed information regarding the various objects that are of interest to the system itself; e.g., tables, views, indexes, users, integrity rules, security rules, etc.

In a relational database, the entities of the ERD are represented as *tables* and their attributes as the *columns* of their respective tables in a database schema.

It includes some important terms, such as:

- *Table*: Tables are the basic storage structures of a database where data about something in the real world is

stored. It is also called a *relation or an entity*.

- **Row:** Rows represent collection of data required for a particular entity. In order to identify each row as unique there should be a *unique identifier* called the *primary key*, which allows no duplicate rows. For example in a library every member is unique and hence is given a membership number, which uniquely identifies each member. A row is also called a *record* or a *tuple*.
- **Column:** Columns represent characteristics or attributes of an entity. Each attribute maps onto a column of a table. Hence, a column is also known as an *attribute*.
- **Relationship:** Relationships represent a logical link between two tables. A relationship is depicted by a *foreign key* column.
- Degree: number of attributes
- Cardinality: number of tuples
- An attribute of an entity has a particular value. The set of possible values that a given attribute can have is called its *domain*.

## KEYS AND THEIR USE

**Key:** An attribute or set of attributes whose values uniquely identify each entity in an entity set is called a key for that entity set.

**Super Key:** If we add additional attributes to a key, the resulting combination would still uniquely identify an instance of the entity set. Such augmented keys are called super keys.

**Primary Key:** It is a minimum super key.

It is a *unique identifier for the table* (a column or a column combination with the property that at any given time no two rows of the table contain the same value in that column or column combination).

**Foreign Key:** A *foreign key* is a field (or collection of fields) in one table that uniquely identifies a row of another table. In simpler words, the foreign key is defined in a second table, but it refers to the primary key in the first table.

**Candidate Key:** There may be two or more attributes or combinations of attributes that uniquely identify an instance of an entity set. These attributes or combinations of attributes are called candidate keys.

**Secondary Key:** A secondary key is an attribute or combination of attributes that may not be a candidate key, but that classifies the entity set on a particular characteristic. Any key consisting of a single attribute is called a **simple key**, while that consisting of a combination of attributes is called a **composite key**.

## Referential Integrity

Referential Integrity can be defined as an integrity constraint that specifies that the value (or existence) of an attribute in one relation depend on the value (or existence) of an attribute in the same or another relation. Referential integrity in a relational database is consistency between coupled tables. It is usually enforced by the combination of a primary key and a foreign key. For referential integrity to hold, any field in a table that is declared a foreign key can contain only values from a parent table's primary key field. For instance, deleting a record that contains a value referred to by a foreign key in another table would break referential integrity.

## Relational Model

Relational data model is the primary data model, which is used widely around the world for data storage and processing. This model is simple and it has all the properties and capabilities required to process data with storage efficiency.

### Concepts

**Tables** – In relational data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represent records and columns represent the attributes.

**Tuple** – A single row of a table, which contains a single record for that relation is called a tuple.

**Relation instance** – A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

**Relation schema** – A relation schema describes the relation name (table name), attributes, and their names.

**Relation key** – Each row has one or more attributes, known as relation key, which can identify the row in the relation (table) uniquely.

**Attribute domain** – Every attribute has some pre-defined value scope, known as attribute domain.

### Constraints

Every relation has some conditions that must hold for it to be a valid relation. These conditions are called **Relational Integrity Constraints**. There are three main integrity constraints –

- Key constraints
- Domain constraints
- Referential integrity constraints

### Key Constraints

There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called **key** for that relation. If there are more than one such minimal subsets, these are called **candidate keys**.

Key constraints force that –

- in a relation with a key attribute, no two tuples can have identical values for key attributes.
- a key attribute can not have NULL values.

Key constraints are also referred to as Entity Constraints.

#### Domain Constraints

Attributes have specific values in real-world scenario. For example, age can only be a positive integer. The same constraints have been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age cannot be less than zero and telephone numbers cannot contain a digit outside 0-9.

#### Referential integrity Constraints

Referential integrity constraints work on the concept of Foreign Keys. A foreign key is a key attribute of a relation that can be referred in other relation.

Referential integrity constraint states that if a relation refers to a key attribute of a different or same relation, then that key element must exist.

Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances. There are two kinds of query languages – relational algebra and relational calculus.

### **Relational Algebra**

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows –

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

We will discuss all these operations in the following sections.

#### **Select Operation ( $\sigma$ )**

It selects tuples that satisfy the given predicate from a relation.

##### **Notation – $\sigma_p(r)$**

Where  $\sigma$  stands for selection predicate and  $r$  stands for relation.  $p$  is prepositional logic formula which may use connectors like **and**, **or**, and **not**. These terms may use relational operators like  $=$ ,  $\neq$ ,  $\geq$ ,  $<$ ,  $>$ ,  $\leq$ .

#### **For example –**

$\sigma_{\text{subject} = \text{"database"}}(\text{Books})$

**Output** – Selects tuples from books where subject is 'database'.

$\sigma_{\text{subject} = \text{"database"} \text{ and } \text{price} = \text{"450"}}(\text{Books})$

**Output** – Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{\text{subject} = \text{"database"} \text{ and } \text{price} = \text{"450"} \text{ or } \text{year} > \text{"2010"}}(\text{Books})$

**Output** – Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

#### **Project Operation ( $\Pi$ )**

It projects column(s) that satisfy a given predicate.

##### **Notation – $\Pi_{A_1, A_2, A_n}(r)$**

Where  $A_1, A_2, A_n$  are attribute names of relation  $r$ .

Duplicate rows are automatically eliminated, as relation is a set.

#### **For example –**

$\Pi_{\text{subject, author}}(\text{Books})$

Selects and projects columns named as subject and author from the relation Books.

#### **Union Operation ( $\cup$ )**

It performs binary union between two given relations and is defined as –

$r \cup s = \{ t \mid t \in r \text{ or } t \in s \}$

#### **Notation – $r \cup s$**

Where  $r$  and  $s$  are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold –

- $r$ , and  $s$  must have the same number of attributes.
- Attribute domains must be compatible.

- Duplicate tuples are automatically eliminated.

$\prod_{\text{author}} (\text{Books}) \cup \prod_{\text{author}} (\text{Articles})$

**Output** – Projects the names of the authors who have either written a book or an article or both.

*Set Difference (-)*

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

**Notation – r – s**

Finds all the tuples that are present in **r** but not in **s**.

$\prod_{\text{author}} (\text{Books}) - \prod_{\text{author}} (\text{Articles})$

**Output** – Provides the name of authors who have written books but not articles.

*Cartesian Product (X)*

Combines information of two different relations into one.

**Notation – r X s**

Where **r** and **s** are relations and their output will be defined as –

$$r X s = \{ q t | q \in r \text{ and } t \in s \}$$

$\sigma_{\text{author} = \text{'tutorialspoint'}}(\text{Books} X \text{Articles})$

**Output** – Yields a relation, which shows all the books and articles written by tutorialspoint.

*Rename Operation ( $\rho$ )*

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter **rho**  $\rho$ .

**Notation –  $\rho_x(E)$**

Where the result of expression **E** is saved with name of **x**.

Additional operations are –

- Set intersection
- Assignment
- Natural join

### SQL FUNDAMENTALS:

SQL is a standard computer language for accessing and manipulating databases.

#### What is SQL?

- SQL stands for Structured Query Language
- SQL allows you to access a database
- SQL is an ANSI standard computer language
- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert new records in a database
- SQL can delete records from a database
- SQL can update records in a database
- SQL is easy to learn

#### SQL is a Standard - BUT....

SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems. SQL statements are used to retrieve and update data in a database. SQL works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, etc. Unfortunately, there are many different versions of the SQL language, but to be in compliance with the ANSI standard, they must support the same major keywords in a similar manner (such as SELECT, UPDATE, DELETE, INSERT, WHERE, and others).

**Note:** Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard!

#### SQL Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data. Below is an example of a table called "Persons":

Last Name	First Name	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

The table above contains three records (one for each person) and four columns (LastName, FirstName, Address, and City).

**SQL Queries**

With SQL, we can query a database and have a result set returned.

A query like this:

```
SELECT LastName FROM Persons
```

Gives a result set like this:

LastName
Hansen
Svendson
Pettersen

**Note:** Some database systems require a semicolon at the end of the SQL statement. We don't use the semicolon in our tutorials.

**SQL Data Manipulation Language (DML)**

SQL (Structured Query Language) is a syntax for executing queries. But the SQL language also includes a syntax to update, insert, and delete records.

These query and update commands together form the Data Manipulation Language (DML) part of SQL:

- **SELECT** - extracts data from a database table
- **UPDATE** - updates data in a database table
- **DELETE** - deletes data from a database table
- **INSERT INTO** - inserts new data into a database table

**SQL Data Definition Language (DDL)**

The Data Definition Language (DDL) part of SQL permits database tables to be created or deleted. We can also define indexes (keys), specify links between tables, and impose constraints between database tables.

The most important DDL statements in SQL are:

- **CREATE TABLE** - creates a new database table
- **ALTER TABLE** - alters (changes) a database table
- **DROP TABLE** - deletes a database table
- **CREATE INDEX** - creates an index (search key)
- **DROP INDEX** - deletes an index

**The SQL SELECT Statement**

The SELECT statement is used to select data from a table. The tabular result is stored in a result table (called the result-set).

Syntax

```
SELECT column_name(s)
FROM table_name
```

 **Note:** SQL statements are not case sensitive. SELECT is the same as select.

**SQL SELECT Example**

To select the content of columns named "LastName" and "FirstName", from the database table called "Persons", use a SELECT statement like this:

```
SELECT LastName, FirstName FROM Persons
```

**The database table "Persons":**

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

**The result**

LastName	FirstName
Hansen	Ola
Svendson	Tove
Pettersen	Kari

**Select All Columns**

To select all columns from the "Persons" table, use a \* symbol instead of column names, like this:

```
SELECT * FROM Persons
```

**Result**

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

**The Result Set**

The result from a SQL query is stored in a result-set. Most database software systems allow navigation of the result set with programming functions, like: Move-To-First-Record, Get-Record-Content, Move-To-Next-Record, etc. Programming functions like these are not a part of this tutorial. To learn about accessing data with function calls,

**Semicolon after SQL Statements?**

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

Some SQL tutorials end each SQL statement with a semicolon. Is this necessary? We are using MS Access and SQL Server 2000 and we do not have to put a semicolon after each SQL statement, but some database programs force you to use it.

**The SELECT DISTINCT Statement**

The DISTINCT keyword is used to return only distinct (different) values.

The SELECT statement returns information from table columns. But what if we only want to select distinct elements?

With SQL, all we need to do is to add a DISTINCT keyword to the SELECT statement:

**Syntax**

```
SELECT DISTINCT column_name(s)
FROM table_name
```

**Using the DISTINCT keyword**

To select ALL values from the column named "Company" we use a SELECT statement like this:

```
SELECT Company FROM Orders
```

**"Orders" table**

Company	OrderNumber
Sega	3412
W3Schools	2312
Trio	4678
W3Schools	6798

**Result**

Company
Sega
W3Schools
Trio
W3Schools

Note that "W3Schools" is listed twice in the result-set.

To select only DIFFERENT values from the column named "Company" we use a SELECT DISTINCT statement like this:

```
SELECT DISTINCT Company FROM Orders
```

**Result:**

Company
Sega
W3Schools
Trio

Now "W3Schools" is listed only once in the result-set.

The WHERE clause is used to specify a selection criterion.

### ***The WHERE Clause***

To conditionally select data from a table, a WHERE clause can be added to the SELECT statement.

#### **Syntax**

```
SELECT column FROM table
WHERE column operator value
```

With the WHERE clause, the following operators can be used:

Operator	Description
=	Equal
$\neq$	Not equal
>	Greater than
<	Less than
$\geq$	Greater than or equal
$\leq$	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern

**Note:** In some versions of SQL the  $\neq$  operator may be written as !=

### ***Using the WHERE Clause***

To select only the persons living in the city "Sandnes", we add a WHERE clause to the SELECT statement:

```
SELECT * FROM Persons
WHERE City='Sandnes'
```

#### **"Persons" table**

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980
Pettersen	Kari	Storgt 20	Stavanger	1960

#### **Result**

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980

### ***Using Quotes***

Note that we have used single quotes around the conditional values in the examples.

SQL uses single quotes around text values (most database systems will also accept double quotes). Numeric values should not be enclosed in quotes.

For text values:

This is correct:

```
SELECT * FROM Persons WHERE FirstName='Tove'
```

This is wrong:

```
SELECT * FROM Persons WHERE FirstName=Tove
```

For numeric values:

This is correct:

```
SELECT * FROM Persons WHERE Year>1965
```

This is wrong:

```
SELECT * FROM Persons WHERE Year>'1965'
```

### ***The LIKE Condition***

The LIKE condition is used to specify a search for a pattern in a column.

#### **Syntax**

```
SELECT column FROM table
WHERE column LIKE pattern
```

A "%" sign can be used to define wildcards (missing letters in the pattern) both before and after the pattern.

### **Using LIKE**

The following SQL statement will return persons with first names that start with an 'O':

```
SELECT * FROM Persons
WHERE FirstName LIKE 'O%'
```

The following SQL statement will return persons with first names that end with an 'a':

```
SELECT * FROM Persons
WHERE FirstName LIKE '%a'
```

The following SQL statement will return persons with first names that contain the pattern 'la':

```
SELECT * FROM Persons
WHERE FirstName LIKE '%la%'
```

### **The INSERT INTO Statement**

The INSERT INTO statement is used to insert new rows into a table.

#### **Syntax**

```
INSERT INTO table_name
VALUES (value1, value2,...)
```

You can also specify the columns for which you want to insert data:

```
INSERT INTO table_name (column1, column2,...)
VALUES (value1, value2,...)
```

### **Insert a New Row**

This "Persons" table:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger

And this SQL statement:

```
INSERT INTO Persons
VALUES ('Hetland', 'Camilla', 'Hagabakka 24', 'Sandnes')
```

Will give this result:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes

### **Insert Data in Specified Columns**

This "Persons" table:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes

And This SQL statement:

```
INSERT INTO Persons (LastName, Address)
VALUES ('Rasmussen', 'Storgt 67')
```

Will give this result:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen		Storgt 67	

Null (no value ...not space not empty)

The UPDATE statement is used to modify the data in a table.

#### **Syntax**

```
UPDATE table_name
SET column_name = new_value
WHERE column_name = some_value
```

**Person:**

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen		Storgt 67	

**Update one Column in a Row**

We want to add a first name to the person with a last name of "Rasmussen":

```
UPDATE Person SET FirstName = 'Nina'
WHERE LastName = 'Rasmussen'
```

**Result:**

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen	Nina	Storgt 67	

**Update several Columns in a Row**

We want to change the address and add the name of the city:

```
UPDATE Person
SET Address = 'Stien 12', City = 'Stavanger'
WHERE LastName = 'Rasmussen'
```

**Result:**

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen	Nina	Stien 12	Stavanger

**The DELETE Statement**

The DELETE statement is used to delete rows in a table.

**Syntax**

```
DELETE FROM table_name
WHERE column_name = some_value
```

**Person:**

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen	Nina	Stien 12	Stavanger

Delete

Drop

**Delete a Row**

"Nina Rasmussen" is going to be deleted:

```
DELETE FROM Person WHERE LastName = 'Rasmussen'
```

**Result**

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger

**Delete All Rows**

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name
```

or

```
DELETE * FROM table_name
```

The ORDER BY keyword is used to sort the result.

**Sort the Rows**

The ORDER BY clause is used to sort the rows.

**Orders:**

Company	OrderNumber
Sega	3412
ABC Shop	5678
W3Schools	2312
W3Schools	6798

**Example**

To display the company names in alphabetical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company ASC (ascending)
```

**Result:**

Company	OrderNumber
ABC Shop	5678
Sega	3412
W3Schools	6798
W3Schools	2312

**Example**

To display the company names in alphabetical order AND the OrderNumber in numerical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company, OrderNumber
```

**Result:**

Company	OrderNumber
ABC Shop	5678
Sega	3412
W3Schools	2312
W3Schools	6798

**Aggregate functions**

Aggregate functions operate against a collection of values, but return a single value.

**Note:** If used among many other expressions in the item list of a SELECT statement, the SELECT must have a GROUP BY clause!!

**"Persons" table (used in most examples)**

Name	Age
Hansen, Ola	34
Svendson, Tove	45
Pettersen, Kari	19

**Aggregate functions in MS Access**

Function	Description
AVG(column)	Returns the average value of a column
COUNT(column)	Returns the number of rows (without a NULL value) of a column
COUNT(*)	Returns the number of selected rows
FIRST(column)	Returns the value of the first record in a specified field
LAST(column)	Returns the value of the last record in a specified field
MAX(column)	Returns the highest value of a column
MIN(column)	Returns the lowest value of a column
STDEV(column)	
STDEVP(column)	
SUM(column)	Returns the total sum of a column
VAR(column)	

VARP(column)	
--------------	--

**Aggregate functions in SQL Server**

Function	Description
AVG(column)	Returns the average value of a column
BINARY_CHECKSUM	
CHECKSUM	
CHECKSUM_AGG	
COUNT(column)	Returns the number of rows (without a NULL value) of a column
COUNT(*)	Returns the number of selected rows
COUNT(DISTINCT column)	Returns the number of distinct results
FIRST(column)	Returns the value of the first record in a specified field (not supported in SQLServer2K)
LAST(column)	Returns the value of the last record in a specified field (not supported in SQLServer2K)
MAX(column)	Returns the highest value of a column
MIN(column)	Returns the lowest value of a column
STDEV(column)	
STDEVP(column)	
SUM(column)	Returns the total sum of a column
VAR(column)	
VARP(column)	

**Scalar functions**

Scalar functions operate against a single value, and return a single value based on the input value.

**Useful Scalar Functions in MS Access**

Function	Description
UCASE(c)	Converts a field to upper case
LCASE(c)	Converts a field to lower case
MID(c,start[,end])	Extract characters from a text field
LEN(c)	Returns the length of a text field
INSTR(c,char)	Returns the numeric position of a named character within a text field
LEFT(c,number_of_char)	Return the left part of a text field requested
RIGHT(c,number_of_char)	Return the right part of a text field requested
ROUND(c,decimals)	Rounds a numeric field to the number of decimals specified
MOD(x,y)	Returns the remainder of a division operation

Aggregate functions (like SUM) often need an added GROUP BY functionality.

**GROUP BY**

GROUP BY... was added to SQL because aggregate functions (like SUM) return the aggregate of all column values every time they are called, and without the GROUP BY function it was impossible to find the sum for each individual group of column values.

The syntax for the GROUP BY function is:

```
SELECT column,SUM(column) FROM table GROUP BY column
```

**GROUP BY Example**

This "Sales" Table:

Company	Amount
W3Schools	5500
IBM	4500
W3Schools	7100

And This SQL:

SELECT Company, SUM(Amount) FROM Sales
----------------------------------------

Returns this result:

Company	SUM(Amount)
W3Schools	17100
IBM	17100
W3Schools	17100

The above code is invalid because the column returned is not part of an aggregate. A GROUP BY clause will solve this problem:

SELECT Company, SUM(Amount) FROM Sales GROUP BY Company
------------------------------------------------------------

Returns this result:

Company	SUM(Amount)
W3Schools	12600
IBM	4500

## HAVING...

HAVING... was added to SQL because the WHERE keyword could not be used against aggregate functions (like SUM), and without HAVING... it would be impossible to test for result conditions.

The syntax for the HAVING function is:

SELECT column, SUM(column) FROM table GROUP BY column HAVING SUM(column) condition value
------------------------------------------------------------------------------------------------

This "Sales" Table:

Company	Amount
W3Schools	5500
IBM	4500
W3Schools	7100

This SQL:

SELECT Company, SUM(Amount) FROM Sales GROUP BY Company HAVING SUM(Amount)>10000
----------------------------------------------------------------------------------------

Returns this result

Company	SUM(Amount)
W3Schools	12600

## EMBEDDED SQL

Embedded SQL is a method of inserting inline SQL statements or queries into the code of a programming language, which is known as a host language. Because the host language cannot parse SQL, the inserted SQL is parsed by an embedded SQL preprocessor.

Embedded SQL is a robust and convenient method of combining the computing power of a programming language with SQL's specialized data management and manipulation capabilities.

### Structure of embedded SQL

Structure of embedded SQL defines step by step process of establishing a connection with DB and executing the code in the DB within the high level language.

#### Connection to DB

This is the first step while writing a query in high level languages. First connection to the DB that we are accessing needs to be established. This can be done using the keyword CONNECT. But it has to precede with 'EXEC SQL' to

indicate that it is a SQL statement.

```
EXEC SQL CONNECT db_name;
EXEC SQL CONNECT HR_USER; //connects to DB HR_USER
```

Once connection is established with DB, we can perform DB transactions. Since these DB transactions are dependent on the values and variables of the host language. Depending on their values, query will be written and executed. Similarly, results of DB query will be returned to the host language which will be captured by the variables of host language. Hence we need to declare the variables to pass the value to the query and get the values from query. There are two types of variables used in the host language.

- **Host variable :** These are the variables of host language used to pass the value to the query as well as to capture the values returned by the query. Since SQL is dependent on host language we have to use variables of host language and such variables are known as host variable. But these host variables should be declared within the SQL area or within SQL code. That means compiler should be able to differentiate it from normal C variables. Hence we have to declare host variables within BEGIN DECLARE and END DECLARE section. Again, these declare block should be enclosed within EXEC SQL and ‘;’.

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
int STD_ID;
char STD_NAME [15];
char ADDRESS[20];
```

```
EXEC SQL END DECLARE SECTION;
```

We can note here that variables are written inside begin and end block of the SQL, but they are declared using C code. It does not use SQL code to declare the variables. Why? This is because they are host variables – variables of C language. Hence we cannot use SQL syntax to declare them. Host language supports almost all the datatypes from int, char, long, float, double, pointer, array, string, structures etc.

When host variables are used in a SQL query, it should be preceded by colon – ‘:’ to indicate that it is a host variable. Hence when pre-compiler compiles SQL code, it substitutes the value of host variable and compiles.

```
EXEC SQL SELECT * FROM STUDENT WHERE STUDENT_ID =:STD_ID;
```

The following code is a simple embedded SQL program, written in C. The program illustrates many, but not all, of the embedded SQL techniques. The program prompts the user for an order number, retrieves the customer number, salesperson, and status of the order, and displays the retrieved information on the screen.

```
int main() {
 EXEC SQL INCLUDE SQLCA;
 EXEC SQL BEGIN DECLARE SECTION;
 int OrderID; /* Employee ID (from user) */
 int CustID; /* Retrieved customer ID */
 char SalesPerson[10] /* Retrieved salesperson name */
 char Status[6] /* Retrieved order status */
 EXEC SQL END DECLARE SECTION;

 /* Set up error processing */
 EXEC SQL WHENEVER SQLERROR GOTO query_error;
 EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

 /* Prompt the user for order number */
 printf("Enter order number: ");
 scanf_s("%d", &OrderID);

 /* Execute the SQL query */
 EXEC SQL SELECT CustID, SalesPerson, Status
 FROM Orders
 WHERE OrderID = :OrderID
 INTO :CustID, :SalesPerson, :Status;
```

```

/* Display the results */
printf ("Customer number: %d\n", CustID);
printf ("Salesperson: %s\n", SalesPerson);
printf ("Status: %s\n", Status);
exit();

query_error:
printf ("SQL error: %ld\n", sqlca->sqlcode);
exit();

bad_number:
printf ("Invalid order number.\n");
exit();
}

```

## DYNAMIC SQL

The main disadvantage of embedded SQL is that it supports only static SQLs. If we need to build up queries at run time, then we can use dynamic sql. That means if query changes according to user input, then it always better to use dynamic SQL. Like we said above, the query when user enters student name alone and when user enters both student name and address, is different. If we use embedded SQL, one cannot implement this requirement in the code. In such case dynamic SQL helps the user to develop query depending on the values entered by him, without making him know which query is being executed. It can also be used when we do not know which SQL statements like Insert, Delete update or select needs to be used, when number of host variables is unknown, or when datatypes of host variables are unknown or when there is direct reference to DB objects like tables, views, indexes are required. However this will make user requirement simple and easy but it may make query lengthier and complex. That means depending upon user inputs, the query may grow or shrink making the code flexible enough to handle all the possibilities. In embedded SQL, compiler knows the query in advance and pre-compiler compiles the SQL code much before C compiles the code for execution. Hence embedded SQLs will be faster in execution. But in the case of dynamic SQL, queries are created, compiled and executed only at the run time. This makes the dynamic SQL little complex, and time consuming.

Since query needs to be prepared at run time, in addition to the structures discussed in embedded SQL, we have three more clauses in dynamic SQL. These are mainly used to build the query and execute them at run time.

### PREPARE

Since dynamic SQL builds a query at run time, as a first step we need to capture all the inputs from the user. It will be stored in a string variable. Depending on the inputs received from the user, string variable is appended with inputs and SQL keywords. These SQL like string statements are then converted into SQL query. This is done by using PREPARE statement.

For example, below is the small snippet from dynamic SQL. Here sql\_stmt is a character variable, which holds inputs from the users along with SQL commands. But is cannot be considered as SQL query as it is still a sting value. It needs to be converted into a proper SQL query which is done at the last line using PREPARE statement. Here sql\_query is also a string variable, but it holds the string as a SQL query.

### EXECUTE

This statement is used to compile and execute the SQL statements prepared in DB.

EXEC SQL EXECUTE sql\_query;

### EXECUTE IMMEDIATE

This statement is used to prepare SQL statement as well as execute the SQL statements in DB. It performs the task of PREPARE and EXECUTE in a single line.

EXEC SQL EXECUTE IMMEDIATE :sql\_stmt;

Dynamic SQL will not have any SELECT queries and host variables. But it can be any other SQL statements like insert, delete, update, grant etc. But when we use insert/ delete/ updates in this type, we cannot use host variables. All the input values will be hardcoded. Hence the SQL statements can be directly executed using EXECUTE IMMEDIATE rather than using PREPARE and then EXECUTE.

```
EXEC SQL EXECUTE IMMEDIATE 'GRANT SELECT ON STUDENT TO Faculty';
```

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM STUDENT WHERE STD_ID = 100';
```

```
EXEC SQL EXECUTE IMMEDIATE 'UPDATE STUDENT SET ADDRESS = 'Troy' WHERE STD_ID =100';
```

STUCOR APP

**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**

**DATABASE DESIGN**

A well-designed database shall:

- Eliminate Data Redundancy: the same piece of data shall not be stored in more than one place. This is because duplicate data not only waste storage spaces but also easily lead to inconsistencies.
- Ensure Data Integrity and Accuracy

**Entity-Relationship Data Model**

- Classical, popular conceptual data model
- First introduced (mid 70's) as a (relatively minor) improvement to the relational model: pictorial diagrams are easier to read than relational database schemas
- Then evolved as a popular model for the first conceptual representation of data structures in the process of database design

**ER Model: Entity and Entity Set**

Considering the above example, **Student** is an entity, **Teacher** is an entity, similarly, **Class**, **Subjectetc** are also entities.

An Entity is generally a real-world object which has characteristics and holds relationships in a DBMS.

If a Student is an Entity, then the complete dataset of all the students will be the **Entity Set**

**ER Model: Attributes**

If a Student is an Entity, then student's roll no., student's name, student's age, student's gender etc will be its attributes.

An attribute can be of many types, here are different types of attributes defined in ER database model:

1. **Simple attribute:** The attributes with values that are atomic and cannot be broken down further are simple attributes. For example, student's age.
2. **Composite attribute:** A composite attribute is made up of more than one simple attribute. For example, student's address will contain, house no., street name, pincode etc.
3. **Derived attribute:** These are the attributes which are not present in the whole database management system, but are derived using other attributes. For example, average age of students in a class.
4. **Single-valued attribute:** As the name suggests, they have a single value.
5. **Multi-valued attribute:** And, they can have multiple values.

**ER Model: Relationships**

When an Entity is related to another Entity, they are said to have a relationship. For example, A ClassEntity is related to Student entity, because students study in classes, hence this is a relationship.

Depending upon the number of entities involved, a degree is assigned to relationships.

For example, if 2 entities are involved, it is said to be Binary relationship, if 3 entities are involved, it is said to be Ternary relationship, and so on.

### Working with ER Diagrams

ER Diagram is a visual representation of data that describes how data is related to each other. In ER Model, we disintegrate data into entities, attributes and setup relationships between entities, all this can be represented visually using the ER diagram.

### Components of ER Diagram

Entity, Attributes, Relationships etc form the components of ER Diagram and there are defined symbols and shapes to represent each one of them.

Let's see how we can represent these in our ER Diagram.

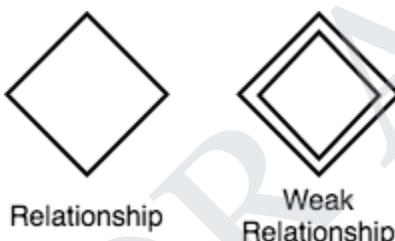
#### Entity

Simple rectangular box represents an Entity.



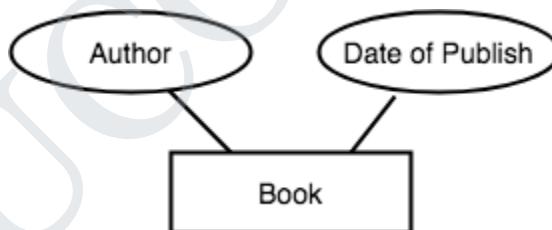
#### Relationships between Entities - Weak and Strong

Rhombus is used to setup relationships between two or more entities.



#### Attributes for any Entity

Ellipse is used to represent attributes of any entity. It is connected to the entity.



#### Weak Entity

A weak Entity is represented using double rectangular boxes. It is generally connected to another entity.



#### Key Attribute for any Entity

To represent a Key attribute, the attribute name inside the Ellipse is underlined.



**Derived Attribute for any Entity**

Derived attributes are those which are derived based on other attributes, for example, age can be derived from date of birth.

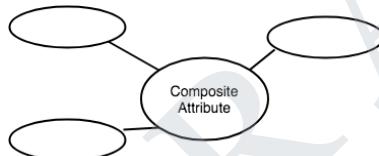
To represent a derived attribute, another dotted ellipse is created inside the main ellipse.

**Multivalued Attribute for any Entity**

Double Ellipse, one inside another, represents the attribute which can have multiple values.

**Composite Attribute for any Entity**

A composite attribute is the attribute, which also has attributes.

**ER Diagram: Entity**

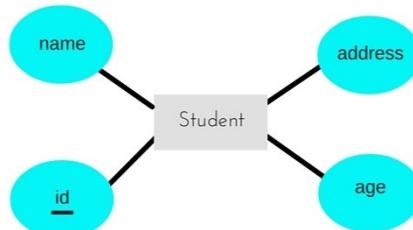
An Entity can be any object, place, person or class. In ER Diagram, an entity is represented using rectangles. Consider an example of an Organisation- Employee, Manager, Department, Product and many more can be taken as entities in an Organisation.



The yellow rhombus in between represents a relationship.

**ER Diagram: Key Attribute**

Key attribute represents the main characteristic of an Entity. It is used to represent a Primary key. Ellipse with the text underlined, represents Key Attribute.



**ER Diagram: Binary Relationship**

Binary Relationship means relation between two Entities. This is further divided into three types.

**One to One Relationship**

This type of relationship is rarely seen in real world.



The above example describes that one student can enroll only for one course and a course will also have only one Student. This is not what you will usually see in real-world relationships.

**One to Many Relationship**

The below example showcases this relationship, which means that 1 student can opt for many courses, but a course can only have 1 student. Sounds weird! This is how it is.

**Many to One Relationship**

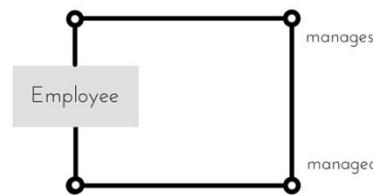
It reflects business rule that many entities can be associated with just one entity. For example, Student enrolls for only one Course but a Course can have many Students.

**Many to Many Relationship**

The above diagram represents that one student can enroll for more than one courses. And a course can have more than 1 student enrolled in it.

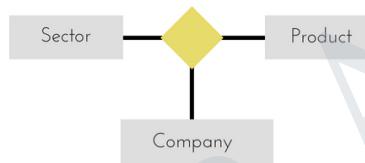
**ER Diagram: Recursive Relationship**

When an Entity is related with itself it is known as Recursive Relationship.

**ER Diagram: Ternary Relationship**

Relationship of degree three is called Ternary relationship.

A Ternary relationship involves three entities. In such relationships we always consider two entities together and then look upon the third.



- The above relationship involves 3 entities.
- Company operates in Sector, producing some Products.

For example, in the diagram above, we have three related entities, Company, Product and Sector. To understand the relationship better or to define rules around the model, we should relate two entities and then derive the third one. A Company produces many Products/ each product is produced by exactly one company.

A Company operates in only one Sector / each sector has many companies operating in it.

Considering the above two rules or relationships, we see that although the complete relationship involves three entities, but we are looking at two entities at a time.

**The Enhanced ER Model**

As the complexity of data increased in the late 1980s, it became more and more difficult to use the traditional ER Model for database modelling. Hence some improvements or enhancements were made to the existing ER Model to make it able to handle the complex applications better.

Hence, as part of the Enhanced ER Model, along with other improvements, three new concepts were added to the existing ER Model, they were:

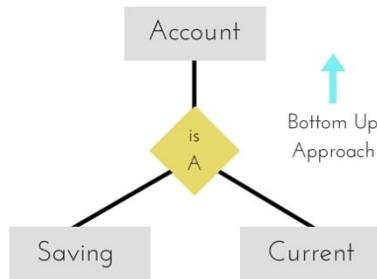
1. Generalization
2. Specialization
3. Aggregation

**Generalization**

Generalization is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entities to make further higher level entity.

It's more like Superclass and Subclass system, but the only difference is the approach, which is bottom-up. Hence, entities are combined to form a more generalised entity, in other words, sub-classes are combined to form a super-

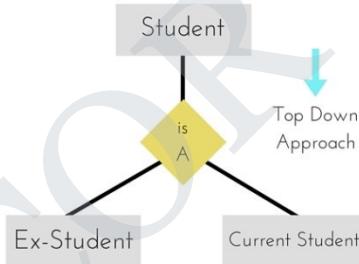
class.



For example, Saving and Current account types entities can be generalised and an entity with name Account can be created, which covers both.

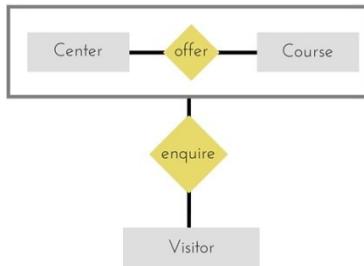
### Specialization

Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, a higher level entity may not have any lower-level entity sets, it's possible.



### Aggregation

Aggregation is a process when relation between two entities is treated as a single entity.



In the diagram above, the relationship between Center and Course together, is acting as an Entity, which is in relationship with another entity Visitor. Now in real world, if a Visitor or a Student visits a Coaching Center, he/she will never enquire about the center only or just about the course, rather he/she will ask enquire about both.

### ER Model to Relational Model

ER Model can be represented using ER Diagrams which is a great way of designing and representing the database design in more of a flow chart form.

It is very convenient to design the database using the ER Model by creating an ER diagram and later on converting it into relational model to design your tables.

Not all the ER Model constraints and components can be directly transformed into relational model, but an approximate schema can be derived.

Few examples of ER diagrams and convert it into relational model schema, hence creating tables in RDBMS.

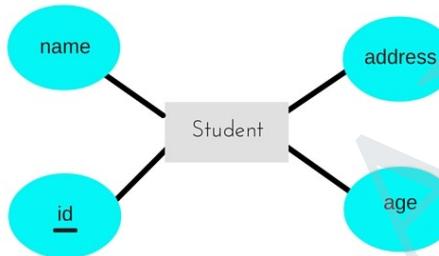
## Entity becomes Table

Entity in ER Model is changed into tables, or we can say for every Entity in ER model, a table is created in Relational Model.

And the attributes of the Entity gets converted to columns of the table.

And the primary key specified for the entity in the ER model, will become the primary key for the table in relational model.

For example, for the below ER Diagram in ER Model,



A table with name Student will be created in relational model, which will have 4 columns, id, name, age, address and id will be the primary key for this table.

### Table:Student

<u>id</u>	name	age	address
-----------	------	-----	---------

## Relationship becomes a Relationship Table

In ER diagram, we use diamond/rhombus to represent a relationship between two entities. In Relational model we create a relationship table for ER Model relationships too.

In the ER diagram below, we have two entities Teacher and Student with a relationship between them.



As discussed above, entity gets mapped to table, hence we will create table for Teacher and a table for Student with all the attributes converted into columns.

Now, an additional table will be created for the relationship, for example StudentTeacher or give it any name you like. This table will hold the primary key for both Student and Teacher, in a tuple to describe the relationship, which teacher teaches which student.

If there are additional attributes related to this relationship, then they become the columns for this table, like subject

name.

Also proper foreign key constraints must be set for all the tables.

## Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

$X \rightarrow Y$

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

### For example:

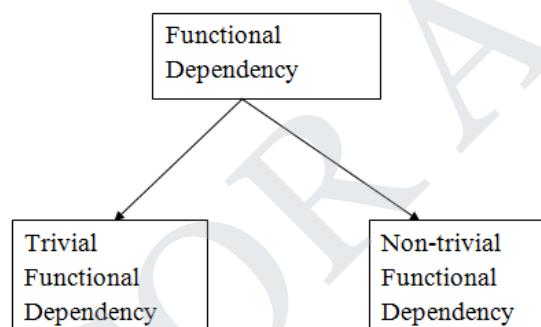
Assume we have an employee table with attributes: Emp\_Id, Emp\_Name, Emp\_Address.

Here Emp\_Id attribute can uniquely identify the Emp\_Name attribute of employee table because if we know the Emp\_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

$$\text{Emp_Id} \rightarrow \text{Emp_Name}$$

## Types of Functional dependency



### Trivial functional dependency

- $A \rightarrow B$  has trivial functional dependency if B is a subset of A.
- The following dependencies are also trivial like:  $A \rightarrow A$ ,  $B \rightarrow B$

#### Example:

Consider a table with two columns Employee\_Id and Employee\_Name.

$\{\text{Employee\_id}, \text{Employee\_Name}\} \rightarrow \text{Employee\_Id}$  is a trivial functional dependency as Employee\_Id is a subset of  $\{\text{Employee\_id}, \text{Employee\_Name}\}$ .

3. Also,  $\text{Employee\_Id} \rightarrow \text{Employee\_Id}$  and  $\text{Employee\_Name} \rightarrow \text{Employee\_Name}$  are trivial dependencies too.

### Non-trivial functional dependency

$A \rightarrow B$  has a non-trivial functional dependency if B is not a subset of A.

When A intersection B is NULL, then  $A \rightarrow B$  is called as complete non-trivial.

#### Example:

$$\begin{aligned} \text{ID} &\rightarrow \text{Name}, \\ \text{Name} &\rightarrow \text{DOB} \end{aligned}$$

## Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic

approach of decomposing tables to eliminate data redundancy(repetition) and undesirable characteristics like Insertion, Update and Deletion anomalies. It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.

Normalization is used for mainly two purposes,

- Eliminating redundant(useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

## Problems Without Normalization

If a table is not properly normalized and have data redundancy then it will not only eat up extra memory space but will also make it difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized. To understand these anomalies let us take an example of a Student table.

rollno	name	branch	hod	office_tel
401	Akon	CSE	Mr. X	53337
402	Bkon	CSE	Mr. X	53337
403	Ckon	CSE	Mr. X	53337
404	Dkon	CSE	Mr. X	53337

In the table above, we have data of 4 Computer Sci. students. As we can see, data for the fields branch, hod(Head of Department) and office\_tel is repeated for the students who are in the same branch in the college, this is Data Redundancy.

### Insertion Anomaly

Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as NULL.

Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students.

These scenarios are nothing but Insertion anomalies.

### Updation Anomaly

What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

### Deletion Anomaly

In our Student table, two different informations are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

## Normalization Rule

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

## First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single(atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

## Rules for First Normal Form

The first normal form expects you to follow a few simple rules while designing your database, and they are:

### Rule 1: Single Valued Attributes

Each column of your table should be single valued which means they should not contain multiple values. We will explain this with help of an example later, let's see the other rules for now.

### Rule 2: Attribute Domain should not change

This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.

**For example:** If you have a column dob to save date of births of a set of people, then you cannot or you must not save 'names' of some of them in that column along with 'date of birth' of others in that column. It should hold only 'date of birth' for all the records/rows.

### Rule 3: Unique name for Attributes/Columns

This rule expects that each column in a table should have a unique name. This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data.

If one or more columns have same name, then the DBMS system will be left confused.

### Rule 4: Order doesn't matter

This rule says that the order in which you store the data in your table doesn't matter.

## EXAMPLE

Create a table to store student data which will have student's roll no., their name and the name of subjects they have opted for.

Here is the table, with some sample data added to it.

roll_no	name	subject
101	Akon	OS, CN
103	Ckon	Java
102	Bkon	C, C++

The table already satisfies 3 rules out of the 4 rules, as all our column names are unique, we have stored data in the order we wanted to and we have not inter-mixed different type of data in columns.

But out of the 3 different students in our table, 2 have opted for more than 1 subject. And we have stored the subject names in a single column. But as per the 1st Normal form each column must contain atomic value. It's very simple, because all we have to do is break the values into atomic values.

Here is our updated table and it now satisfies the First Normal Form.

roll_no	name	subject
101	Akon	OS
101	Akon	CN
103	Ckon	Java
102	Bkon	C
102	Bkon	C++

By doing so, although a few values are getting repeated but values for the subject column are now atomic for each record/row. Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

### Second Normal Form (2NF)

For a table to be in the Second Normal Form,

1. It should be in the First Normal form.
2. And, it should not have Partial Dependency.

### Dependency

Let's take an example of a Student table with columns student\_id, name, reg\_no(registration number), branch and address(student's home address).

student_id	name	reg_no	branch	address

In this table, student\_id is the primary key and will be unique for every row, hence we can use student\_id to fetch any row of data from this table

Even for a case, where student names are same, if we know the student\_id we can easily fetch the correct record.

student_id	name	reg_no	branch	address
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat

Hence we can say a Primary Key for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table.

I can ask from branch name of student with student\_id 10, and I can get it. Similarly, if I ask for name of student with student\_id 10 or 11, I will get it. So all I need is student\_id and every other column depends on it, or can be fetched using it. This is Dependency and we also call it Functional Dependency.

### Partial Dependency

Now that we know what dependency is, we are in a better state to understand what partial dependency is. For a simple table like Student, a single column like student\_id can uniquely identify all the records in a table. But this is not true all the time. So now let's extend our example to see if more than 1 column together can act

as a primary key.

Let's create another table for Subject, which will have subject\_id and subject\_name fields and subject\_id will be the primary key.

subject_id	subject_name
1	Java
2	C++
3	Php

Now we have a Student table with student information and another table Subject for storing subject information.

Let's create another table Score, to store the marks obtained by students in the respective subjects. We will also be saving name of the teacher who teaches that subject along with marks.

score_id	student_id	subject_id	marks	teacher
1	10	1	70	Java Teacher
2	10	2	75	C++ Teacher
3	11	1	80	Java Teacher

In the score table we are saving the **student\_id** to know which student's marks are these and **subject\_id** to know for which subject the marks are for.

Together, student\_id + subject\_id forms a **Candidate Key** which can be the **Primary key**.

To get me marks of student with student\_id 10, can you get it from this table? No, because you don't know for which subject. And if I give you subject\_id, you would not know for which student. Hence we need student\_id + subject\_id to uniquely identify any row.

## But where is Partial Dependency?

Now if you look at the Score table, we have a column names teacher which is only dependent on the subject, for Java it's Java Teacher and for C++ it's C++ Teacher & so on.

Now as we just discussed that the primary key for this table is a composition of two columns which is student\_id & subject\_id but the teacher's name only depends on subject, hence the subject\_id, and has nothing to do with student\_id.

This is Partial Dependency, where an attribute in a table depends on only a part of the primary key and not on the whole key.

## How to remove Partial Dependency?

There can be many different solutions for this, but our objective is to remove teacher's name from Score table. The simplest solution is to remove columns teacher from Score table and add it to the Subject table. Hence, the Subject table will become:

subject_id	subject_name	teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

And our Score table is now in the second normal form, with no partial dependency.

score_id	student_id	subject_id	marks
1	10	1	70
2	10	2	75
3	11		

### Third Normal Form (3NF)

A table is said to be in the Third Normal Form when,

1. It is in the Second Normal form.
2. And, it doesn't have Transitive Dependency.

So let's use the same example, where we have 3 tables, **Student**, **Subject** and **Score**.

#### Student Table

student_id	name	reg_no	branch	address
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat
12	Bkon	09-WY	IT	Rajasthan

#### Subject Table

subject_id	subject_name	teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

#### Score Table

In the Score table, we need to store some more information, which is the exam name and total marks, so let's add 2 more columns to the Score table.

score_id	student_id	subject_id	marks
1	10	1	70
2	10	2	75
3	11	1	80

### Transitive Dependency

With exam\_name and total\_marks added to our Score table, it saves more data now. Primary key for the Score table is a composite key, which means it's made up of two attributes or columns → student\_id + subject\_id. The new column exam\_name depends on both student and subject. For example, a mechanical engineering student will have Workshop exam but a computer science student won't. And for some subjects you have Practical exams and for some you don't. So we can say that exam\_name is dependent on both student\_id and subject\_id.

And what about our second new column total\_marks? Does it depend on our Score table's primary key?

Well, the column total\_marks depends on exam\_name as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.

But, exam\_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total\_marks depends on it.

This is Transitive Dependency. When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

### How to remove Transitive Dependency

Again the solution is very simple. Take out the columns exam\_name and total\_marks from Score table and put them in an Exam table and use the exam\_id wherever required.

### Score Table: In 3rd Normal Form

score_id	student_id	subject_id	marks	exam_id

### The new Exam table

exam_id	exam_name	total_marks
1	Workshop	200
2	Mains	70
3	Practicals	30

### Advantage of removing Transitive Dependency

The advantage of removing transitive dependency is,

- Amount of data duplication is reduced.
- Data integrity achieved.

### Boyce and Codd Normal Form (BCNF)

Boyce and Codd Normal Form is a higher version of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

- R must be in 3rd Normal Form
- and, for each functional dependency ( $X \rightarrow Y$ ), X should be a super Key. In simple words, it means, that for a dependency  $A \rightarrow B$ , A cannot be a non-prime attribute, if B is a prime attribute.

### Example

College enrolment table with columns student\_id, subject and professor.

student_id	subject	professor
101	Java	P.Java
101	C++	P.Cpp
102	Java	P.Java2
103	C#	P.Chash
104	Java	P.Java

In the table above:

One student can enroll for multiple subjects. For example, student with student\_id 101, has opted for subjects - Java & C++

- For each subject, a professor is assigned to the student.
- And, there can be multiple professors teaching one subject like Java.

What do you think should be the Primary Key?

Well, in the table above student\_id, subject together form the primary key, because using student\_id and subject, we can find all the columns of the table.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between subject and professor here, where subject depends on the professor name.

This table satisfies the 1st Normal form because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the 2nd Normal Form as there is no Partial Dependency.

And, there is no Transitive Dependency, hence the table also satisfies the 3rd Normal Form.

But this table is not in Boyce-Codd Normal Form.

### **Why this table is not in BCNF?**

In the table above, student\_id, subject form primary key, which means subject column is a prime attribute. But, there is one more dependency, professor → subject.

And while subject is a prime attribute, professor is a non-prime attribute, which is not allowed by BCNF.

### **How to satisfy BCNF?**

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, student table and professor table.

Below we have the structure for both the tables.

#### **Student Table**

student_id	p_id
101	1
101	2

#### **Professor Table**

p_id	professor	subject
1	P.Java	Java
2	P.Cpp	C++

And now, this relation satisfy Boyce-Codd Normal Form.

### **Fourth Normal Form (4NF)**

A table is said to be in the Fourth Normal Form when,

1. It is in the Boyce-Codd Normal Form.
2. And, it doesn't have Multi-Valued Dependency.

### **Multi-valued Dependency**

A table is said to have multi-valued dependency, if the following conditions are true,

1. For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
2. Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
3. And, for a relation R(A,B,C), if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.

If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

**Example**

Below we have a college enrolment table with columns s\_id, course and hobby.

s_id	course	hobby
1	Science	Cricket
1	Maths	Hockey
2	C#	Cricket
2	PHP	Hockey

From the table above, student with s\_id 1 has opted for two courses, Science and Maths, and has two hobbies, Cricket and Hockey.

You must be thinking what problem this can lead to, right?

Well the two records for student with s\_id 1, will give rise to two more records, as shown below, because for one student, two hobbies exists, hence along with both the courses, these hobbies should be specified.

s_id	course	hobby
1	Science	Cricket
1	Maths	Hockey
1	Science	Hockey
1	Maths	Cricket

And, in the table above, there is no relationship between the columns course and hobby. They are independent of each other.

So there is multi-value dependency, which leads to un-necessary repetition of data and other anomalies as well.

**How to satisfy 4th Normal Form?**

To make the above relation satisfy the 4th normal form, we can decompose the table into 2 tables.

**CourseOpted Table**

s_id	course
1	Science
1	Maths
2	C#
2	PHP

**Hobbies Table,**

s_id	hobby
1	Cricket
1	Hockey
2	Cricket
2	Hockey

Now this relation satisfies the fourth normal form.

A table can also have functional dependency along with multi-valued dependency. In that case, the functionally dependent columns are moved in a separate table and the multi-valued dependent columns are moved to separate tables.

**Fifth Normal Form (5NF)**

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

1. It's in 4NF
2. 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.

**What is Join Dependency**

If a table can be recreated by joining multiple tables and each of this table have a subset of the attributes of the table, then the table is in Join Dependency. It is a generalization of Multivalued DependencyJoin Dependency can be related to 5NF, wherein a relation is in 5NF, only if it is already in 4NF and it cannot be decomposed further.

Example

<Employee>

EmpName	EmpSkills	EmpJob (Assigned Work)
Tom	Networking	EJ001
Harry	Web Development	EJ002
Katie	Programming	EJ002

The above table can be decomposed into the following three tables; therefore it is not in 5NF:

<EmployeeSkills>

EmpName	EmpSkills
Tom	Networking
Harry	Web Development
Katie	Programming

<EmployeeJob>

EmpName	EmpJob
Tom	EJ001
Harry	EJ002
Katie	EJ002

<JobSkills>

EmpSkills	EmpJob
Networking	EJ001
Web Development	EJ002
Programming	EJ002

Our Join Dependency:

{(EmpName, EmpSkills), (EmpName, EmpJob), (EmpSkills, EmpJob)}

The above relations have join dependency, so they are not in 5NF. That would mean that a join relation of the above three relations is equal to our original relation <Employee>.

### FIFTH NORMAL FORM EXAMPLE

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	Mathematics	Alex	SEMESTER 1
LECTURER	Mathematics	Rose	SEMESTER 1
CLASS	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.

### UNIT III - TRANSACTIONS

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

#### 1. TRANSACTION CONCEPTS

##### **What is Transaction?**

A set of logically related operations is known as transaction. The main operations of a transaction are:

**Read(A):** Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in main memory.

**Write (A):** Write operation Write(A) or W(A) writes the value back to the database from buffer.

Let us take a debit transaction from an account which consists of following operations:

- 1.R(A);
- 2.A=A-1000;
- 3.W(A);

Assume A's value before starting of transaction is 5000.

- The first operation reads the value of A from database and stores it in a buffer.
- Second operation will decrease its value by 1000. So buffer will contain 4000.
- Third operation will write the value from buffer to database. So A's final value will be 4000.

But it may also be possible that transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power** etc. For example, if debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank.

To avoid this, Database has two important operations:

**Commit:** After all instructions of a transaction are successfully executed, the changes made by transaction are made permanent in the database.

**Rollback:** If a transaction is not able to execute all operations successfully, all the changes made by transaction are undone.

##### **Why do you need concurrency in Transactions?**

A database is a shared resource accessed. It is used by many users and processes concurrently. For example, the banking system, railway, and air reservations systems, stock market monitoring, supermarket inventory, and checkouts, etc.

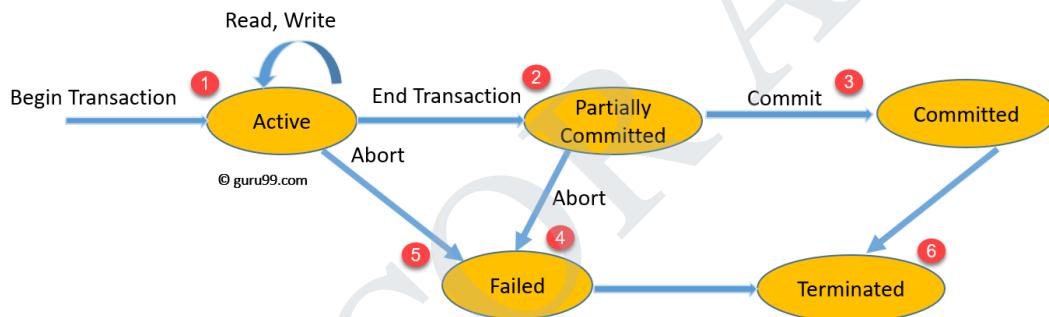
Not managing concurrent access may create issues like:

- Hardware failure and system crashes
- Concurrent execution of the same transaction, deadlock, or slow performance

#### **States of Transactions**

The various states of a Database Transaction are listed below

State	Transaction types
<b>Active State</b>	A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.
<b>Partially Committed</b>	A transaction goes into the partially committed state after the end of a transaction.
<b>Committed State</b>	When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently.
<b>Failed State</b>	A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state.
<b>Terminated State</b>	State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted.



### Transition Diagram for a Database Transaction

Let's study a state transition diagram that highlights how a transaction moves between these various states.

1. Once a transaction starts execution, it becomes active. It can issue READ or WRITE operation.
2. Once the READ and WRITE operations complete, the transaction becomes partially committed state.
3. Next, some recovery protocols need to ensure that a system failure will not result in an inability to record changes in the transaction permanently. If this check is a success, the transaction commits and enters into the committed state.
4. If the check is a fail, the transaction goes to the Failed state.
5. If the transaction is aborted while it's in the active state, it goes to the failed state. The transaction should be rolled back to undo the effect of its write operations on the database.
6. The terminated state refers to the transaction leaving the system.

## 2. ACID PROPERTIES

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after transaction, certain properties are followed. These are called **ACID properties**.

### Atomicity

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves following two operations.

—Abort: If a transaction aborts, changes made to database are not visible.

—Commit: If a transaction commits, changes made are visible.

Atomicity is also known as the ‘All or nothing rule’.

Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.

<b>Before:</b> X : 500	Y: 200
Transaction T	
<b>T1</b>	<b>T2</b>
Read (X) X: = X - 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
<b>After:</b> X : 400	Y : 300

If the transaction fails after completion of T1 but before completion of T2. ( say, after write(X) but before write(Y)), then amount has been deducted from X but not added to Y. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

### Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to **correctness of a database**.

Referring to the example above,

The total amount before and after the transaction must be maintained.

Total before T occurs =  $500 + 200 = 700$ .

Total after T occurs =  $400 + 300 = 700$ .

Therefore, database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete.

### Isolation

This property ensures that multiple transactions can occur concurrently without leading to inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved if these were executed serially in some order.

Let X= 500, Y = 500.

Consider two transactions T and T”.

T	T''
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y - 50	
Write	

Suppose T has been executed till Read (Y) and then T'' starts. As a result , interleaving of operations takes place due to which T'' reads correct value of X but incorrect value of Y and sum computed by T'':  $(X+Y = 50,000+500=50,500)$

is thus not consistent with the sum at end of transaction:

T:  $(X+Y = 50,000 + 450 = 50,450)$ .

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

### Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost.

## 3. SCHEDULES

### 1. Serial Schedules

Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

**Example:** Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

where R(A) denotes that a read operation is performed on some data item 'A'  
This is a serial schedule since the transactions perform serially in the order T1 → T2

### 2. Complete Schedules

Schedules in which the last operation of each transaction is either abort (or) commit are called complete schedules.

**Example:** Consider the following schedule involving three transactions T1, T2 and T3.

T1	T2	T3
R(A)		
	W(A)	
R(B)		
		W(B)
commit		
	commit	
		abort

This is a complete schedule since the last operation performed under every transaction is either “commit” or “abort”.

### 3. Recoverable Schedules

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

**Example –** Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

### 4. Cascadeless Schedules –

Also called **Avoids cascading aborts/rollbacks (ACA)**. Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called **cascadeless schedules**. Avoids that a single transaction abort leads to a series of **transaction rollbacks**. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.

**Example:** Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of A is read by T2 only after the updating transaction i.e. T1 commits.

### 5. Strict Schedules

A schedule is strict if for any two transactions  $T_i, T_j$ , if a write operation of  $T_i$  precedes a conflicting operation of  $T_j$  (either read or write), then the commit or abort event of  $T_i$  also precedes that conflicting operation of  $T_j$ .

In other words,  $T_j$  can read or write updated or written value of  $T_i$  only after  $T_i$  commits/aborts.

**Example:** Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T2 reads and writes A which is written by T1 only after the commit of T1.

**Note –** It can be seen that:

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

## 4. SERIALIZABILITY

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. **Serializability** is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

### What is a serializable schedule?

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.

#### Types of Serializability

There are two types of Serializability.

- I. Conflict Serializability
- II. View Serializability

#### 1. Conflict Serializability

**Conflict Serializability** is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.

#### What is Conflict Serializability?

A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

#### Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.
2. Both the operations are working on same data item.
3. At least one of the operations is a write operation.

Let's see some examples to understand this:

**Example 1:** Operation W(X) of transaction T1 and operation R(X) of transaction T2 are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operations is write operation.

**Example 2:** Similarly, Operations W(X) of T1 and W(X) of T2 are conflicting operations.

**Example 3:** Operations W(X) of T1 and W(Y) of T2 are non-conflicting operations because both the write operations are not working on same data item so these operations don't satisfy the second condition.

**Example 4:** Similarly, R(X) of T1 and R(X) of T2 are non-conflicting operations because none of them is write operation.

**Example 5:** Similarly, W(X) of T1 and R(X) of T1 are non-conflicting operations because both the operations belong to same transaction T1.

#### Conflict Equivalent Schedules

Two schedules are said to be conflict Equivalent if one schedule can be converted into other schedule after swapping non-conflicting operations.

#### Conflict Serializable check

Let's check whether a schedule is conflict serializable or not. If a schedule is conflict Equivalent to its serial schedule then it is called **Conflict Serializable schedule**. Let's take few examples of schedules.

#### Example of Conflict Serializability

Let's consider this schedule:

T1	T2
-----	-----
R(A)	
R(B)	

R(A)  
R(B)  
W(B)

W(A)

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However, we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable**.

**Let's take another example:**

**T1      T2**

-----  
R(A)  
R(A)  
R(B)  
W(B)

R(B)

W(A)

**Let's swap non-conflicting operations:**

After swapping R(A) of T1 and R(A) of T2 we get:

**T1      T2**

-----  
R(A)  
R(A)  
R(B)  
W(B)

R(B)

W(A)

After swapping R(A) of T1 and R(B) of T2 we get:

**T1      T2**

-----  
R(A)  
R(B)  
R(A)  
W(B)

R(B)

W(A)

After swapping R(A) of T1 and W(B) of T2 we get:

**T1      T2**

-----  
R(A)  
R(B)  
W(B)

R(A)

R(B)

W(A)

We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is **Conflict Serializable**.

## 2. View Serializability

View Serializability is a process to find out that a given schedule is view serializable or not.

To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule. Lets take an example to understand what I mean by that.

Given Schedule:

T1      T2

----- -----

R(X)

W(X)

R(X)

W(X)

R(Y)

W(Y)

R(Y)

W(Y)

### Serial Schedule of the above given schedule:

As we know that in Serial schedule a transaction only starts when the current running transaction is finished. So the serial schedule of the above given schedule would look like this:

T1      T2

----- -----

R(X)

W(X)

R(Y)

W(Y)

R(X)

W(X)

R(Y)

W(Y)

If we can prove that the given schedule is **View Equivalent** to its serial schedule then the given schedule is called **view Serializable**.

### Testing for Serializability

**Example 3.5.6** Consider the following schedules. The actions are listed in the order they are scheduled, and prefixed with the transaction name.

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(Z), 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 schedules ?

iii) Is the schedule view-serializable ? If so, what are all the view equivalent serial schedules ?

AU : May-15, Marks 2 + 7 + 7

**Solution:** i) We will find **conflicting operations**. Two operations are called as conflicting operations if all the following conditions hold true for them-

Both the operations belong to different transactions.

Both the operations are on same data item.

At least one of the two operations is a write operation

For S1: From above given example in the top to bottom scanning we find the conflict as

$T_1 : W(Y), T_2 : W(Y)$  and

$T_2 : W(Y), T_1 : R(Y)$

Hence we will build the precedence graph. Draw the edge between conflicting transactions. For example in above given scenario, the conflict occurs while moving from  $T_1 : W(Y)$  to  $T_2 : W(Y)$ . Hence edge must be from  $T_1$  to  $T_2$ . Similarly for second conflict, there will be the edge from  $T_2$  to  $T_1$



Fig. 3.5.6 Precedence graph for S1

For S2: The conflicts are

- o  $T_3 : W(X), T_1 : R(X)$
- o  $T_2 : W(Z), T_3 : R(Z)$

Hence the precedence graph is as follows -



Fig. 3.5.7 Precedence graph for S2

(ii)

- o S1 is **not conflict-serializable** since the dependency graph has a cycle.
- o S2 is conflict-serializable as the dependency graph is acyclic. The order  $T_2-T_3-T_1$  is the only equivalent serial order.

(iii)

- o S1 is **not view serializable**.
- o S2 is trivially view-serializable as it is conflict serializable. The only serial order allowed is

**T2-T3-T1.**

## 5. CONCURRENCY CONTROL

In the concurrency control, the multiple transactions can be executed simultaneously.

It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

## 6. NEED FOR CONCURRENCY

### Problems of concurrency control

Several problems can occur when concurrent transactions are executed in an uncontrolled manner.

Following are the three problems in concurrency control.

- Lost updates
- Dirty read
- Unrepeatable read

### 1. Lost update problem

When two transactions that access the same database items contain their operations in a way that makes the value of some database item incorrect, then the lost update problem occurs.

If two transactions T1 and T2 read a record and then update it, then the effect of updating of the first record will be overwritten by the second update.

**Example:**

Transaction-X	Time	Transaction-Y
—	t1	—
Read A	t2	—
—	t3	Read A
Update A	t4	—
—	t5	Update A
—	t6	—

Here,

- At time t2, transaction-X reads A's value.
- At time t3, Transaction-Y reads A's value.
- At time t4, Transactions-X writes A's value on the basis of the value seen at time t2.
- At time t5, Transactions-Y writes A's value on the basis of the value seen at time t3.
- So at time T5, the update of Transaction-X is lost because Transaction y overwrites it without looking at its current value.
- Such type of problem is known as Lost Update Problem as update made by one transaction is lost here.

### 2. Dirty Read

- The dirty read occurs in the case when one transaction updates an item of the database, and then the transaction fails for some reason. The updated database item is accessed by another transaction before it is changed back to the original value.
- A transaction T1 updates a record which is read by T2. If T1 aborts then T2 now has values which have never formed part of the stable database.

**Example:**

Transaction-X	Time	Transaction-Y
—	t1	—
—	t2	Update A
Read A	t3	—
—	t4	Rollback
—	t5	—

- At time t2, transaction-Y writes A's value.
- At time t3, Transaction-X reads A's value.

- At time t4, Transaction-Y rollbacks. So, it changes A's value back to that of prior to t1.
- So, Transaction-X now contains a value which has never become part of the stable database.
- Such type of problem is known as Dirty Read Problem, as one transaction reads a dirty value which has not been committed.

### 3. Inconsistent Retrievals Problem

- Inconsistent Retrievals Problem is also known as unrepeatable read. When a transaction calculates some summary function over a set of data while the other transactions are updating the data, then the Inconsistent Retrievals Problem occurs.
- A transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now when the transaction T1 reads the record, then the new value will be inconsistent with the previous value.

#### Example:

Suppose two transactions operate on three accounts.

Account-1	Account-2	Account-3
Balance = 200	Balance = 250	Balance = 150

Transaction-X	Time	Transaction-Y
—	t1	—
Read Balance of Acc-1 sum <-- 200	t2	—
Read Balance of Acc-2		—
Sum <-- Sum + 250 = 450	t3	—
—	t4	Read Balance of Acc-3
—	t5	Update Balance of Acc-3 150 --> 150 - 50 --> 100
—	t6	Read Balance of Acc-1
—	t7	Update Balance of Acc-1 200 --> 200 + 50 --> 250
Read Balance of Acc-3	t8	COMMIT
Sum <-- Sum + 250 = 550	t9	—

- Transaction-X is doing the sum of all balance while transaction-Y is transferring an amount 50 from Account-1 to Account-3.
- Here, transaction-X produces the result of 550 which is incorrect. If we write this produced result in the database, the database will become an inconsistent state because the actual sum is 600.
- Here, transaction-X has seen an inconsistent state of the database.

## Concurrency Control Protocol

Concurrency control protocols ensure atomicity, isolation, and serializability of concurrent transactions. The concurrency control protocol can be divided into three categories:

1. Lock based protocol
2. Time-stamp protocol

## 7. LOCKING PROTOCOLS

### Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

#### 1. Shared lock:

- o It is also known as a Read-only lock. In a shared lock, the data item can only be read by the transaction.
- o It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

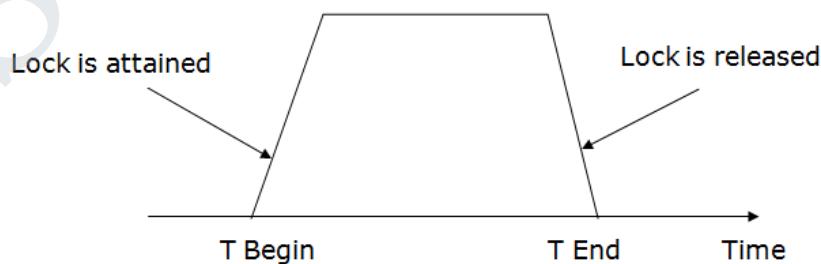
#### 2. Exclusive lock:

- o In the exclusive lock, the data item can be both reads as well as written by the transaction.
- o This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

## 8. TWO PHASE LOCKING (2PL)

The two-phase locking protocol divides the execution phase of the transaction into three parts.

- o In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- o In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- o In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



**There are two phases of 2PL:**

- **Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.
- **Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

**In the below example, if lock conversion is allowed then the following phase can happen:**

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

**Example:**

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	—	—
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	—	—

The following way shows how unlocking and locking work with 2-PL.

**Transaction T1:**

- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3

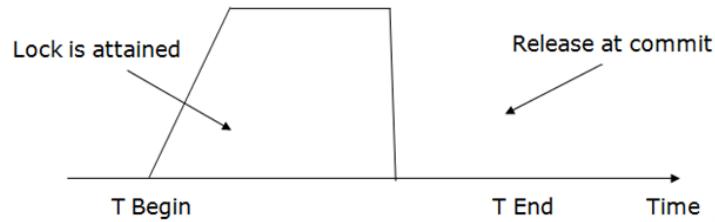
**Transaction T2:**

- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

**Types of Two Phase Locking (2PL)****1. Strict Two-phase locking (Strict-2PL)**

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.

- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



It does not have cascading abort as 2PL does.

## 2. Rigorous Two-Phase Locking

- Rigorous Two – Phase Locking Protocol avoids cascading rollbacks.
- This protocol requires that all the share and exclusive locks to be held until the transaction commits.

### Timestamp Ordering Protocol

- 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.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

#### Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction  $T_i$  issues a **Read (X)** operation:
  - If  $W\_TS(X) > TS(T_i)$  then the operation is rejected.
  - If  $W\_TS(X) \leq TS(T_i)$  then the operation is executed.
  - Timestamps of all the data items are updated.
2. Check the following condition whenever a transaction  $T_i$  issues a **Write(X)** operation:
  - If  $TS(T_i) < R\_TS(X)$  then the operation is rejected.
  - If  $TS(T_i) < W\_TS(X)$  then the operation is rejected and  $T_i$  is rolled back otherwise the operation is executed.

**Where,**

$TS(T_i)$  denotes the timestamp of the transaction  $T_i$ .

$R_TS(X)$  denotes the Read time-stamp of data-item X.

$W_TS(X)$  denotes the Write time-stamp of data-item X.

**Thomas write Rule**

Thomas Write Rule provides the guarantee of serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.

**The basic Thomas write rules are as follows:**

- o If  $TS(T) < R_TS(X)$  then transaction T is aborted and rolled back, and operation is rejected.
- o If  $TS(T) < W_TS(X)$  then don't execute the  $W_{-item}(X)$  operation of the transaction and continue processing.
- o If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction  $T_i$  and set  $W_TS(X)$  to  $TS(T)$ .

**Problems**

Scanned with  
CamScanner

**Example 3.10.2** 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 two phase locking protocol. Can the execution of these transactions result in deadlock ?

AU : Dec.-16, Marks 6

**Solution :**

T1	T2
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Read(B)	Read(A)
if A=0 then B=B+1	if B=0 then A=A+1
Write(B)	Write(A)
Unlock(A)	Unlock(B)
Commit	Commit
Unlock(B)	Unlock(A)

This is lock-unlock instruction sequence help to satisfy the requirements for strict two phase locking for the given transactions.

The execution of these transactions result in deadlock. Consider following partial execution scenario which leads to deadlock.

T1	T2
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Now it will wait for T2 to release exclusive lock on A	Now it will wait for T1 to release exclusive lock on B

## 9. DEADLOCK

A **deadlock** is a condition wherein two or more tasks are waiting for each other in order to be finished but none of the task is willing to give up the resources that other task needs. In this situation no task ever gets finished and is in waiting state forever.

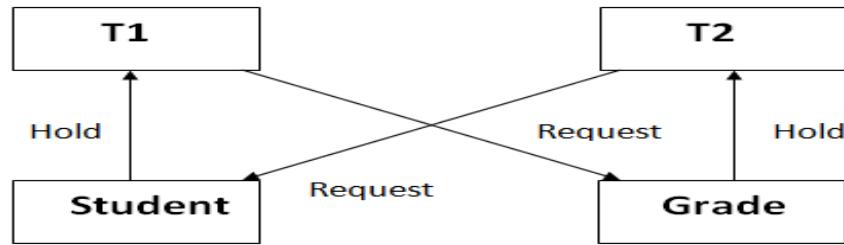
### Coffman conditions

Coffman stated four conditions for a deadlock occurrence. A deadlock may occur if all the following conditions holds true.

- **Mutual exclusion condition:** There must be at least one resource that cannot be used by more than one process at a time.
- **Hold and wait condition:** A process that is holding a resource can request for additional resources that are being held by other processes in the system.
- **No preemption condition:** A resource cannot be forcibly taken from a process. Only the process can release a resource that is being held by it.
- **Circular wait condition:** A condition where one process is waiting for a resource that is being held by second process and second process is waiting for third process ....so on and the last process is waiting for the first process. Thus, making a circular chain of waiting.

**For example:** In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.

**Figure: Deadlock in DBMS**

### Deadlock Avoidance

- When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.
- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

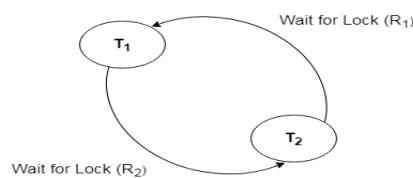
### Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

### Wait for Graph

- This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.
- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:



## Deadlock Prevention

- Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.
- The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

### I. Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions  $T_i$  and  $T_j$  and let  $TS(T)$  is a timestamp of any transaction  $T$ . If  $T_2$  holds a lock by some other transaction and  $T_1$  is requesting for resources held by  $T_2$  then the following actions are performed by DBMS:

1. **Check if  $TS(T_i) < TS(T_j)$**  - If  $T_i$  is the older transaction and  $T_j$  has held some resource, then  $T_i$  is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.
2. **Check if  $TS(T_i) < TS(T_j)$**  - If  $T_i$  is older transaction and has held some resource and if  $T_j$  is waiting for it, then  $T_j$  is killed and restarted later with the random delay but with the same timestamp.

### II. Wound wait scheme

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

Here is the table representation of resource allocation for each algorithm. Both of these algorithms take process age into consideration while determining the best possible way of resource allocation for deadlock avoidance. One of the famous deadlock avoidance algorithm is **Banker's algorithm**

## 10. TRANSACTION RECOVERY

### 3.13 Transaction Recovery

- An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure.
- The recovery scheme must also provide high availability; that means, it must minimize the time for which the database is not usable after a failure.

#### 3.13.1 Failure Classification

Various types of failures are -

- 1) **Transaction Failure** : Following are two types of errors due to which the transaction gets failed.
  - **Logical Error :**
    - i) This error is caused due to internal conditions such as bad input, data not found, overflow of resource limit and so on.
    - ii) Due to logical error the transaction can not be continued.
  - **System Error :**
    - i) When the system enters in an undesired state and then the transaction can not be continued then this type of error is called as system error.
- 2) **System Crash** : The situation in which there is a hardware malfunction, or a bug in the database software or the operating system, and because of which there is a loss of the content of volatile storage, and finally the transaction processing come to a halt is called system crash.
- 3) **Disk Failure** : A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. The backup of data is maintained on the secondary disks or DVD to recover from such failure.

#### 3.13.2 Storage

A DBMS stores the data on external storage because the amount of data is very huge and must persist across program executions.

The storage structure is a memory structure in the system. It has following categories -

##### 1) Volatile :

- Volatile memory is a primary memory in the system and is placed along with the CPU.
- These memories can store only small amount of data, but they are very fast. For example - main memory, cache memory.
- A volatile storage cannot survive system crashes.
- That means data in these memories will be lost on failure.

**2) Non Volatile :**

Non volatile memory is a secondary memory and is huge in size. For example : Hard disk, Flash memory, magnetic tapes.

These memories are designed to withstand system crashes.

**3) Stable :**

- Information residing in stable storage is never lost.
- To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes.

**Stable Storage Implementation**

- Stable storage is a kind of storage on which the information residing on it is never lost.
- Although stable storage is theoretically impossible to obtain it can be approximately built by applying a technique in which **data loss is almost impossible**.
- That means the information is replicated in several nonvolatile storage media with **independent failure modes**.
- Updates must be done with care to ensure that a failure during an update to stable storage **does not cause a loss of information**.

**3.13.3 Recovery with Concurrent Transactions**

There are four ways for recovery with concurrent transactions :

- 1) Interaction with concurrency control :** In this scheme recovery depends upon the concurrency control scheme which is used for the transaction. If a transaction gets failed, then rollback and undo all the updates performed by transaction.
- 2) Transaction Rollback :** In this scheme, if the transaction gets failed, then the failed transaction can be rolled back with the help of **log**. The system scans the log backward, and with the help of log entries the system can restore the data items.
- 3) Checkpoints :** The checkpoints are used to reduce the number of log records.
- 4) Restart recovery :** When the system recovers from the crash it constructs two lists : Undo-list and Redo-list. The undo list consists of transactions to be undone. The redo list consists of transactions to be redone. These two lists are handled as follows

**Step 1 :** Initially both the lists are empty.

**Step 2 :** The system scans the log entries from backwards. It scans each entry until it finds **first checkpoint record**.

### 3.13.4 Shadow Copy Technique

- In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database.
- All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched.
- If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
- The current copy of the database is identified by a pointer, called **db-pointer**, which is stored on disk.
- If the transaction **partially commits**, it is committed as follows :
  - First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk.
  - After the operating system has written all the pages to disk, the database system updates the pointer **db-pointer** to point to the new copy of the database
  - The new copy then becomes the current copy of the database.
  - The old copy of the database is then deleted.
    - The transaction is said to have been **committed** at the point where the **updated db-pointer** is written to disk.
    - The disk system guarantees that it will update **db-pointer** atomically, as long as we make sure that **db-pointer** lies entirely in a single sector.

#### Where is shadow copy technique used ?

- Shadow copy schemes are commonly used by text editors.
- Shadow copying can be used for small databases.

### 3.13.5 Log Based Recovery Approach

- Log is the most commonly used structure for recording the modifications that are to be made in the actual database. Hence during the recovery procedure a **log file** is maintained.
- A log record maintains four types of operations. Depending upon the type of operations there are four types of log records-
  1. **<Start>** Log record: It is represented as  $\langle T_i, \text{Start} \rangle$
  2. **<Update>** Log record

- 3. **<Commit>** Log record: It is represented as  $\langle T_i, \text{Commit} \rangle$
- 4. **<Abort>** Log record: It is represented as  $\langle T_i, \text{Abort} \rangle$
- The log contains various fields as shown in following Fig. 3.13.1. This structure is for **<update>** operation

Transaction ID( $T_i$ )	Data Item Name	Old Value of Data Item	New Value of Data Item
-------------------------	----------------	------------------------	------------------------

- For example : The sample log file is

$\langle T_1, \text{Start} \rangle$   
 $\langle T_1, a, 10, 20 \rangle$   
 $\langle T_1, \text{Commit} \rangle$

Here 10 represents the old value before commit operation and 20 is the new value that needs to be updated in the database after commit operation

**Fig. 3.13.1 Sample Log File**

- The log must be maintained on the stable storage and the entries in the log file are maintained before actually updating the physical database.
- There are two approaches used for log based recovery technique - Deferred Database Modification and Immediate Database Modification.

#### **I. Deferred Database Modification :**

- In this technique, the database is not updated immediately.
- Only log file is updated on each transaction.
- When the transaction reaches to its commit point, then only the database is physically updated from the log file.
- In this technique, if a transaction fails before reaching to its commit point, it will not have changed database anyway. Hence there is no need for the UNDO operation. The REDO operation is required to record the operations from log file to physical database. Hence deferred database modification technique is also called as **NO UNDO/REDO algorithm**.
- For example :**

Consider two transactions  $T_1$  and  $T_2$  as follows :

$T_1$	$T_2$
Read (A, a)	Read (C, c)
$a = a - 10$	$c = c - 20$
Write (A, a)	Write (C, c)
Read (B, b)	
$b = b + 10$	
Write (B, b)	

If  $T_1$  and  $T_2$  are executed serially with initial values of  $A = 100$ ,  $B = 200$  and  $C = 300$ , then the state of log and database if crash occurs

- a) Just after write (B, b)
- b) Just after write (C, c)
- c) Just after  $\langle T_2 \text{, commit} \rangle$

The result of above 3 scenarios is as follows :

Initially the log and database will be

Log	Database
$\langle T_1, \text{Start} \rangle$	
$\langle T_1, A, 90 \rangle$	
$\langle T_1, B, 210 \rangle$	
$\langle T_1, \text{Commit} \rangle$	
	$A = 90$
	$B = 210$
$\langle T_2, \text{Start} \rangle$	
$\langle T_2, C, 280 \rangle$	
$\langle T_2, \text{Commit} \rangle$	
	$C = 280$

- a) Just after write (B, b)

Just after write operation, no commit record appears in log. Hence no write operation is performed on database. So database retains only old values. Hence  $A = 100$  and  $B = 200$  respectively.

*Ques.* The incomplete transaction of  $T_1$  can be deleted from log.

b) Just after write (C, c)

The state of log records is as follows

Note that crash occurs before  $T_2$  commits. At this point  $T_1$  is completed successfully, so new values of A and B are written from log to database. But as  $T_2$  is not committed, there is no redo ( $T_2$ ) and the incomplete transaction  $T_2$  can be deleted from log.

The redo ( $T_1$ ) is done as  $\langle T_1, \text{commit} \rangle$  gets executed. Therefore A = 90, B = 210 and C = 300 are the values for database.

c) Just after  $\langle T_2, \text{commit} \rangle$

The log records are as follows :

$\langle T_1, \text{Start} \rangle$

$\langle T_1, A, 90 \rangle$

$\langle T_1, B, 210 \rangle$

$\langle T_1, \text{Commit} \rangle$

$\langle T_2, \text{Start} \rangle$

$\langle T_2, 6, 280 \rangle$

$\langle T_2, \text{Commit} \rangle$

———— Crash occurs here

Clearly both  $T_1$  and  $T_2$  reached at commit point and then crash occurs. So both redo ( $T_1$ ) and redo ( $T_2$ ) are done and updated values will be A = 90, B = 210, C = 280.

## 2. Immediate Database Modification :

In this technique, the database is updated during the execution of transaction even before it reaches to its commit point.

If the transaction gets failed before it reaches to its commit point, then the a ROLLBACK Operation needs to be done to bring the database to its earlier consistent state. That means the effect of operations need to be undone on the database. For that purpose both Redo and Undo operations are both required during the recovery. This technique is known as UNDO/ REDO technique.

For example : Consider Two transaction  $T_1$  and  $T_2$  as follows :

$T_1$	$T_2$
Read(A,a)	Read(C,c)
$a=a-10$	$c=c-20$
Write(A,a)	Write(C,c)
Read(B,b)	
$b=b+10$	
Write(B,b)	

Here  $T_1$  and  $T_2$  are executed serially. Initially A=100, B=200 and C=300

If the crash occurs after

- i) Just after Write(B,b) ii) Just after Write(C,c) iii) Just after  $\langle T_2, \text{Commit} \rangle$

Then using the immediate Database modification approach the result of above three scenarios can be elaborated as follows :

The contents of log and database is as follows :

Log	Database
$\langle T_1, \text{Start} \rangle$	
$\langle T_1, A, 100, 90 \rangle$	A=90
$\langle T_1, B, 200, 210 \rangle$	B=210
$\langle T_1, \text{Commit} \rangle$	
$\langle T_2, \text{Start} \rangle$	
$\langle T_2, C, 300, 280 \rangle$	C=280
$\langle T_2, \text{Commit} \rangle$	

The recovery scheme uses two recovery techniques -

- i) **UNDO ( $T_i$ )** : The transaction  $T_i$  needs to be undone if the log contains  $\langle T_i, \text{Start} \rangle$  but does not contain  $\langle T_i, \text{Commit} \rangle$ . In this phase, it restores the values of all data items updated by  $T_i$  to the old values.
- ii) **REDO ( $T_i$ )** : The transaction  $T_i$  needs to be redone if the log contains both  $\langle T_i, \text{Start} \rangle$  and  $\langle T_i, \text{Commit} \rangle$ . In this phase, the data item values are set to the new values as per the transaction. After a failure has occurred log record is consulted to determine which transaction need to be redone.

- a) Just after Write (B,b) : When system comes back from this crash, it sees that there is  $\langle T_1, \text{Start} \rangle$  but no  $\langle T_1, \text{Commit} \rangle$ . Hence  $T_1$  must be undone. That means old values of A and B are restored. Thus old values of A and B are taken from log and both the transaction  $T_1$  and  $T_2$  are re-executed.
- b) Just after Write (C,c) : Here both the redo and undo operations will occur
- c) Undo : When system comes back from this crash, it sees that there is  $\langle T_2, \text{Start} \rangle$  but no  $\langle T_2, \text{Commit} \rangle$ . Hence  $T_2$  must be undone. That means old values of C is restored. Thus old value of C is taken from log and the transaction  $T_2$  is re-executed.
- d) Redo : The transaction  $T_1$  must be done as log contains both the  $\langle T_1, \text{Start} \rangle$  and  $\langle T_1, \text{Commit} \rangle$   
So A=90, B=210 and C=300
- e) Just after  $\langle T_2, \text{Commit} \rangle$  : When the system comes back from this crash, it sees that there are two transaction  $T_1$  and  $T_2$  with both start and commit points. That means  $T_1$  and  $T_2$  need to be redone. So A=90, B=210 and C=280

### 3.14 Save Points

The COMMIT, ROLLBACK, and SAVEPOINT are collectively considered as Transaction Commands

(1) **COMMIT** : The COMMIT command is used to save permanently any transaction to database.

When we perform, Read or Write operations to the database then those changes can be undone by rollback operations. To make these changes permanent, we should make use of commit

(2) **ROLLBACK** : The ROLLBACK command is used to undo transactions that have not already saved to database. For example

Consider the database table as

RollNo	Name
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Fig.3.14.1 Student Table

Following command will delete the record from the database, but if we immediately performs ROLLBACK, then this deletion is undone.

For instance –

```
DELETE FROM Student
WHERE RollNo =2;
ROLLBACK;
```

Then the resultant table will be

RollNo	Name
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

**(3) SAVEPOINT :** A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction. The SAVEPOINT can be created as

```
SAVEPOINT savepoint_name;
```

Then we can ROLLBACK to SAVEPOINT as

```
ROLLBACK TO savepoint_name;
```

For example – Consider Student table as follows –

RollNo	Name
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Fig.3.14.2 Student Table

Consider Following commands

```
SQL> SAVEPOINT S1
```

SQL>DELETE FROM Student

Where RollNo=2;

SQL> SAVEPOINT S2

SQL>DELETE FROM Student

Where RollNo=3;

SQL> SAVEPOINT S3

SQL>DELETE FROM Student

Where RollNo=4

SQL> SAVEPOINT S4

SQL>DELETE FROM Student

Where RollNo=5

SQL> ROLLBACK TO S3;

Then the resultant table will be

RollNo	Name
1	AAA
2	BBB
3	CCC

Thus the effect of deleting the record having RollNo 2, and RollNo3 is undone.

### 3.15 Isolation Levels

- The consistency of the database is maintained with the help of isolation property(one of the property from ACID properties ) of transaction.
- The transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system at particular instance.
- Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.
- There are four levels of transaction isolation defined by SQL -
- **Serializable :**
  - This is the Highest isolation level.
  - Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

- **Repeatable Read :**
  - This is the **most restrictive isolation level**.
  - The transaction holds **read locks on all rows** it references.
  - It holds **write locks on all rows** it inserts, updates, or deletes.
  - Since other transaction cannot read, update or delete these rows, it avoids <sup>non</sup> repeatable read.
- **Read Committed :**
  - This isolation level allows only **committed data to be read**.
  - Thus it does **not allow dirty read** (i.e. one transaction reading of data immediately after written by another transaction).
  - The transaction hold a **read or write lock** on the current row, and thus prevent other rows from reading, updating or deleting it.
- **Read Uncommitted :**
  - It is **lowest isolation level**.
  - In this level, one transaction may read not yet committed changes made by other transaction.
  - This level **allows dirty reads**.

In this level **transactions are not isolated** from each other.

### 3.16 SQL Facilities for Concurrency and Recovery

AU : May-14, Marks 8

It is possible to achieve concurrency control and recovery using SQL statements. Following is an illustration of implementation of isolation level

The different isolation levels can be set in SQL as follows

**(1) READ UNCOMMITTED :** It permits dirty reads, nonrepeatable reads, phantoms.

In SQL, we can write

**SET TRANSACTION**

**ISOLATION LEVEL READ UNCOMMITTED**

**(2) READ COMMITTED :** It permits nonrepeatable reads and phantoms, but prohibits dirty reads

In SQL, we can write

**SET TRANSACTION**

**ISOLATION LEVEL READ COMMITTED**

**(3) REPEATABLE READ :** It permits phantoms, but not dirty reads, nor nonrepeatable reads.

In SQL, we can write

**SET TRANSACTION**

**ISOLATION LEVEL REPEATABLE READ**

(4) **SERIALIZABLE** : In this isolation level there is true two phase locking, and it keeps all transactions serializable.

In SQL, we can write

**SET TRANSACTION**

**ISOLATION LEVEL SERIALIZABLE**

For example : Let us discuss the Implementation of READ COMMITTED isolation level using SQL facilities.

**Step 1 :** Create Table

```
CREATE TABLE Emp(ID int,Name char(50),Salary int)
```

**Step 2 :** Insert values into the table

```
INSERT INTO Emp(ID,Name,Salary)
```

```
VALUES (1,'Sharda',1000)
```

```
INSERT INTO Emp(ID,Name,Salary)
```

```
VALUES(2,'Ashwini',2000)
```

```
INSERT INTO Emp(ID,Name,Salary)
```

```
VALUES(3,'Madhura',3000)
```

The Table will be

ID	Name	Salary
1	Sharda	1000
2	Ashwini	2000
3	Madhura	3000

**Step 3 :** In select query it will take only committed values of table. If any transaction is opened and incompletely completed on table in others sessions then select query will wait till no transactions are pending on same table. Read Committed is the default transaction isolation level.

**Session 1:**

```
BEGIN TRAN
```

```
UPDATE emp SET Salary=999 WHERE ID=1
```

```
WAITFOR DELAY '00:00:15'
```

```
COMMIT
```

**Session 2 :**

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

```
SELECT Salary FROM emp WHERE ID=1
```

**Step 4 :** If we run above mentioned two sessions concurrently then

The output will be 999

**Step 5 :** Following SQL facilities demonstrate the **REPEATABLE READ** isolation level. In this technique, select query data of table that is used under transaction of isolation level "Repeatable Read" can not be modified from any other sessions till transaction is completed.

Assume that we have already created database and inserted some values in it.

**Session 1 :**

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

```
BEGIN TRAN
```

```
SELECT * FROM emp WHERE ID IN(1,2)
```

```
WAITFOR DELAY '00:00:15'
```

```
SELECT * FROM Emp WHERE ID IN (1,2)
```

```
ROLLBACK
```

**Session 2 :**

```
UPDATE emp SET Salary=999 WHERE ID=1
```

If we run above two sessions concurrently, then Update command in session 2 will wait till session 1 transaction is completed because emp table row with ID=1 has locked in session1 transaction.

**UNIT IV****IMPLEMENTATION TECHNIQUES**

**RAID – File Organization** – Organization of Records in Files – Indexing and Hashing – Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for SELECT and JOIN operations – Query optimization using Heuristics and Cost Estimation

**RAID**

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.

**RAID: Redundant Arrays of Independent Disks**

Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of high capacity and high speed by using multiple disks in parallel, and high reliability by storing data redundantly, so that data can be recovered even if a disk fails

**Motivation for RAID**

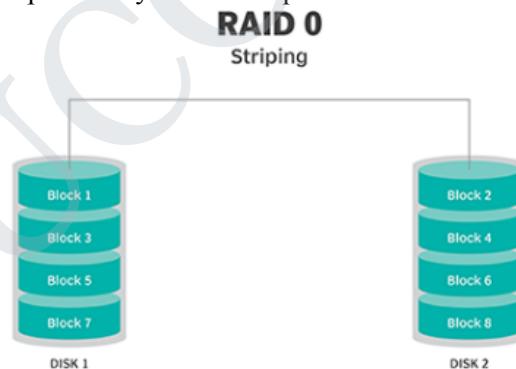
- Just as additional memory in form of cache, can improve the system performance, in the same way additional disks can also improve system performance.
- In RAID we can use an array of disks which operates independently since there are many disks, multiple I/O requests can be handled in parallel if the data required is on separate disks
- A single I/O operation can be handled in parallel if the data required is distributed across multiple disks.

**Benefits of RAID**

- Data loss can be very dangerous for an organization
- RAID technology prevents data loss due to disk failure
- RAID technology can be implemented in hardware or software
- Servers make use of RAID Technology

**RAID Level 0- Stripping and non-redundant**

- RAID level 0 divides data into block units and writes them across a number of disks. As data is placed across multiple disks it is also called “data Striping”.
- The advantage of distributing data over disks is that if different I/O requests are pending for two different blocks of data, then there is a possibility that the requested blocks are on different disks



There is no parity checking of data. So if data in one drive gets corrupted then all the data would be lost. Thus RAID 0 does not support data recovery. Spanning is another term that is used with RAID level 0 because the logical disk will span all the physical drives. RAID 0 implementation requires minimum 2 disks.

**Advantages**

- I/O performance is greatly improved by spreading the I/O load across many channels & drives.
- Best performance is achieved when data is striped across multiple controllers with only one driver per controller

**Disadvantages**

- It is not fault-tolerant, failure of one drive will result in all data in an array being lost

**RAID Level 1: Mirroring (or shadowing)**

- Also known as *disk mirroring*, this configuration consists of at least two drives that duplicate the storage of

data. There is no striping.

- Read performance is improved since either disk can be read at the same time. Write performance is the same as for single disk storage.
- Every write is carried out on both disks. If one disk in a pair fails, data still available in the other.
- Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired. Probability of combined event is very small.

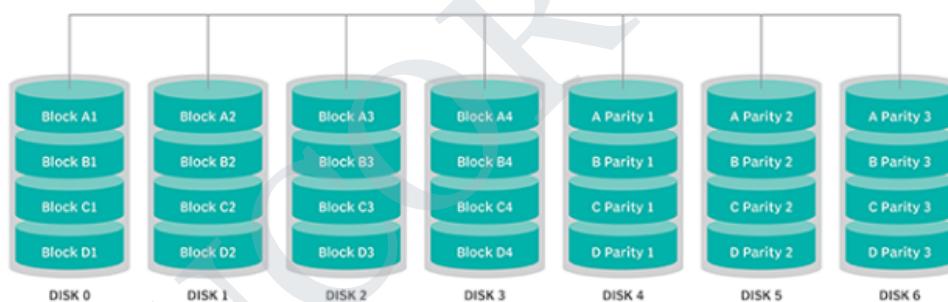
**RAID 1**  
Mirroring



#### RAID Level 2:

This configuration uses striping across disks, with some disks storing error checking and correcting (ECC) information. It has no advantage over RAID 3 and is no longer used.

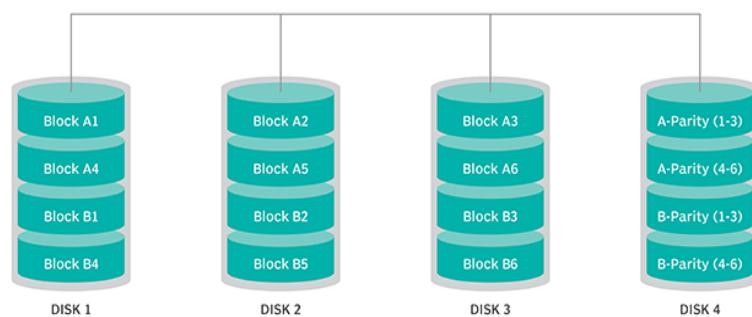
**RAID 2**



#### RAID Level 3: Bit-Interleaved Parity

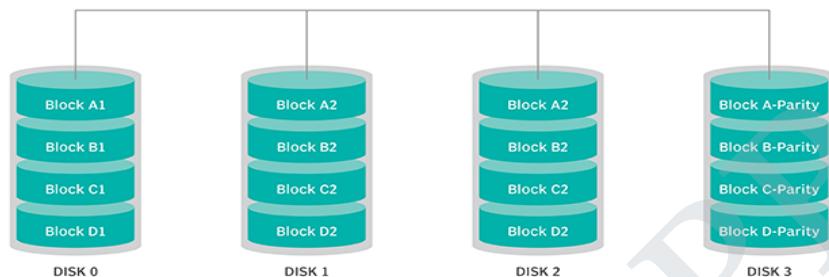
- A single parity bit is enough for error correction, not just detection, since we know which disk has failed
  - When writing data, corresponding parity bits must also be computed and written to a parity bit disk
  - To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)
- I/O operation addresses all the drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications.

**RAID 3**  
Parity on separate disk

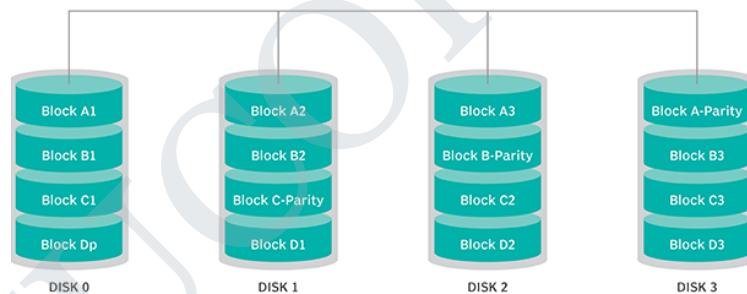


**RAID Level 4: Block-Interleaved Parity**

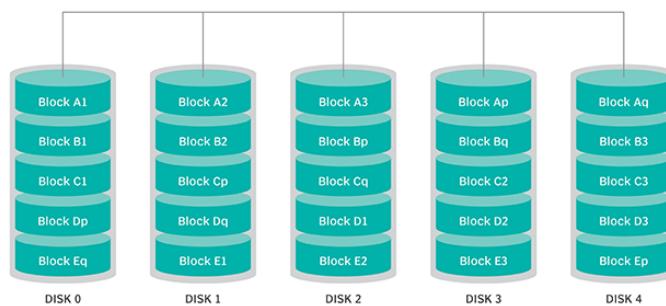
- When writing data block, corresponding block of parity bits must also be computed and written to parity disk
- To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.

**RAID 4****RAID Level 5:**

- RAID 5 uses striping as well as parity for redundancy. It is well suited for heavy read and low write operations.
- Block-Interleaved Distributed Parity; partitions data and parity among all  $N + 1$  disks, rather than storing data in  $N$  disks and parity in 1 disk.

**RAID 5****RAID Level 6:**

- This technique is similar to RAID 5, but includes a second parity scheme that is distributed across the drives in the array. The use of additional parity allows the array to continue to function even if two disks fail simultaneously. However, this extra protection comes at a cost.
- P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
  - Better reliability than Level 5 at a higher cost; not used as widely.

**RAID 6**

**File Organization**

- The database is stored as a collection of *files*.
- Each file is a sequence of *records*.
- A record is a sequence of fields.
- Classifications of records
  - Fixed length record
  - Variable length record
- Fixed length record approach:  
Assume record size is fixed each file has records of one particular type only different files are used for different relations

**Simple approach**

- Record access is simple

Example pseudo code

type account = record

```

account_number char(10);
branch_name char(22);
balance numeric(8);
end

```

Total bytes 40 for a record

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

**Two problems**

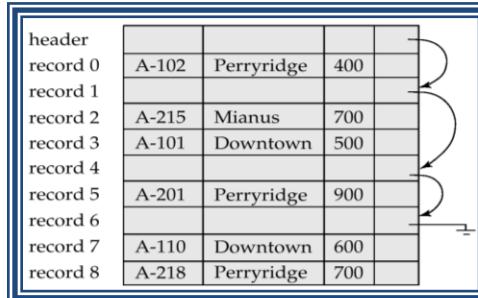
- Difficult to delete record from this structure.
- Some record will cross block boundaries, that is part of the record will be stored in one block and part in another. It would require two block accesses to read or write

Reuse the free space alternatives:

- move records  $i + 1, \dots, n$  to  $n, i, \dots, n - 1$
- do not move records, but link all free records on a free list
- Move the final record to deleted record place.

**Free Lists**

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on



**Variable-Length Records**

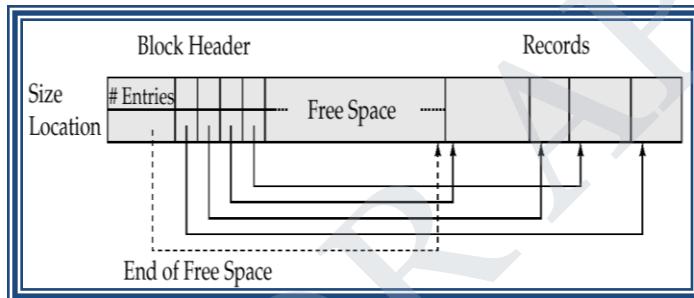
Byte string representation

- Attach an *end-of-record* ( $\perp$ ) control character to the end of each record
- Difficulty with deletion

0	perryridge	A-102	400	A-201	900	$\perp$
1	roundhill	A-305	350	$\perp$		
2	mianus	A-215	700	$\perp$		

**Disadvantage**

- It is not easy to reuse space occupied formerly by deleted record.
- There is no space in general for records grows longer

**Slotted Page Structure**

- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record

**Pointer Method**

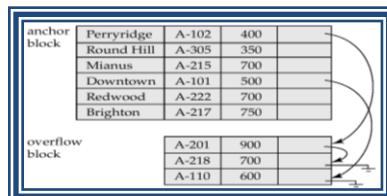
0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

- A variable-length record is represented by a list of fixed-length records, chained together via pointers.
- Can be used even if the maximum record length is not known.

Disadvantage to pointer structure; space is wasted in all records except the first in a chain.

Solution is to allow two kinds of block in file:

- Anchor block – contains the first records of chain
- Overflow block – contains records other than those that are the first records of chains.

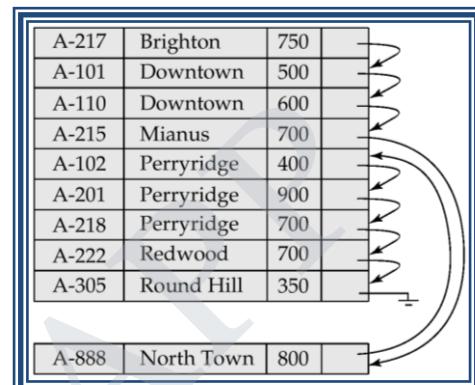
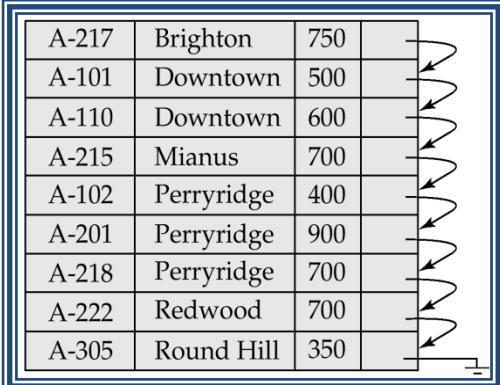


**Organization of Records in Files**

- Sequential – store records in sequential order, based on the value of the search key of each record
- Heap – a record can be placed anywhere in the file where there is space
- Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

**Sequential File Organization**

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key



**Deletion** – use pointer chains

**Insertion** – locate the position where the record is to be inserted

- if there is free space insert there
- if no free space, insert the record in an overflow block
- In either case, pointer chain must be updated

### **Indexing and Hashing**

**Basic Concepts**

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

Search-key	pointer
------------	---------

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” and by using a “hash function” the values are determined.

**Ordered Indices**

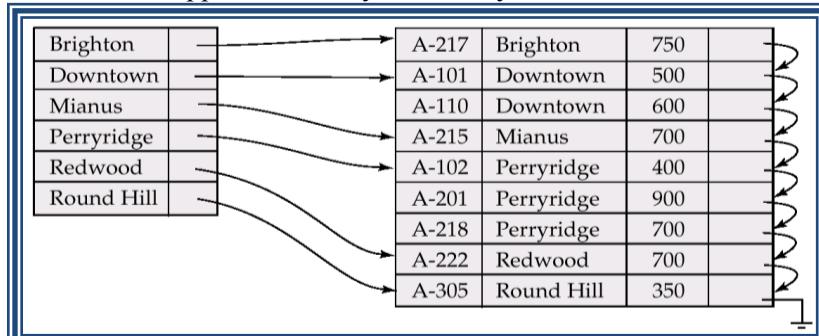
- In an **ordered index**, index entries are stored sorted on the search key value.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
- **Secondary index:** an index whose search key specifies an order different from the sequential order of the file.

**Types of Ordered Indices**

- Dense index
- Sparse index

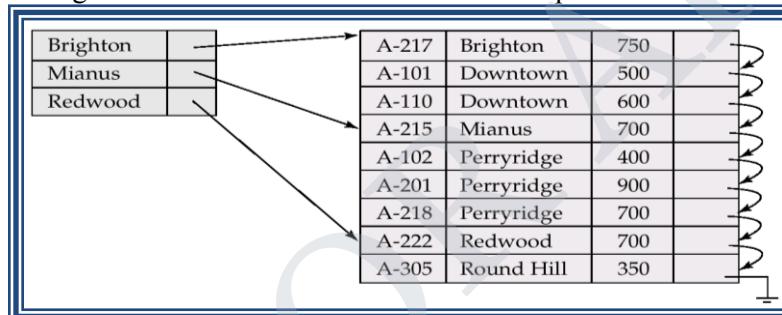
**Dense Index Files**

- Dense index — Index record appears for every search-key value in the file.



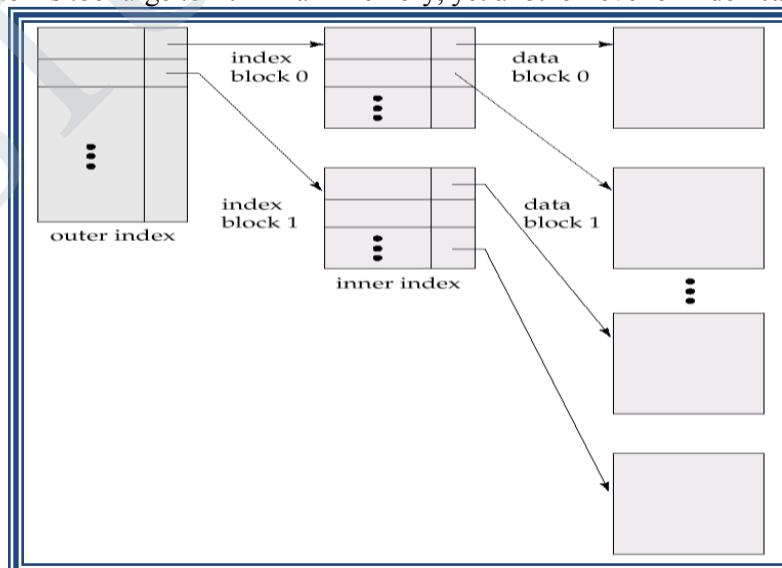
### Sparse Index Files

- Sparse Index
  - contains index records for only some search-key values.
- To locate a record with search-key value  $K$  we:
  - Find index record with largest search-key value that is less than or equal to  $K$
  - Search file sequentially starting at the record to which the index record points



### Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.



**Index Update: Deletion**

If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- **Single-level index deletion:**

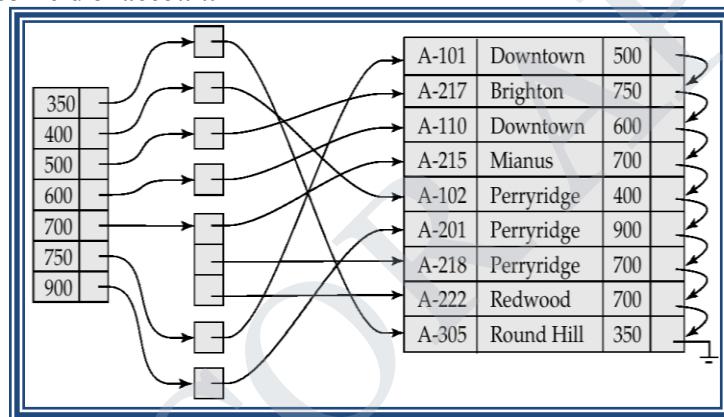
- Dense indices – deletion of search-key is similar to file record deletion.
- Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

**Index Update: Insertion**

- **Single-level index insertion:**

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices** – if the search-key value does not appear in the index, insert it.
- **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.

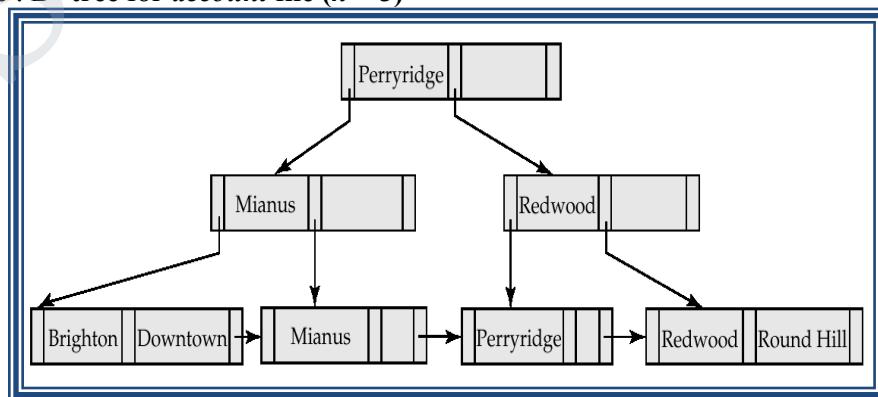
Secondary Index on *balance* field of *account*

**Primary and Secondary Indices**

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk

**B<sup>+</sup>-Tree Index Files**

Example of a B<sup>+</sup>-tree : B<sup>+</sup>-tree for *account* file (*n* = 3)



- **Disadvantage of indexed-sequential files:** performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- **Advantage of B<sup>+</sup>-tree index files:** automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- **Disadvantage of B<sup>+</sup>-trees:** extra insertion and deletion overhead, space overhead.

A B<sup>+</sup>-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between  $[n/2]$  and  $n$  children.
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf, it can have between 0 and  $(n-1)$  values.

### B<sup>+</sup>-Tree Node Structure

- **Typical node**

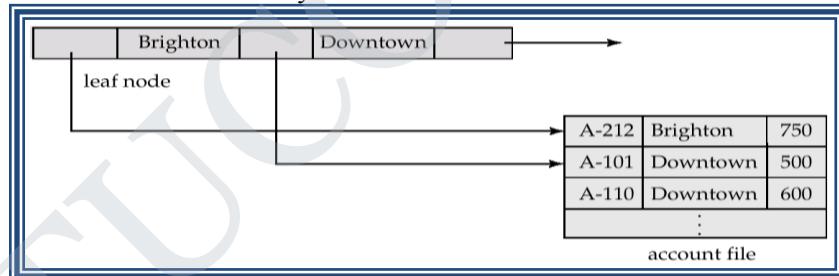
$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------

- $K_i$  are the search-key values
- $P_i$  are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered  

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

### Properties of Leaf Nodes

- For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  either points to a file record with search-key value  $K_i$ , or to a bucket of pointers to file records, each record having search-key value  $K_i$ .
- $P_n$  points to next leaf node in search-key order



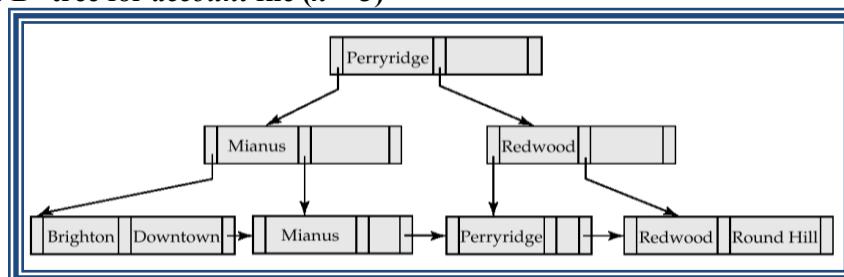
### Non-Leaf Nodes in B<sup>+</sup>-Trees

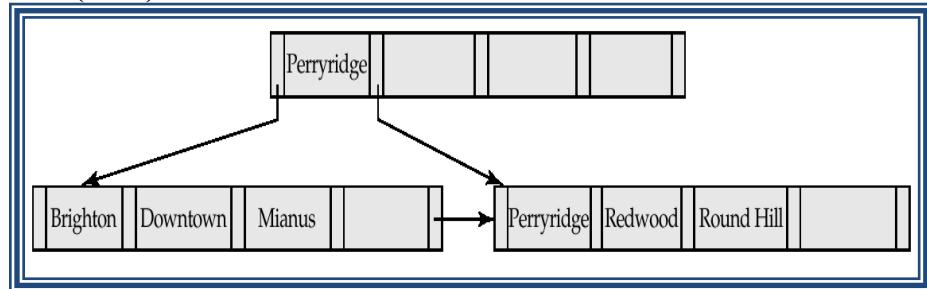
Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with  $m$  pointers:

- All the search-keys in the subtree to which  $P_1$  points are less than  $K_1$ .

$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------

Example of a B<sup>+</sup>-tree: B<sup>+</sup>-tree for **account file (n = 3)**



**B<sup>+</sup>-tree for account file (n = 5)**

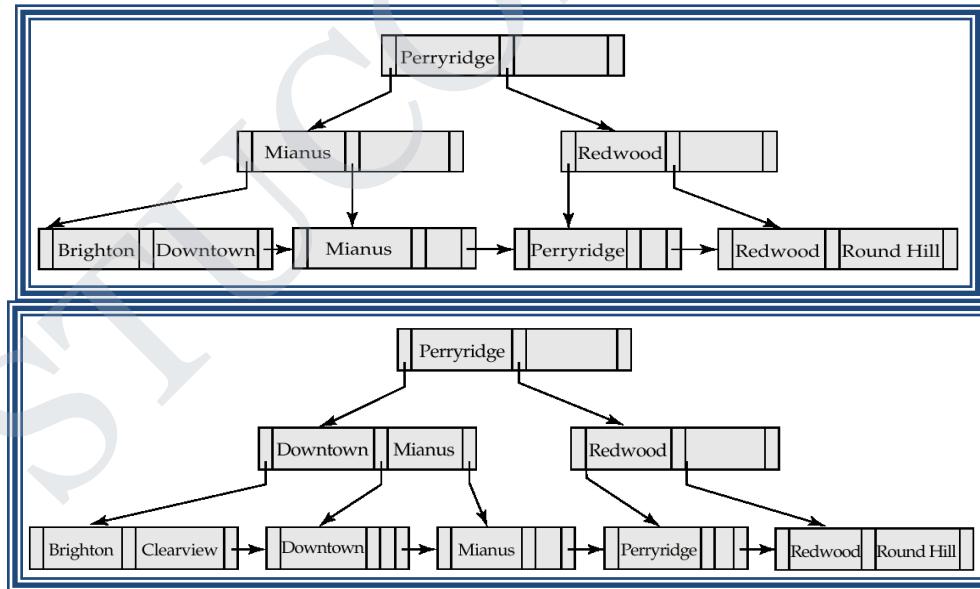
- Non-leaf nodes other than root must have between 3 and 5 children ( $\lceil n/2 \rceil$  and  $n$  with  $n=5$ ).
- Root must have at least 2 children.

**Observations about B<sup>+</sup>-trees**

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The B<sup>+</sup>-tree contains a relatively small number of levels thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently.

**Updates on B<sup>+</sup>-Trees: Insertion**

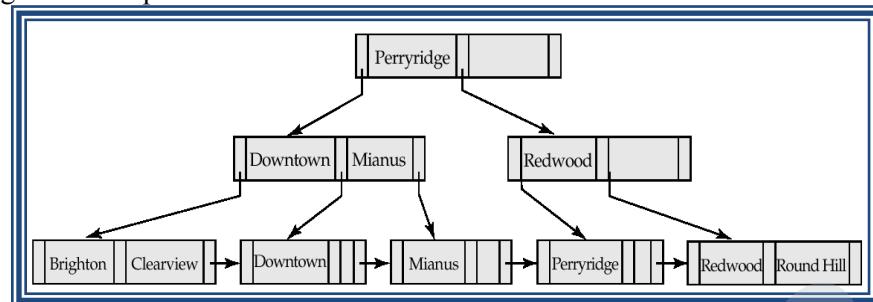
- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
- If the search-key value is not there, then add the record to the main file and create a bucket if necessary. Then:
  - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node otherwise, split the node.

**Example: B<sup>+</sup>-Tree before and after insertion of “Clearview”****Updates on B<sup>+</sup>-Trees: Deletion**

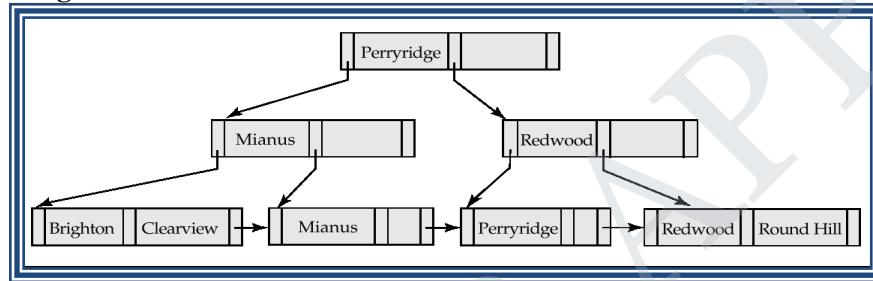
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete

the other node.

- Delete the pair  $(K_{i-1}, P_i)$ , where  $P_i$  is the pointer to the deleted node, from its parent, recursively using the above procedure.

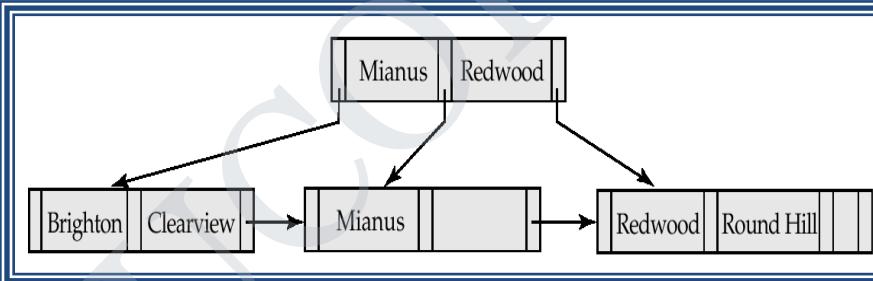


Before and after deleting “Downtown”



- The removal of the leaf node containing “Downtown” did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node’s parent.

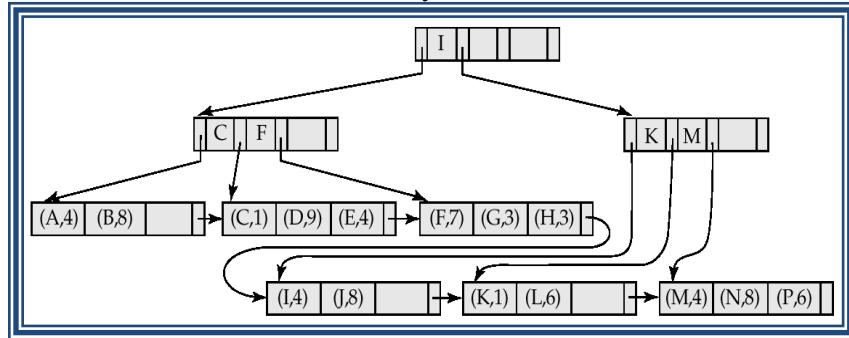
Deletion of “Perryridge” from result of previous example



- Node with “Perryridge” becomes empty and merged with its sibling.
- Root node then had only one child, and was deleted and its child became the new root node

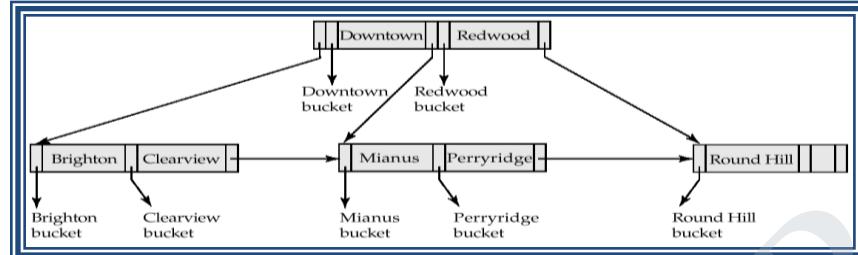
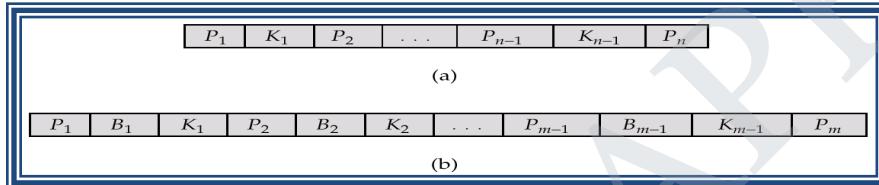
#### B<sup>+</sup>-Tree File Organization

- The leaf nodes in a B<sup>+</sup>-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B<sup>+</sup>-tree index.

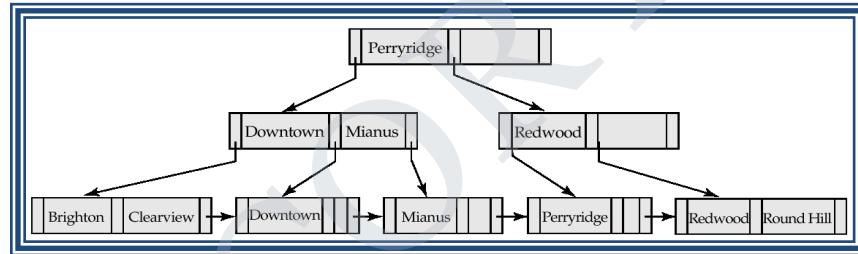


**B-Tree Index Files**

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.

**Generalized B-tree leaf node**

Nonleaf node – pointers  $B_i$  are the bucket or file record pointers.

**B+-tree on same data****Advantages of B-Tree indices:**

- May use less tree nodes than a corresponding B<sup>+</sup>-Tree.
- Sometimes possible to find search-key value before reaching leaf node.

**Disadvantages of B-Tree indices:**

- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced (no. of pointers). Thus, B-Trees typically have greater depth than corresponding B<sup>+</sup>-Tree
- Insertion and deletion more complicated than in B<sup>+</sup>-Trees
- Implementation is harder than B<sup>+</sup>-Trees.

**HASHING**

- Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.
- Hashing uses hash functions with search keys as parameters to generate the address of a data record.

**Hash Organization****Bucket**

A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

**Hash Function**

A hash function,  $h$ , is a mapping function that maps all the set of search-keys  $K$  to the address where actual records are placed. It is a function from search keys to bucket addresses.

- Worst hash function maps all search-key values to the same bucket.

- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal hash function is random, so each bucket will have the same number of records.

#### Types

- Static Hashing
- Dynamic Hashing

#### Static Hashing

- In static hashing, when a search-key value is provided, the hash function always computes the same address.
- For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function.
- The number of buckets provided remains unchanged at all times.

#### Example of Hash File Organization

- There are 10 buckets,
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g.  $h(\text{Perryridge}) = 5 \quad h(\text{Round Hill}) = 3 \quad h(\text{Brighton}) = 3$

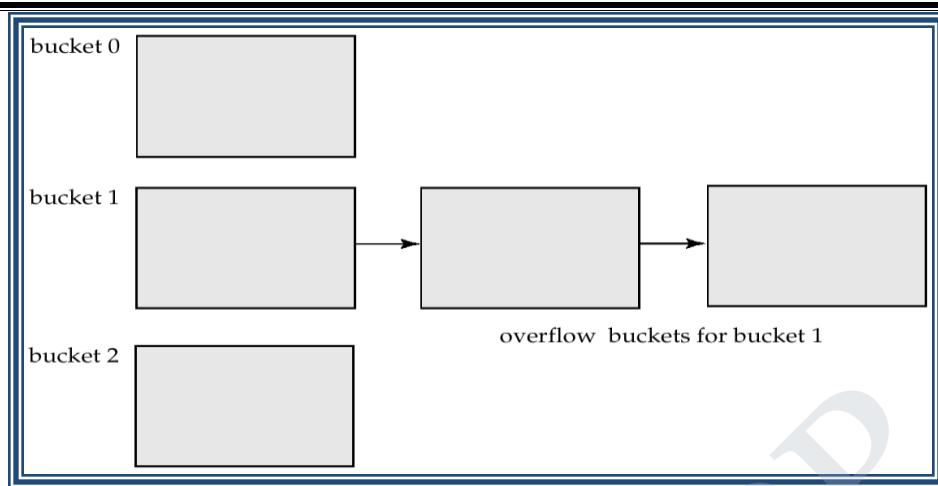
bucket 0				bucket 5	A-102	Perryridge	400
					A-201	Perryridge	900
					A-218	Perryridge	700
bucket 1				bucket 6			
bucket 2				bucket 7	A-215	Mianus	700
bucket 3				bucket 8	A-101	Downtown	500
					A-110	Downtown	600
bucket 4				bucket 9			

#### Operation

- **Insertion** – When a record is required to be entered using static hash, the hash function  $h$  computes the bucket address for search key  $K$ , where the record will be stored.  
Bucket address =  $h(K)$
- **Search** – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** – This is simply a search followed by a deletion operation.

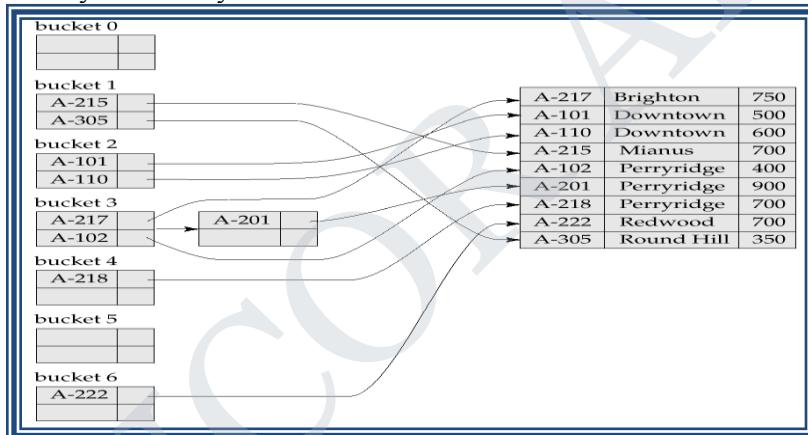
#### Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to :
    - multiple records have same search-key value
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using overflow buckets.
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.



### 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 record pointers, into a hash file structure.
- Hash indices are always secondary indices



### Deficiencies of Static Hashing

- In static hashing, function  $h$  maps search-key values to a fixed set of  $B$  of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
- These problems can be avoided by using techniques that allow the number of buckets to be **modified dynamically**.

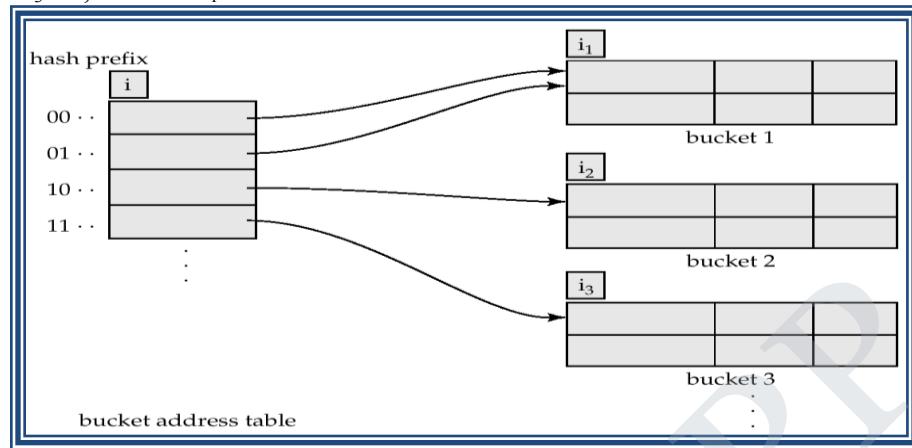
### Dynamic Hashing

- 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 bucket addresses.
  - Let the length of the prefix be  $i$  bits,  $0 \leq i \leq 32$ .
  - Bucket address table size =  $2^i$ . Initially  $i = 0$
  - Value of  $i$  grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket.
  - Thus, actual number of buckets is  $< 2^i$

- The number of buckets also changes dynamically due to coalescing and splitting of buckets.

### General Extendable Hash

In this structure,  $i_2 = i_3 = i$ , whereas  $i_1 = i - 1$



### Insertion in Extendable Hash Structure

To split a bucket  $j$  when inserting record with search-key value  $K_j$ :

- If  $i > i_j$  (more than one pointer to bucket  $j$ )
  - allocate a new bucket  $z$ , and set  $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to  $j$ , to point to  $z$
  - remove each record in bucket  $j$  and reinsert (in  $j$  or  $z$ )
  - recompute new bucket for  $K_j$  and insert record in the bucket (further splitting is required if the bucket is still full)
- If  $i = i_j$  (only one pointer to bucket  $j$ )
  - If  $i$  reaches some limit  $b$ , or too many splits have happened in this insertion, create an overflow bucket
- Else
  - increment  $i$  and double the size of the bucket address table.
  - replace each entry in the table by two entries that point to the same bucket.
  - recompute new bucket address table entry for  $K_j$

Now  $i > i_j$  so use the first case above.

### Deletion in Extendable Hash Structure

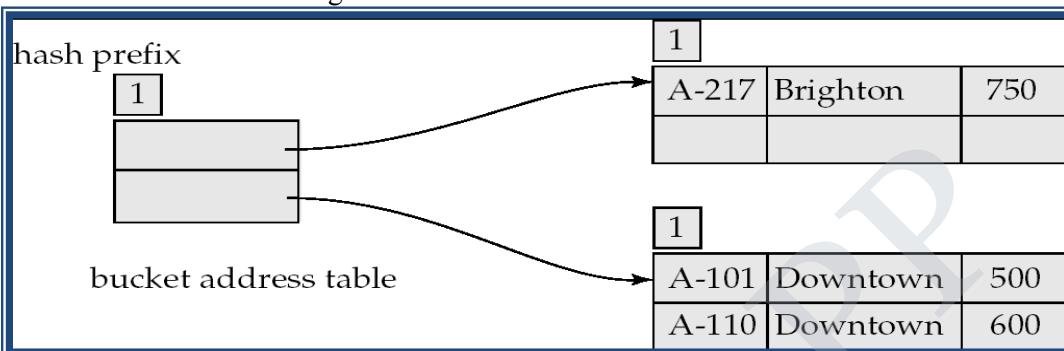
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of  $i_j$  and same  $i_j - 1$  prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

### Example

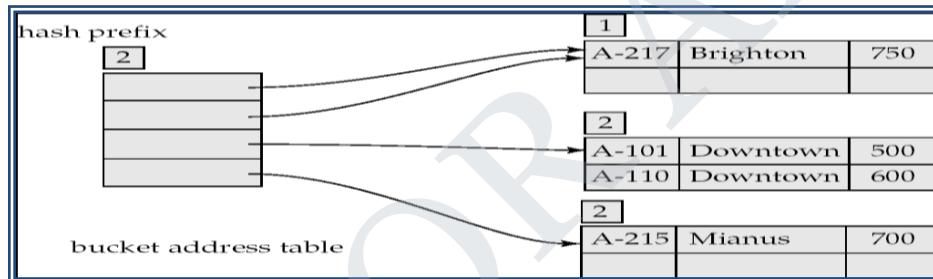
<i>branch_name</i>	$h(branch\_name)$
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



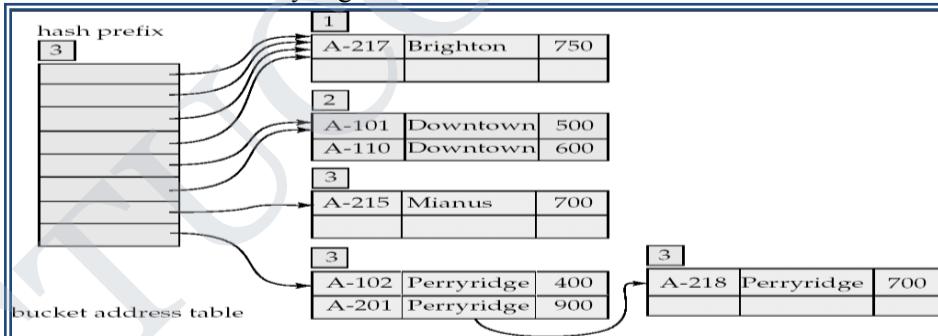
Hash structure after insertion of one Brighton and two Downtown records



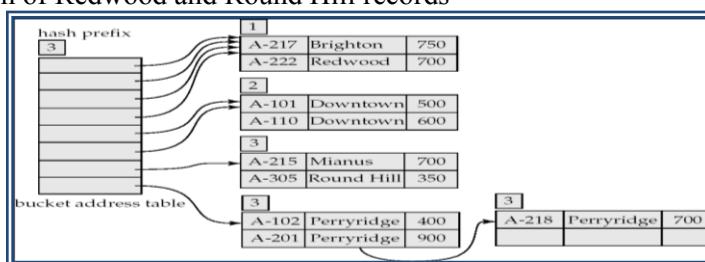
Hash structure after insertion of Mianus record



Hash structure after insertion of three Perryridge records



Hash structure after insertion of Redwood and Round Hill records



### Use of Extendable Hash Structure

- To locate the bucket containing search-key  $K_j$ :
  - Compute  $h(K_j) = X$
  - Use the first  $i$  high order bits of  $X$  as a displacement into bucket address table, and follow the pointer to

appropriate bucket

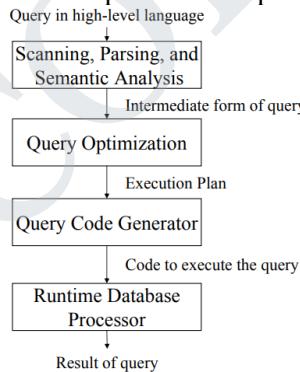
### Updates in Extendable Hash Structure

- **To insert a record with search-key value  $K_i$** 
  - follow same procedure as look-up and locate the bucket, say  $j$ .
  - If there is room in the bucket  $j$  insert record in the bucket.
  - Overflow buckets used instead in some cases.
- **To delete a key value,**
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty
  - Coalescing of buckets can be done
  - Decreasing bucket address table size is also possible
- **Benefits of extendable hashing:**
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- **Disadvantages of extendable hashing**
  - Extra level of indirection to find desired record

Bucket address table may itself become very big.

### QUERY PROCESSING OVERVIEW

1. The scanning, parsing, and validating module produces an internal representation of the query.
2. The query optimizer module devises an execution plan which is the execution strategy to retrieve the result of the query from the database files. A query typically has many possible execution strategies differing in performance, and the process of choosing a reasonably efficient one is known as query optimization. Query optimization is beyond this course. The code generator generates the code to execute the plan. The runtime database processor runs the generated code to produce the query result.



### Evaluation of SQL Statement

The query is evaluated in a different order.

- The tables in the from clause are combined using Cartesian products. The where predicate is then applied.
- The resulting tuples are grouped according to the group by clause. The having predicate is applied to each group, possibly eliminating some groups. The aggregates are applied to each remaining group. The select clause is performed last.

### Translating SQL Queries into Relational Algebra

- SQL query is first translated into an equivalent extended relational algebra expression.
- SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized.
- Query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses.
- Nested queries within a query are identified as separate query blocks.

**Example:**

```

SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > (SELECT MAX(SALARY)
 FROM EMPLOYEE
 WHERE DNO=5);

```

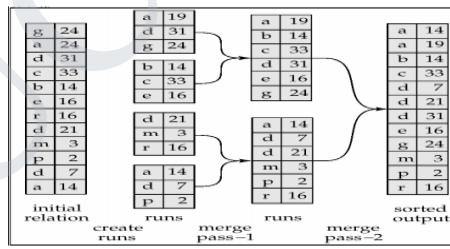
- The inner block
  - $(\text{SELECT MAX } (\text{SALARY}) \text{ FROM EMPLOYEE WHERE DNO}=5)$
  - Translated in:
  - $\exists \text{ MAX SALARY}(\sigma_{DNO=5}(\text{EMPLOYEE}))$
- The Outer block
  - $\text{SELECT LNAME, FNAME FROM EMPLOYEE WHERE SALARY} > C$
  - Translated in:
  - $\prod \text{LNAME, FNAME } (\sigma_{\text{SALARY}>C}(\text{EMPLOYEE}))$   
(C represents the result returned from the inner block.)
- The query optimizer would then choose an execution plan for each block.
- The inner block needs to be evaluated only once. (Uncorrelated nested query).
- It is much harder to optimize the more complex correlated nested queries.

### External Sorting

It refers to sorting algorithms that are suitable for large files of records on disk that do not fit entirely in main memory, such as most database files..

- ORDER BY.
- Sort-merge algorithms for JOIN and other operations (UNION, INTERSECTION). Duplicate elimination algorithms for the PROJECT operation (DISTINCT). Typical external sorting algorithm uses a sort-merge strategy: Sort phase: Create sort small sub-files (sorted sub-files are called runs).
- Merge phase: Then merges the sorted runs. N-way merge uses N memory buffers to buffer input runs, and 1 block to buffer output. Select the 1st record (in the sort order) among input buffers, write it to the output buffer and delete it from the input buffer. If output buffer full, write it to disk. If input buffer empty, read next block from the corresponding run..

### E.g. 2-way Sort-Merge



### Basic Algorithms for Executing Relational Query Operations

- An RDBMS must include one or more alternative algorithms that implement each relational algebra operation (SELECT, JOIN,...) and, in many cases, that implement each combination of these operations.
- Each algorithm may apply only to particular storage structures and access paths (such index,...).
- Only execution strategies that can be implemented by the RDBMS algorithms and that apply to the particular query and particular database design can be considered by the query optimization module.

### Algorithms for implementing SELECT operation

- These algorithms depend on the file having specific access paths and may apply only to certain types of selection conditions.
- We will use the following examples of SELECT operations:
  - (OP1): $\sigma_{SSN='123456789'}(\text{EMPLOYEE})$
  - (OP2): $\sigma_{DNUMBER > 5}(\text{DEPARTMENT})$

- (OP3): $\sigma$ DNO=5 (EMPLOYEE)
- (OP4): $\sigma$  DNO=5 AND SALARY>30000 AND SEX = 'F' (EMPLOYEE)
- (OP5): $\sigma$ ESSN='123456789' AND PNO=10 (WORKS\_ON)
- Many search methods can be used for simple selection: S1 through S6
- **S1: Linear Search (brute force) –full scan in Oracle's terminology-**
  - Retrieves every record in the file, and test whether its attribute values satisfy the selection condition: an expensive approach.
  - Cost: b/2 if key and b if no key
- **S2: Binary Search**
  - If the selection condition involves an equality comparison on a key attribute on which the file is ordered.
  - $\sigma$ SSN='1234567' (EMPLOYEE), SSN is the ordering attribute.
  - Cost:  $\log_2 b$  if key.
- **S3: Using a Primary Index (hash key)**
  - An equality comparison on a key attribute with a primary index (or hash key).
  - This condition retrieves a single record (at most).
  - Cost :primary index : bind/2 + 1(hash key: 1bucket if no collision).
- **S4: Using a primary index to retrieve multiple records**
  - Comparison condition is  $>$ ,  $\geq$ ,  $<$ , or  $\leq$  on a key field with a primary index
  - $\sigma$ DNUMBER >5(DEPARTMENT)
  - 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.
  - Method used for range queries too(i.e. queries to retrieve records in certain range)
  - Cost: bind/2 + ?. '?' could be known if the number of duplicates known.
- **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.
  - $\sigma$ DNO=5(EMPLOYEE)
  - Use the index to retrieve all the records satisfying the condition.
  - Cost:  $\log_2 \text{bind} + ?$ . '?' could be known if the number of duplicates known.
- **S6: Using a secondary (B+-tree) index on an equality comparison**
  - The method can be used to retrieve a single record if the indexing field is a key or to retrieve multiple records if the indexing field is not a key.
  - This can also be used for comparisons involving  $>$ ,  $\geq$ ,  $<$ , or  $\leq$ .
  - Method used for range queries too.
  - Cost to retrieve: a key= height + 1; a non key= height+1(extra-level)+?, comparison=(height-1)+?+?
- Many search methods can be used for complex selection which involve a Conjunctive Condition: S7 through as S9.
  - Conjunctive condition: several simple conditions connected with the AND logical connective.
  - (OP4): s DNO=5 AND SALARY>30000 AND SEX = 'F' (EMPLOYEE).
- **S7:Conjunctive selection using an individual index.**
  - If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the Methods S2 to S6, use that condition to retrieve the records.
  - Then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition
- **S8:Conjunctive selection using a composite index:**
  - If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined fields.
  - Example: If an index has been created on the composite key (ESSN, PNO) of the WORKS\_ON file,

we can use the index directly.

- (OP5):  $\sigma_{ESSN=123456789} \text{ AND } PNO=10 (\text{WORKS\_ON})$ .

- **S9: Conjunctive selection by intersection of record pointers**

- If the secondary indexes are available on more than one of the fields involved in simple conditions in the conjunctive 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 condition.
- If only some of the conditions have secondary indexes, each retrieval record is further tested to determine whether it satisfies the remaining conditions.

### Algorithms for implementing JOIN Operation

- **Join: time-consuming operation. We will consider only natural join operation**

- Two-way join: join on two files.
- Multiway join: involving more than two files.

- **The following examples of two-way JOIN operation (Rθ A=BS) will be used:**

- OP6: EMPLOYEE θ DNO=DNUMBER DEPARTMENT
- OP7: DEPARTMENT θ MGRSSN=SSN EMPLOYEE

- **J1: Nested-loop join (brute force)**

- 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  $t[A] = s[B]$ .

- **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 (e.g B of S), retrieve each record t in R, one at a time (single loop), and then use the access structure 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.
- 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.
- Pairs of file blocks are copied into memory buffers in order and records of each file are scanned only once each for matching with the other file if A & B are key attributes.
- The method is slightly modified in case where A and B are not key attributes.

- **J4: Hash-join**

- The records of files R and S are both hashed to the same hash file using the same hashing function on the join attributes A of R and B of S as hash keys.

- **Partitioning Phase**

- First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets.
- Assumption: The smaller file fits entirely into memory buckets after the first phase.
- (If the above assumption is not satisfied, the method is a more complex one and number of variations have been proposed to improve efficiency: partition has join and hybrid hash join.)

- **Probing Phase**

- A single pass through the other file (S) then hashes each of its records to probe appropriate bucket, and that record is combined with all matching records from R in that bucket.

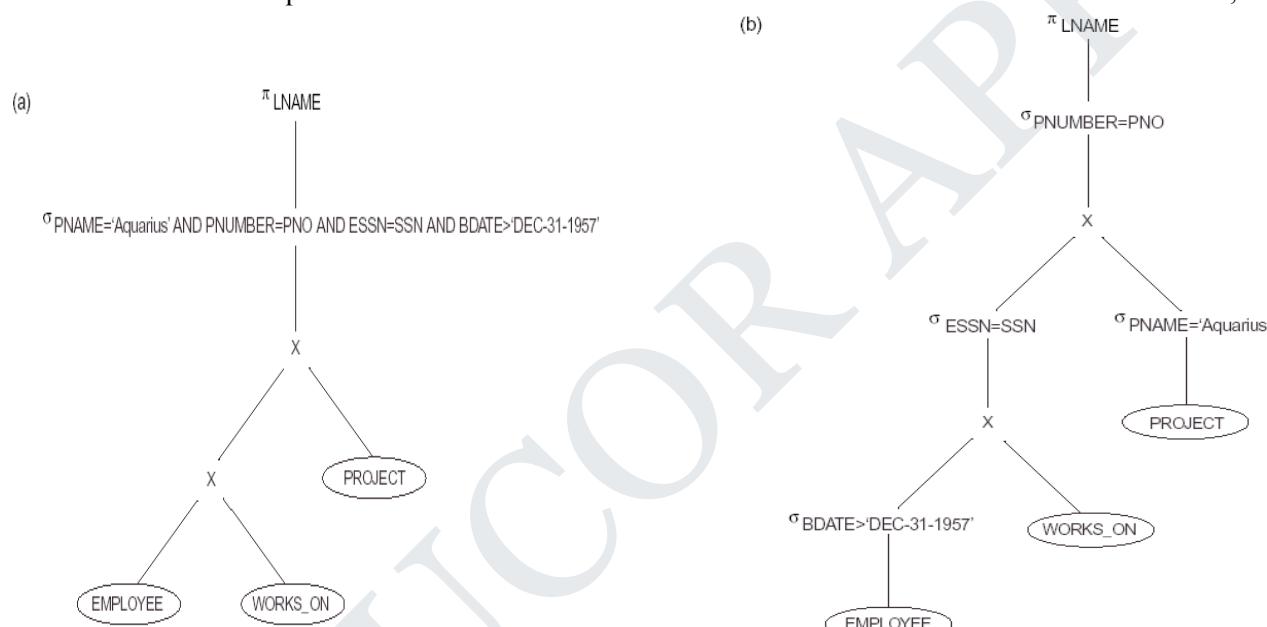
### Heuristic-Based Query Optimization

1. Break up SELECT operations with conjunctive conditions into a cascade of SELECT operations
2. Using the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition

3. Using commutativity and associativity of binary operations, rearrange the leaf nodes of the tree
4. Combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition
5. Using the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed
6. Identify sub-trees that represent groups of operations that can be executed by a single algorithm

**Example**

- Query  
"Find the last names of employees born after 1957 who work on a project named 'Aquarius'."
- SQL

**SELECT LNAME****FROM EMPLOYEE, WORKS\_ON, PROJECT****WHERE PNAME='Aquarius' AND PNUMBER=PNO AND ESSN=SSN AND BDATE>'1957-12-31';****Cost Estimation in Query Optimization**

The main aim of query optimization is to choose the most efficient way of implementing the relational algebra operations at the lowest possible cost.

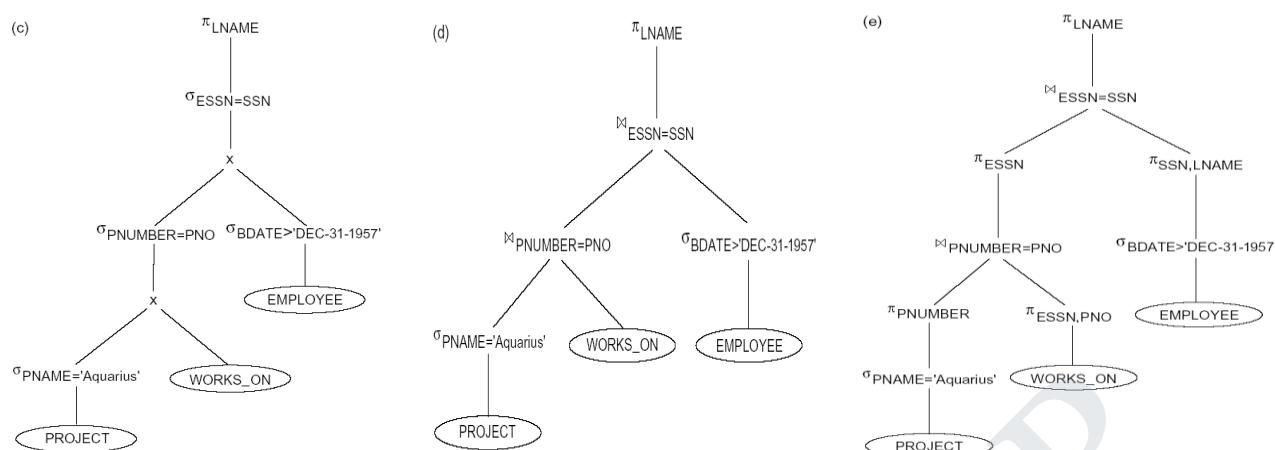
- The query optimizer should not depend solely on heuristic rules, but, it should also estimate the cost of executing the different strategies and find out the strategy with the minimum cost estimate.

The cost functions used in query optimization are estimates and not exact cost functions.

- The cost of an operation is heavily dependent on its selectivity, that is, the proportion of select operation(s) that forms the output.
- In general the different algorithms are suitable for low or high selectivity queries.
- In order for query optimizer to choose suitable algorithm for an operation an estimate of the cost of executing that algorithm must be provided

The cost of an algorithm depends on cardinality of its input.

- To estimate the cost of different query execution strategies, the query tree is viewed as containing a series of basic operations which are linked in order to perform the query.
- It is also important to know the expected cardinality of an operation's output because this forms the input to the next operation.



### Cost Components of Query Execution

The cost of executing the query includes the following components:

- Access cost to secondary storage.
- Storage cost.
- Computation cost.
- Memory uses cost.
- Communication cost.

### Importance of Access cost

Out of the above five cost components, the most important is the secondary storage access cost.

- The emphasis of the cost minimization depends on the size and type of database applications.
- For example in smaller database the emphasis is on the minimizing computing cost as because most of the data in the files involve in the query can be completely store in the main memory.
- For large database, the main emphasis is on minimizing the access cost to secondary device.
- For distributed database, the communication cost is minimized as because many sites are involved
- for the data transfer.

### Cost functions for SELECT Operation

#### Linear Search:

- $[n\text{Blocks}(R)/2]$ , if the record is found.
- $[n\text{Blocks}(R)]$ , if no record satisfied the condition.

#### Binary Search :

- o  $[\log_2(n\text{Blocks}(R))]$ , if equality condition is on key attribute, because  $SCA(R) = 1$  in this case.
- o  $[\log_2(n\text{Blocks}(R))] + [SCA(R)/b\text{Factor}(R)] - 1$ , otherwise.

#### Equity condition on Primary key

- $[n\text{LevelA}(I) + 1]$

#### Equity condition on Non-Primary key

- $[n\text{LevelA}(I) + 1] + [n\text{Blocks}(R)/2]$

### Cost functions for JOIN Operation

Join operation is the most time consuming operation to process.

- An estimate for the size (number of tuples) of the file that results after the JOIN operation is required to develop reasonably accurate cost functions for JOIN operations.
- The JOIN operations define the relation containing tuples that satisfy a specific predicate F from the Cartesian product of two relations R and S.

**STUCOR - MADE FOR ANNA UNIVERSITY STUDENTS****DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
QUESTION BANK****Subject Code: CS8492****Subject Name : DATABASE MANAGEMENT SYSTEMS****Branch/Year/Sem: CSE/ II / IV****Batch:2018-2022****Academic Year:2019-2020(EVEN)**

<b>UNIT-1 RELATIONAL DATABASES</b>	
<b>S.NO</b>	<b>PART-A</b>
1	Define Database Management System and its applications. Database management system (DBMS) is a collection of interrelated data and a set of programs to access those data. Applications: Banking, Airlines, Universities, Credit card transactions, Tele communication, Finance, Sales, Manufacturing and Human resource.
2	List the advantages of DBMS The advantages of using a DBMS are a) Controlling redundancy b) Restricting unauthorized access c) Providing multiple user interfaces d) Enforcing integrity constraints. e) Providing backup and recovery
3	Classify the levels of abstraction. a) Physical level - Describes how the data are actually stored. b) Logical level - Describes what data are stored and what relationships exist among those data c) View level - Describes the part of entire database
4	What are the types of data models? a) Entity relationship model b) Relational model c) Hierarchical model d) Network model e) Object based data model (Object Oriented & Object relational)
5	Define storage manager. 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.
6	List out the functions of database administrator. a) Schema definition

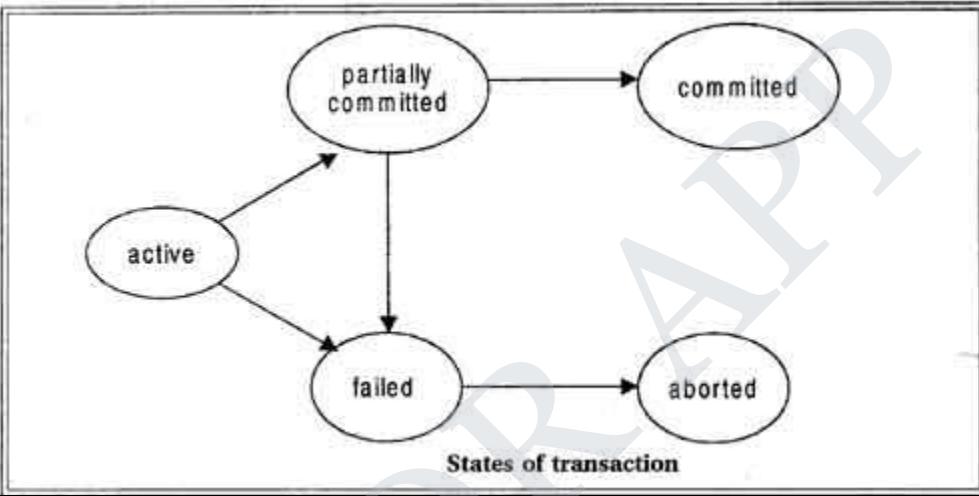
	b) Storage structure and access method definition c) Schema and physical organization modification d) Granting user authority to access the database e) Specifying integrity constraints f) Monitoring performance and responding to changes in requirements															
7	What is data dictionary? A data dictionary is a data structure which stores meta data about the structure of the database i.e. the schema of the database.															
8	What is a primary key? A primary key is an attribute that allows us to identify uniquely an entity in the entity set.															
9	Define relational algebra. The relational algebra is a procedural query language. It consists of a set of operations that take one or two relation as input and produce a new relation as output.															
10	Define foreign and candidate key A key in which primary key attribute of one table and same attribute be used in another table, then that key is known as foreign key.															
11	What are referential integrity constraints? A value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.															
12	Define SQL . Structured Query Language (SQL) is the standard common set used to communicate with the relational database management systems. All tasks related to relational data management— creating tables, querying the database for information, modifying the data in the database, deleting them, granting access to users, and so on — can be done using SQL.															
13	List the DML Commands used in SQL. SELECT: This command is used to retrieve rows from a table. The select syntax: SELECT [column name(s)] from [table name] where [conditions]. UPDATE: This command modifies data of one or more records. The update command syntax: UPDATE table name SET column name = value where [condition]. INSERT: This command adds one or more records to a database table. The insert command syntax: INSERT INTO table name [column(s)] VALUES [value(s)]. DELETE: This command removes one or more records from a table according to specified conditions. Delete command syntax: DELETE FROM table name where [condition].															
14	Compare delete command and truncate command. <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>S. No</th> <th>Delete</th> <th>Truncate</th> </tr> </thead> <tbody> <tr> <td>1.</td> <td>DELETE is a DML Command.</td> <td>TRUNCATE is a DDL command.</td> </tr> <tr> <td>2.</td> <td>specify filters in where clause</td> <td>Cannot use Where Condition</td> </tr> <tr> <td>3.</td> <td>deletes specified data if where condition exists</td> <td>Removes all the data</td> </tr> <tr> <td>4.</td> <td>Rollback is possible</td> <td>Rollback is not possible</td> </tr> </tbody> </table>	S. No	Delete	Truncate	1.	DELETE is a DML Command.	TRUNCATE is a DDL command.	2.	specify filters in where clause	Cannot use Where Condition	3.	deletes specified data if where condition exists	Removes all the data	4.	Rollback is possible	Rollback is not possible
S. No	Delete	Truncate														
1.	DELETE is a DML Command.	TRUNCATE is a DDL command.														
2.	specify filters in where clause	Cannot use Where Condition														
3.	deletes specified data if where condition exists	Removes all the data														
4.	Rollback is possible	Rollback is not possible														
15	What is Data Control Language? create privileges to allow users access to, and manipulation of, the database. There are two main commands:															

	GRANT to grant a privilege to a user REVOKE to revoke (remove) a privilege from a user
16	Define Dynamic SQL. Programs that contain embedded dynamic SQL statements must be precompiled like those that contain static SQL, but unlike static SQL, the dynamic statements are constructed and prepared at run time. The source form of a dynamic statement is a character string that is passed to DB2 by the program using the static SQL statement PREPARE or EXECUTE IMMEDIATE.
17	Define instance and schemas. April/May 2012 - Database change over times as information is inserted and deleted. - The collection of information stored in the database at a particular moment called an instance of the database. - The overall design of the database is called the database schema
18	List the aggregation functions in SQL Aggregate functions are functions that take a collection of values as input and return a single value as output. SQL offers 5 built in aggregate functions: avg - average value min - minimum value max - maximum value sum - sum of values count - number of values.
19	Define views. A view is an object that gives the user a logical view of data from an underlying table or tables (relation or relations). <sup>TM</sup> It is not desirable for all users to see the entire logical model.
20	What is Embedded SQL? An embedded SQL statement is distinguished from the host language statements by enclosing it between EXEC SQL or EXEC SQL BEGIN and a matching END-EXEC or EXEC SQL END (or semicolon) <sup>TM</sup> Syntax may vary with language <sup>TM</sup> Shared variables (used in both languages) usually prefixed with a colon (:) in SQL
<b>PART-B</b>	
1	Explain the basic architecture of a database management systems.
2	Discuss about the different types of model in DBMS.
3	With relevant example to discuss about the various operations of Relation Algebra.
4	Explain the different types of languages used in SQL with an example.
5	Discuss about the types of keys with an example.
6	Explain in detail about Advanced SQL and Embedded SQL.
7	Consider the employee database , where the primary keys are Underlined. employee(empname,street,city),works(empname,companynam, salary) company(companyname,city), manages(empname,management) Give an expression in the relational algebra for each request.

<b>UNIT II DATABASE DESIGN</b>	
<b>S.NO</b>	<b>PART-A</b>
1	<p>What is an entity-relationship model? 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.</p>
2	<p>What is an entity and entity set? An entity is an object that exists and is distinguishable from other objects. Example: specific person, company, event, plant Entity set: The set of all entities of the same type is termed as an entity set.</p>
3	<p>Define single valued and multivalued attributes. Single valued attributes : Attributes with a single value for a particular entity Multivalued attributes : Attributes with a set of value for a particular entity</p>
4	<p>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</p>
5	<p>Define Mapping cardinalities. Mapping cardinalities or cardinality ratios express the number of entities to which another entity can be associated. Mapping cardinalities must be one of the following:</p> <ul style="list-style-type: none"> <li>• One to one</li> <li>• One to many</li> <li>• Many to one</li> <li>• Many to many</li> </ul>
6	<p>Define the terms Generalization and Aggregation Generalization is a containment relationship that exists between a high-level Entity set and one or more low-level entity set. Aggregation is an abstraction through which relationships are treated as higher-level entities</p>
7	<p>Compare weak and strong entity sets. Weak entity set : entity set that do not have key attribute of their own are called weak entity sets. Strong entity set : Entity set that has a primary key is termed a strong entity set</p>
8	<p>What is ER diagrams? An entity-relationship diagram is a data modeling technique that creates a graphical representation of the entities, and the relationships between entities, within an information system.</p>
9	<p>What are the steps involved in creating ERD?</p> <ol style="list-style-type: none"> <li>a) Identify the entities.</li> <li>b) Find the relationships</li> <li>c) Identify key attributes for every entity</li> <li>d) Draw the ERD.</li> </ol>
10	<p>What is normalization? Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and eliminate dependency. Normalization usually involves dividing large tables into smaller (and less redundant) tables and defining relationships between them.</p>
11	<p>Define 1NF. A relation said to be first normal form if and if only all attributes are atomic in nature</p>
12	<p>Define 2NF. A relation said to be second normal form if and if only relation should be in first normal form and to</p>

	eliminate partial dependency
13	<p>Define 3NF.</p> <p>A relation said to be third normal form if and only if every relation should be in second normal form and to eliminate transitive dependency.</p>
14	<p>Define BCNF.</p> <p>A relational schema R is in Boyce-Codd normal form if and only if for every one of its dependencies <math>X \rightarrow Y</math>, at least one of the following conditions hold</p> <ul style="list-style-type: none"> <li>™ <math>X \rightarrow Y</math> is a trivial functional dependency (<math>Y \subseteq X</math>)</li> <li>™ X is a superkey for schema R</li> </ul>
15	<p>What is 4NF and 5NF?</p> <p>A Table is in 4NF if and only if, for every one of its non-trivial multivalued dependencies <math>X \rightarrow Y</math>, X is a superkey that is, X is either a candidate key or a superset</p> <p>A table is said to be in the 5NF if and only if every non-trivial join dependency in it is implied by the candidate keys.</p>
16	<p>Define De-normalization.</p> <p>De-normalization is the process of attempting to optimize the read performance of a database by adding redundant data or by grouping data.</p>
17	<p>What is sub-class and super class?</p> <p>For example, members of entity Employee can be grouped further into Secretary, Engineer, Manager, Technician, Salaried_Employee.</p> <p>The set listed is a subset of the entities that belong to the Employee entity, which means that every entity that belongs to one of the sub sets is also an Employee.</p> <p>Each of these sub-groupings is called a subclass, and the Employee entity is called the super-class.</p>
18	<p>List out the constraints on Specialization and Generalization.</p> <p>Several specializations can be defined on an entity type. ™ Entities may belong to subclasses in each of the specializations. ™ The specialization may also consist of a single subclass, such as the manager specialization, in this case we don't use the circle notation.</p>
	<b>PART-B</b>
1	Explain about Entity relationship model with ER diagram and example.
2	Draw an ER diagram for Banking System and Payroll system.
3	What is functional dependency? Explain the types of it with an example.
4	What is normalization? Explain the different types of normal forms with an example.
5	Draw an ER diagram for Hospital Management System.
6	Construct ER-Relation mapping diagram for College management system.
7	Define generalization and aggregation. Demonstrate it using ER diagram.
8	Compare the different types of normal form in detail.

<b>UNIT III TRANSACTIONS</b> <b>PART-A</b>	
1	What are the ACID properties? Atomicity, Consistency, Isolation, Durability is a set of properties that guarantee database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even though that might involve multiple changes (such as debiting one account and crediting another), is a single transaction.
2	What is transaction? <u>Collections of operations that form a single logical unit of work</u> are called transactions.
3	What are the two statements regarding transaction? The two statements regarding transaction of the form: Begin transaction and End transaction
4	What are the two types of serializability? A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence gives rise to the notions of: <ul style="list-style-type: none"> <li>• Conflict serializability</li> <li>• View serializability</li> </ul>
5	List the SQL statements used for transaction control. <i>Commit</i> : Saves all transactions <i>Rollback</i> : Used to undo transactions <i>Savepoint</i> : Establishes a point back to which you may roll <i>Set Transaction</i> : Establishes properties for current transaction.
6	What is Conflict-Serializability? Conflict-Serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).
7	What is View-Serializability? View-Serializability of a schedule is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that respective transactions in the two schedules read and write the same data values ("view" the same data values)
8	What are two pitfalls (problem) of lock-based protocols? <i>Shared Lock</i> : If a Transaction Ti has obtained a shared mode lock on data item Q, then Ti can read, but cannot write Q. <i>Exclusive Lock</i> : If the transaction Ti has obtained a shared mode Lock on item Q, then Ti can read and also write Q.
9	What is meant by deadlock? A system is in a deadlock state if there exists a set of transaction such that every transaction in the set is waiting for another transaction in the set.
10	Define the phases of two phase locking protocol <i>Growing phase</i> : a transaction may obtain locks but not release any lock. <i>Shrinking phase</i> : a transaction may release locks but may not obtain any new locks.
11	What is Time-stamp based protocol? Timestamp based protocol ensures Serializability. It selects an ordering among transactions in advance using time stamps. With each Transaction in the system, a unique fixed timestamp is associated. It is denoted by TS(Ti). This

	timestamp is assigned by the database system before the transaction $T_i$ status execution. If a transaction $T_i$ has been assigned time- stamp $TS(T_i)$ and new transaction $T_j$ enters the system, then $TS(T_i) < TS(T_j)$ .
12	What is Concurrency Control? Process of managing simultaneous execution of transactions in a shared database, to ensure the Serializability of transactions, is known as concurrency control.
13	List out the statements associated with a database transaction? 6. <ul style="list-style-type: none"> <li>• Commit work . </li> <li>• Rollback work</li> </ul>
14	Draw the states of Transaction.  <pre> graph TD     active((active)) --&gt; partiallycommitted((partially committed))     partiallycommitted --&gt; committed((committed))     active --&gt; failed((failed))     failed --&gt; aborted((aborted))     failed --&gt; partiallycommitted     </pre> <p style="text-align: center;"><b>States of transaction</b></p>
15	Difference between deadlock prevention and deadlock avoidance.  <b>Deadlock Prevention:</b> <ul style="list-style-type: none"> <li>• Preventing deadlocks by constraining how requests for resources can be made in the system and how they are handled (system design).</li> <li>• The goal is to ensure that at least one of the necessary conditions for deadlock can never hold.</li> </ul> <b>Deadlock Avoidance:</b> <ul style="list-style-type: none"> <li>• The system dynamically considers every request and decides whether it is safe to grant it at this point,</li> <li>• The system requires additional apriori information regarding the overall potential use of each resource for each process.</li> <li>• Allows more concurrency.</li> </ul>
<b>PART-B</b>	
1	Explain about ACID properties with suitable example.
2	Describe briefly about Serializability and its types with relevant example.
3	Discuss the following transaction with relevant example: <ol style="list-style-type: none"> <li>I. Read Only Transaction</li> <li>II. Read Write Transaction</li> <li>III. Aborted Transaction</li> </ol>
4	Discuss in detail about two phase commit protocol.
5	Explain the concepts of Concurrency control mechanism.
6	Describe briefly about deadlock handling.
7	Why concurrency control is needed? Explain the problems that would arise when concurrency control is not provided by the database system.

**UNIT IV –IMPLEMENTATION TECHNIQUES****PART-A**

1	<p>List out the levels of RAID.</p> <p>RAID level 0 –Block striping non-redundant</p> <p>RAID level 1 –Mirroring</p> <p>RAID level 2 –Memory style Error correcting code</p> <p>RAID level 3 –Bit interleaved parity</p> <p>RAID level 4 –Block interleaved parity</p> <p>RAID level 5 – Block interleaved distributed parity</p> <p>RAID level 6 – P+Q redundancy disk</p>																												
2	<p>List the merits and demerits of B+ tree index structure.</p> <p>Mertis</p> <p>Insertion in B+tree is easy.</p> <p>Deletion in B+ tree is simple than B tree.</p> <p>Demerits :</p> <p>Requires redundant storage for search-key values every search key appears in some leaf node.</p> <p>Several are repeated in non-leaf node.</p> <p>Lookup on B+-tree requires traversal of a path from root of tree to some leaf node.</p>																												
3	<p>Differentiate Static hashing and Dynamic hashing.</p> <table border="1"> <thead> <tr> <th>S. No</th> <th>Static hashing</th> <th>Dynamic hashing</th> </tr> </thead> <tbody> <tr> <td>1.</td> <td>Numbers of buckets are fixed</td> <td>Numbers of buckets are not fixed</td> </tr> <tr> <td>2.</td> <td>As the file grows, performance decreases.</td> <td>As the file grows, performances do Not degrade</td> </tr> <tr> <td>3.</td> <td>Space overhead is more</td> <td>Space overhead is less</td> </tr> <tr> <td>4.</td> <td>Do not use bucket address table</td> <td>Bucket address table is used</td> </tr> <tr> <td>5.</td> <td>Open hashing and closed hashing are forms of it.</td> <td>Extendable hashing and linear hashing are forms of it.</td> </tr> <tr> <td>6.</td> <td>Implementation is simple</td> <td>Implementation is complex</td> </tr> <tr> <td>7.</td> <td>It is less attractive techniques</td> <td>It is more attractive techniques</td> </tr> <tr> <td>8.</td> <td>Overflow chaining is not used</td> <td>Overflow chaining is not used</td> </tr> </tbody> </table>		S. No	Static hashing	Dynamic hashing	1.	Numbers of buckets are fixed	Numbers of buckets are not fixed	2.	As the file grows, performance decreases.	As the file grows, performances do Not degrade	3.	Space overhead is more	Space overhead is less	4.	Do not use bucket address table	Bucket address table is used	5.	Open hashing and closed hashing are forms of it.	Extendable hashing and linear hashing are forms of it.	6.	Implementation is simple	Implementation is complex	7.	It is less attractive techniques	It is more attractive techniques	8.	Overflow chaining is not used	Overflow chaining is not used
S. No	Static hashing	Dynamic hashing																											
1.	Numbers of buckets are fixed	Numbers of buckets are not fixed																											
2.	As the file grows, performance decreases.	As the file grows, performances do Not degrade																											
3.	Space overhead is more	Space overhead is less																											
4.	Do not use bucket address table	Bucket address table is used																											
5.	Open hashing and closed hashing are forms of it.	Extendable hashing and linear hashing are forms of it.																											
6.	Implementation is simple	Implementation is complex																											
7.	It is less attractive techniques	It is more attractive techniques																											
8.	Overflow chaining is not used	Overflow chaining is not used																											
4	<p>What are the two types of ordered indices?</p> <ul style="list-style-type: none"> <li>• Primary index</li> <li>• Secondary index</li> </ul>																												
5	<p>What are different types of file organization?</p> <ul style="list-style-type: none"> <li>• Sequential file organization</li> <li>• Heap file organization</li> <li>• Hashing file organization</li> </ul>																												
6	<p>Difference between dense index and parse index.</p> <table border="1"> <thead> <tr> <th>S.No</th> <th>Dense index</th> <th>Parse index</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Index entry for every search key value (and hence every record) in the data file.</td> <td>Index entries for only some of the search values.</td> </tr> <tr> <td>2.</td> <td>Dense indices are faster than sparse</td> <td>Sparse indices are slower than dense</td> </tr> <tr> <td>3</td> <td>Require more space and impose more</td> <td>Require less space and impose less</td> </tr> </tbody> </table>		S.No	Dense index	Parse index	1	Index entry for every search key value (and hence every record) in the data file.	Index entries for only some of the search values.	2.	Dense indices are faster than sparse	Sparse indices are slower than dense	3	Require more space and impose more	Require less space and impose less															
S.No	Dense index	Parse index																											
1	Index entry for every search key value (and hence every record) in the data file.	Index entries for only some of the search values.																											
2.	Dense indices are faster than sparse	Sparse indices are slower than dense																											
3	Require more space and impose more	Require less space and impose less																											

		maintenance for insertion and deletion	maintenance for insertion and deletion																
7	What are the factors to evaluate the indexing technique?	<ul style="list-style-type: none"> <li>• Access types</li> <li>• Access time</li> <li>• Insertion time</li> <li>• Deletion time</li> <li>• Space overhead</li> </ul>																	
8	What is B+ tree?	A B+ tree is an n-ary tree with a variable but often large number of children per node. The root may be either a leaf or a node with two or more children. A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.																	
9	Difference between B+ tree and B tree.	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>S. No</th> <th>B+ tree</th> <th>B- tree</th> </tr> </thead> <tbody> <tr> <td>1.</td> <td>B+ tree leaf node data are ordered in a sequential linked list.</td> <td>B tree the leaf node cannot store using linked list.</td> </tr> <tr> <td>2.</td> <td>B +tree store redundant search key</td> <td>B tree store non-redundant search Key</td> </tr> <tr> <td>3.</td> <td>B+-tree data store only leaf nodes.</td> <td>B tree search keys and data stored in internal or leaf nodes</td> </tr> <tr> <td>4.</td> <td>Insertion of a B+ tree is not complicated</td> <td>Insertion of a B tree is more complicated</td> </tr> </tbody> </table>	S. No	B+ tree	B- tree	1.	B+ tree leaf node data are ordered in a sequential linked list.	B tree the leaf node cannot store using linked list.	2.	B +tree store redundant search key	B tree store non-redundant search Key	3.	B+-tree data store only leaf nodes.	B tree search keys and data stored in internal or leaf nodes	4.	Insertion of a B+ tree is not complicated	Insertion of a B tree is more complicated		
S. No	B+ tree	B- tree																	
1.	B+ tree leaf node data are ordered in a sequential linked list.	B tree the leaf node cannot store using linked list.																	
2.	B +tree store redundant search key	B tree store non-redundant search Key																	
3.	B+-tree data store only leaf nodes.	B tree search keys and data stored in internal or leaf nodes																	
4.	Insertion of a B+ tree is not complicated	Insertion of a B tree is more complicated																	
10	Define Seek time.	<p><i>Seek time:</i> Time it takes to reposition the arm over the correct track. The seek time ranges from 2 to 30 milliseconds. The average seek time is one-third the worst case seek time and one half the maximum seek time. Average seek time currently ranges between 4 to 10 milliseconds.</p>																	
11	Define File organization.	<p>The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields. We have a file with 2 types of records.</p> <p>Fixed length records. </p> <p>Variable length records.</p>																	
12	What are the two main goals of parallelism?	<p>Load –balance multiple small accesses, so that the throughput of such accesses increases.</p> <p>Parallelize large accesses so that the response time of large accesses is reduced</p>																	
13	What are the factors to evaluate the indexing technique?	<ul style="list-style-type: none"> <li>• Access types</li> <li>• Access time</li> <li>• Insertion time</li> <li>• Deletion time</li> <li>• Space overhead</li> </ul>																	
14	What are the two types of blocks in the fixed-length representation?	<p><i>Anchor block:</i> Contains the first record of a chain.</p> <p><i>Overflow block:</i> Contains the records other than those that are the first record o f a chain.</p>																	

15	List the five methods of file organization <ul style="list-style-type: none"> <li>• Sequential organization.</li> <li>• Indexed-sequential organization.</li> <li>• Direct or Hash organization.</li> <li>• Heap organization.</li> <li>• Clustered Organization</li> </ul>															
16	What are the two indices in files? There are 2 basic kinds of indices.  Ordered indices - Based on a stored ordering of the values  Hash indices - Based on a uniform distribution of values across a range of buckets.															
17	List the advantages and disadvantages of B+ tree.  Advantage of B+-tree index files <ul style="list-style-type: none"> <li>• Automatically reorganizes itself with small, local changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.</li> </ul> Disadvantage of B+-tree <ul style="list-style-type: none"> <li>• Extra insertion and deletion overhead, space overhead.</li> </ul>															
18	Difference between Primary index and Secondary index.6. <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>S. No</th> <th>Primary index</th> <th>Secondary index</th> </tr> </thead> <tbody> <tr> <td>1.</td> <td>An index on a set of fields that includes the unique primary key for the field</td> <td>An index that is not a primary key</td> </tr> <tr> <td>2.</td> <td>Guaranteed not to contain duplicates</td> <td>May have duplicates</td> </tr> <tr> <td>3.</td> <td>Also Called a Clustered index.</td> <td>Also Called a Non-Clustered index.</td> </tr> <tr> <td>4.</td> <td>Eg: Employee ID</td> <td>Eg: Employee name</td> </tr> </tbody> </table>	S. No	Primary index	Secondary index	1.	An index on a set of fields that includes the unique primary key for the field	An index that is not a primary key	2.	Guaranteed not to contain duplicates	May have duplicates	3.	Also Called a Clustered index.	Also Called a Non-Clustered index.	4.	Eg: Employee ID	Eg: Employee name
S. No	Primary index	Secondary index														
1.	An index on a set of fields that includes the unique primary key for the field	An index that is not a primary key														
2.	Guaranteed not to contain duplicates	May have duplicates														
3.	Also Called a Clustered index.	Also Called a Non-Clustered index.														
4.	Eg: Employee ID	Eg: Employee name														
<b>PART-B</b>																
1	What are the various ways of organizing records in files and explain any one file organization in detail.															
2	Describe the structure of B+ tree and list the characteristics of B+ tree.															
3	What is RAID? Briefly explain different level of RAID.															
4	Give brief notes on overviews of physical storage media.															
5	Describe briefly about indexing and hashing.															
6	Explain about query processing in detail.															
7	Discuss about query optimization in detail.															

<b>UNIT V – ADVANCED TOPICS</b>													
<b>PART-A</b>													
1	<p>What are the two approaches to store a relation in the distributed database?</p> <p><b>Replication:</b> System maintains several identical replicas (copies) of the relation and stores each replica at a different site.</p> <p><b>Fragmentation:</b> System Partitions the relation into several fragments and stores each fragment at a different site.</p>												
2	<p>Define Distribute Database.</p> <p>The computers in a distributed system communicate with one another through various communication media, such as high-speed networks or telephone lines. They do not share main memory or disk. The computers in distributed system are referred by names such as sites or nodes.</p>												
3	<p>What are the types of Transactions?</p> <p>Distributed database system supports two types of transactions.</p> <p><i>Local transaction :</i> It is one that accesses data only from site where that transaction was initiated.</p> <p><i>Global transaction :</i> It is one that either accesses data from a site other than the site where that transaction was initiated or accesses data from several different sites.</p>												
4	<p>Difference between homogeneous and heterogeneous database.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">S. No</th> <th style="text-align: center;">Homogeneous Database</th> <th style="text-align: center;">Heterogeneous Database</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">1.</td> <td>Different nodes may have same hardware &amp; software</td> <td>Database application used at each location must be same or compatible.</td> </tr> <tr> <td style="text-align: center;">2.</td> <td>Much easier to design and manage</td> <td>Tough to design and manage</td> </tr> <tr> <td style="text-align: center;">3.</td> <td>Database application used at each location must be same or compatible</td> <td>Database application used at each location must be incompatible.</td> </tr> </tbody> </table>	S. No	Homogeneous Database	Heterogeneous Database	1.	Different nodes may have same hardware & software	Database application used at each location must be same or compatible.	2.	Much easier to design and manage	Tough to design and manage	3.	Database application used at each location must be same or compatible	Database application used at each location must be incompatible.
S. No	Homogeneous Database	Heterogeneous Database											
1.	Different nodes may have same hardware & software	Database application used at each location must be same or compatible.											
2.	Much easier to design and manage	Tough to design and manage											
3.	Database application used at each location must be same or compatible	Database application used at each location must be incompatible.											
5	<p>What are the advantages &amp; disadvantages of distributed databases?</p> <p><i>Advantages :</i></p> <ul style="list-style-type: none"> <li>• Sharing data</li> <li>• Autonomy</li> <li>• Availability</li> </ul> <p><i>Disadvantages :</i></p> <ul style="list-style-type: none"> <li>• Software development cost</li> <li>• Greater potential for bugs</li> <li>• Increased processing overhead</li> </ul>												
6	<p>Define Transparency.</p> <p>The user of a distributed database system should not be required to know either where the data are physically located or how the data can be accessed at the specific local site. This characteristic called data transparency.</p>												
7	<p>What is Object Oriented Database?</p> <p>An object-oriented database is a database that subscribes to a model with information represented by objects. Object-oriented databases are a niche offering in the relational database management system (RDBMS) field and are not as successful or well-known as mainstream database engines.</p>												

8	What is Object? Object consists of entity and attributes which can describe the state of real world object and action associated with that object.																											
9	What are the three types of attributes in OODB? <ul style="list-style-type: none"><li>• Simple Attributes</li><li>• Complex Attributes</li><li>• Reference Attribute</li></ul>																											
10	What is ODMG? The Object Database Management Group (ODMG) is made up of the leading Object Database vendors plus a large number of companies that are interested in an ODBMS standard. The ODMG has produced a standard for object databases. The ODMG Standard is an interoperability standard which allows applications written to the standard to run on any compliant system																											
11	List out the standard in ODMG. <ul style="list-style-type: none"><li>• Object Model</li><li>• Object Definition Language (ODL)</li><li>• Object Query Language (OQL)</li><li>• C++ Language Binding</li><li>• Java Language Binding</li></ul>																											
12	What is XML Database? XML Database is used to store huge amount of information in the XML format. As the use of XML is increasing in every field, it is required to have a secured place to store the XML documents. The data stored in the database can be queried using XQuery, serialized, and exported into a desired format.																											
13	Difference between HTML and XML.  <table border="1"> <thead> <tr> <th>S.No.</th> <th>HTML</th> <th>XML</th> </tr> </thead> <tbody> <tr> <td>1)</td> <td>HTML is used to display data and focuses on how data looks.</td> <td>XML is a software and hardware independent tool used to transport and store data. It focuses on what data is.</td> </tr> <tr> <td>2)</td> <td>HTML is a markup language itself.</td> <td>XML provides a framework to define markup languages.</td> </tr> <tr> <td>3)</td> <td>HTML is not case sensitive.</td> <td>XML is case sensitive.</td> </tr> <tr> <td>4)</td> <td>HTML is a presentation language.</td> <td>XML is neither a presentation language nor a programming language.</td> </tr> <tr> <td>5)</td> <td>HTML has its own predefined tags.</td> <td>You can define tags according to your need.</td> </tr> <tr> <td>6)</td> <td>In HTML, it is not necessary to use a closing tag.</td> <td>XML makes it mandatory to use a closing tag.</td> </tr> <tr> <td>7)</td> <td>HTML is static because it is used to display data.</td> <td>XML is dynamic because it is used to transport data.</td> </tr> <tr> <td>8)</td> <td>HTML does not preserve whitespaces.</td> <td>XML preserves whitespaces.</td> </tr> </tbody> </table>	S.No.	HTML	XML	1)	HTML is used to display data and focuses on how data looks.	XML is a software and hardware independent tool used to transport and store data. It focuses on what data is.	2)	HTML is a markup language itself.	XML provides a framework to define markup languages.	3)	HTML is not case sensitive.	XML is case sensitive.	4)	HTML is a presentation language.	XML is neither a presentation language nor a programming language.	5)	HTML has its own predefined tags.	You can define tags according to your need.	6)	In HTML, it is not necessary to use a closing tag.	XML makes it mandatory to use a closing tag.	7)	HTML is static because it is used to display data.	XML is dynamic because it is used to transport data.	8)	HTML does not preserve whitespaces.	XML preserves whitespaces.
S.No.	HTML	XML																										
1)	HTML is used to display data and focuses on how data looks.	XML is a software and hardware independent tool used to transport and store data. It focuses on what data is.																										
2)	HTML is a markup language itself.	XML provides a framework to define markup languages.																										
3)	HTML is not case sensitive.	XML is case sensitive.																										
4)	HTML is a presentation language.	XML is neither a presentation language nor a programming language.																										
5)	HTML has its own predefined tags.	You can define tags according to your need.																										
6)	In HTML, it is not necessary to use a closing tag.	XML makes it mandatory to use a closing tag.																										
7)	HTML is static because it is used to display data.	XML is dynamic because it is used to transport data.																										
8)	HTML does not preserve whitespaces.	XML preserves whitespaces.																										
14	Define DTD.  DTD stands for Document Type Definition. It defines the legal building blocks of an XML document. It is used to define document structure with a list of legal elements and attributes.																											

15	What is Information Retrieval (IR)? <ul style="list-style-type: none"> <li>Information retrieval (IR) systems use a simpler data model than database systems</li> <li>Information organized as a collection of documents</li> <li>Documents are unstructured, no schema</li> <li>Information retrieval locates relevant documents, on the basis of user input such as keywords or example documents</li> </ul>
16	What is Web Crawling? Crawling is done by multiple processes on multiple machines, running in parallel .Set of links to be crawled stored in a database.New links found in crawled pages added to this set, to be crawled later Indexing process also runs on multiple machinesCreates a new copy of index instead of modifying old indexOld index is used to answer queries .After a crawl is “completed” new index becomes “old” index
17	Define Simple attributes.  Attributes can be of primitive data type such as, integer, string, real etc. which can take literal value. <i>Example:</i> 'ID' is simple attribute and value is 07.
18	List out the features of OODB. <ul style="list-style-type: none"> <li>Complexity</li> <li>Inheritance</li> <li>Encapsulation</li> <li>Persistency</li> </ul>
19	What is XQuery?  XQuery is a standardized language for combining documents, databases, Web pages and almost anything else. It is very widely implemented. It is powerful and easy to learn. XQuery is replacing proprietary middleware languages and Web Application development languages.
20	What is XML schema?  XML schema is a language which is used for expressing constraint about XML documents. There are so many schema languages which are used now a days for example Relax- NG and XSD (XML schema definition).An XML schema is used to define the structure of an XML document. It is like DTD but provides more control on XML structure.
<b>PART-B</b>	
1	Discuss in detail about the various types of distributed database with suitable examples.
2	Explain about Object Oriented Databases with suitable example.
3	Describe brief notes on ODMG.
4	Discuss in detail about XML Database with relevant examples.
5	Explain the DTD with suitable example.
6	Describe briefly on Information Retrieval (IR) with suitable example.