UNIT – I

Database System Applications:

Database: A database is a collection of related data which represents some aspect of the real world. A database system is designed to be built and populated with data for a certain task.

Or

It is a collection of interrelated data.
These can be stored in the form of tables.
A database can be of any size and varying complexity.
A database may be generated and manipulated manually or it may be computerized. Example:
Customer database consists the fields as cname, cno, and ccity

Cname	Cno	Ccity

Let us see a simple example of a university database. This database is maintaining information concerning students, courses, and grades in a university environment. The database is organized as five files:

The STUDENT file stores data of each student
The COURSE file stores contain data on each course.
The SECTION stores the information about sections in a particular course.
The GRADE file stores the grades which students receive in the various sections
The TUTOR file contains information about each professor.

Data: Facts, figures, statistics etc. having no particular meaning (e.g. 1, ABC, 19 etc).
Record: Collection of related data items, e.g. in the above example the three data items had no meaning. But if we organize them in the following way, then they collectively represent meaningful information.

Roll	**Name**	**Age**
1	ABC	19

Table or **Relation**: Collection of related records.

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

The columns of this relation are called **Fields**, **Attributes** or **Domains**. The rows are called **Tuples** or **Records**.

Database System: It is computerized system, whose overall purpose is to maintain the information and to make that the information is available on demand.

Advantages:

1. Redundency can be reduced.
2. Inconsistency can be avoided.
3. Data can be shared.
4. Standards can be enforced.
5. Security restrictions can be applied.
6. Integrity can be maintained.
7. Data gathering can be possible.
8. Requirements can be balanced.

Database Management System: The software which is used to manage database is called Database Management System (DBMS). For Example, MySQL, Oracle etc. are popular commercial DBMS used in different applications. DBMS allows users the following tasks:

Data Definition: It helps in creation, modification and removal of definitions that define the organization of data in database.

Data Updation: It helps in insertion, modification and deletion of the actual data in the database.

Data Retrieval: It helps in retrieval of data from the database which can be used by applications for various purposes.

User Administration: It helps in registering and monitoring users, enforcing data security, monitoring performance, maintaining data integrity, dealing with concurrency control and recovering information corrupted by unexpected failure.

Database Management System (DBMS) and Its Applications:

A Database management system is a computerized record-keeping system. It is a repository or a container for collection of computerized data files. The overall purpose of DBMS is to allow he users to define, store, retrieve and update the information contained in the database on demand. Information can be anything that is of significance to an individual or organization.

Databases touch all aspects of our lives. Some of the major areas of application are as follows:

1. Banking

2. Airlines

3. Universities

4. Manufacturing and selling

5. Human resources

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 inwarehouses 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.

◦ 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.

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

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

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

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- Add new students, instructors, and courses
- Register students for courses and generate class rosters
- Assign grades to students, compute grade point averages (GPA), and generate transcripts

This 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. Before database management systems (DBMSs) were introduced, organizations usually stored information in such systems. Keeping organizational information in a file-processing system has a number of major disadvantages:

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.

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.

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.

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.

Atomicity problems. A computer system, like any other device, 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.

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. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department *A* may contain either $9500 or

$9900, rather than the correct value of $9400. 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.

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, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file- processing systems.

Advantages of DBMS:

Controlling of Redundancy: Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

Improved Data Sharing : DBMS allows a user to share the data in any number of application programs.

Data Integrity : Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in customer database we can can enforce an integrity that it must accept the customer only from Noida and Meerut city.

Security : Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to be carried out whenever access to sensitive data is attempted.

Data Consistency : By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.

Efficient Data Access : In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing

Enforcements of Standards : With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.

Data Independence : Ina database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta data obtained by the DBMS is changed but the DBMS is continues to

provide the data to application program in the previously used way. The DBMs handles the task of transformation of data wherever necessary.

Reduced Application Development and Maintenance Time : DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application.

Disadvantages of DBMS

1) It is bit complex. Since it supports multiple functionality to give the user the best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software to get the most out of it.

2) Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.

3) DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.

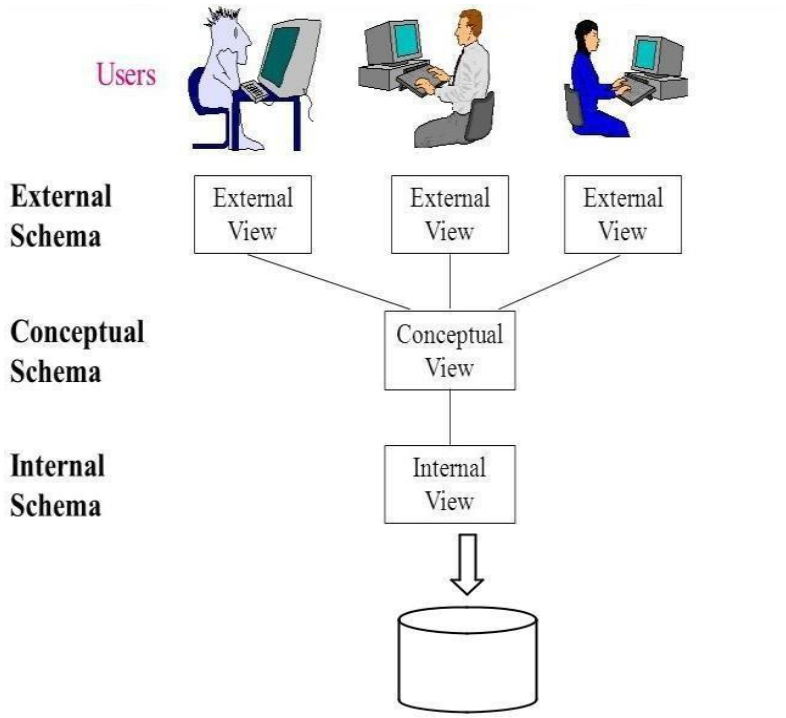
4) DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

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.

Data Abstraction

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:



Levels of Abstraction in a DBMS

• Physical level (or Internal View / Schema): 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 (or Conceptual View / Schema): 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.

• View level (or External View / Schema): 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.

For example, we may describe a record as follows:

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

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive storage locations.

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.

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.

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. The concept of database schemas and instances can be understood by analogy to a program written in a programming language.

Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while 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**, which 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. 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.

Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

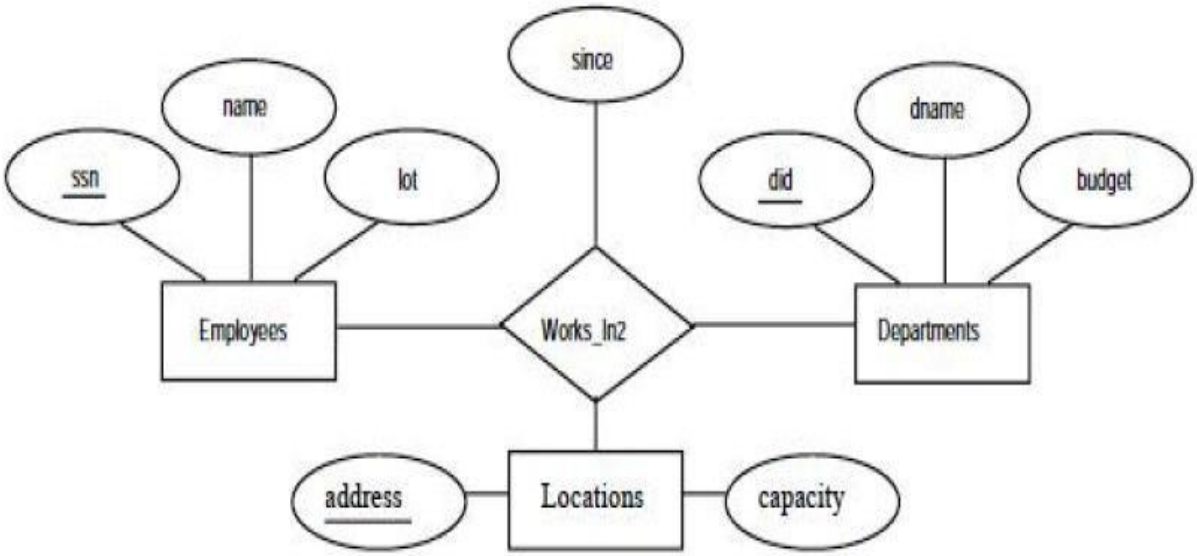
The data models can be classified into four different categories:

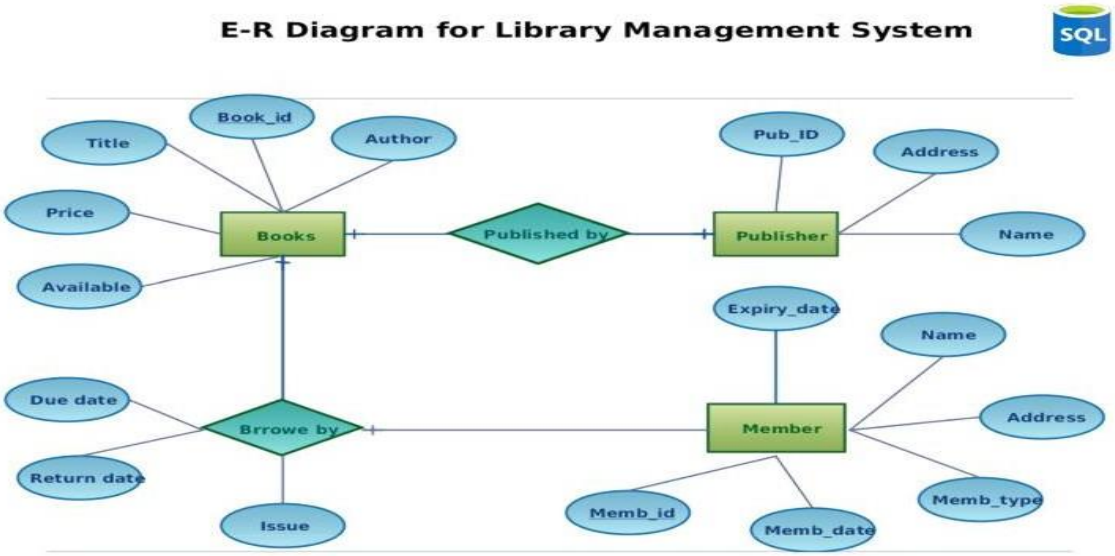
- **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model.

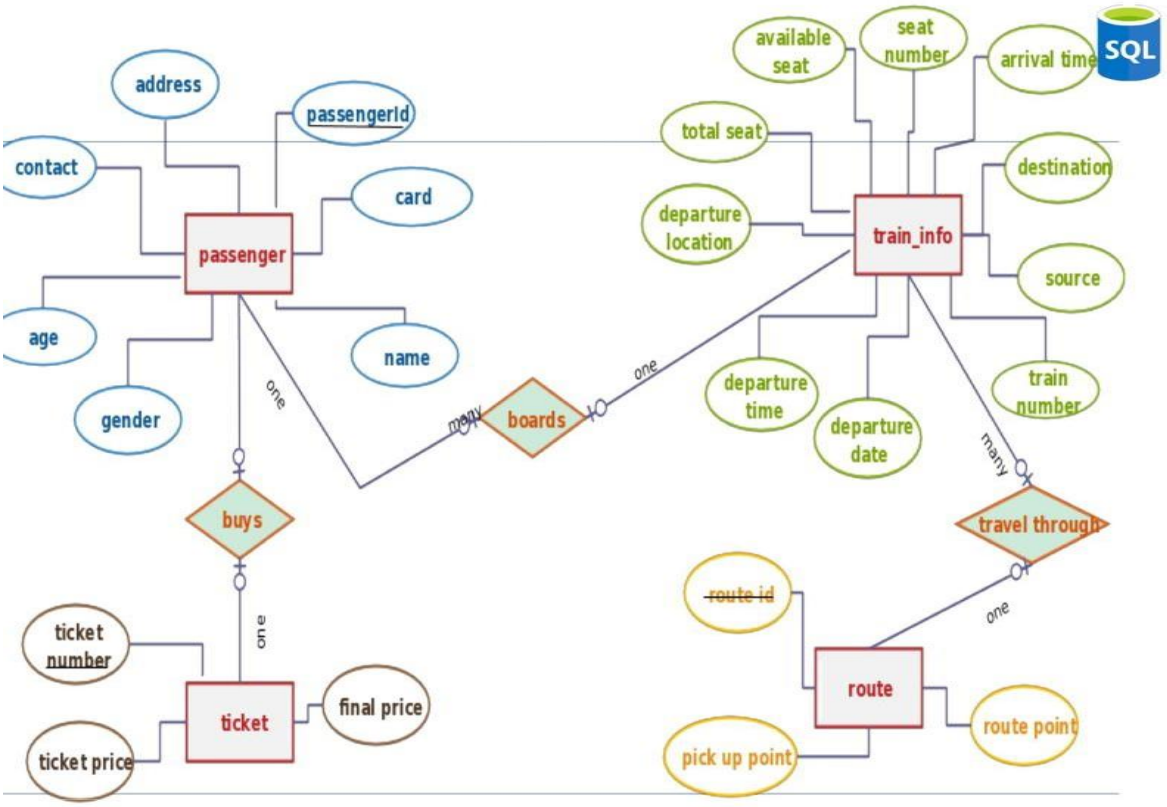
Entity-Relationship Model. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects.

Suppose that each department has offices in several locations and we want to record the locations at which each employee works. The ER diagram for this variant of Works In, which we call Works In2

Example - ternary







E R Model -(Railway Booking System)

E R Model -(Banking Transaction System)

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.

Semi-structured Data Model. The semi-structured 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 semi- structured data.

Historically, 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.

Database Languages

A database system provides a **data-definition language** to specify the database

schema and a **data-manipulation language** to express database queries and updates. In practice, the data-definition and data-manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

Data-Manipulation Language

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion of information from the database
- Modification of information stored in the database

There are basically two types:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.

- **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

Data-Definition Language (DDL)

We specify a database schema by a set of definitions expressed by a special language called a **data- definition language** (**DDL**). The DDL is also used to specify additional properties of the data.

- **Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

- **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept name* value in a *course* record must appear in the *dept name* attribute of some record of the *department* relation.

- **Assertions**. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, ―Every department must have at least five courses offered every semester‖ must be expressed as an assertion.

- **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**,which contains **metadata**—that is, data about data.

Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such ―data about data‖ were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company‘s entire data set, and is a crucial element in the data administration function.

For example, the data dictionary typically stores descriptions of all:

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, datatypes, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.
- Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the

DBA is and so on.

- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database.

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.

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 bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*.

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 without writing a program.

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Online analytical processing (OLAP) tools simplify analysts‘ tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region).

Database Architecture:

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.

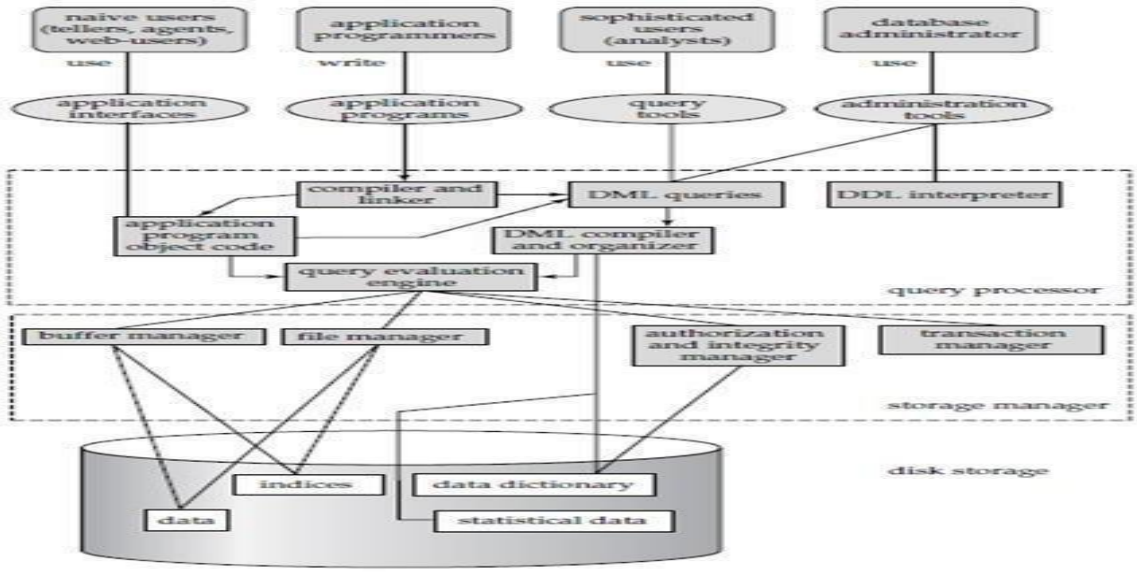


Figure 1.3: Database System Architecture

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. The query processor is important because it helps the database system simplify and facilitate access to data.

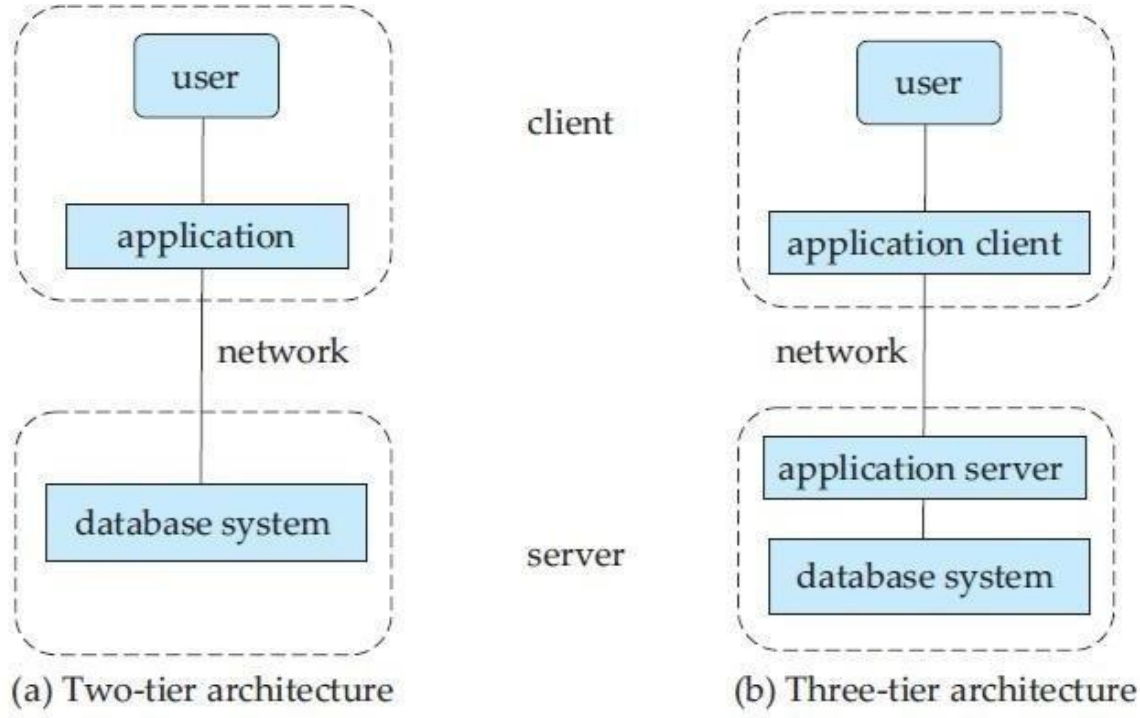


Figure 1.4: Two-tier and three-tier architectures.

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.

Storage Manager:

A *storage manager* is a program module that provides the interface between the lowlevel 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 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 (correct) 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.

Transaction Manager:

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints.

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, consider the *instructor* table of Figure:1.5, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor‘s *ID*, *name*, *dept name*, and *salary*.

Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which 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.

Keys

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*. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

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. 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.

Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure 1.12 shows the schema diagram for our university organization. Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram

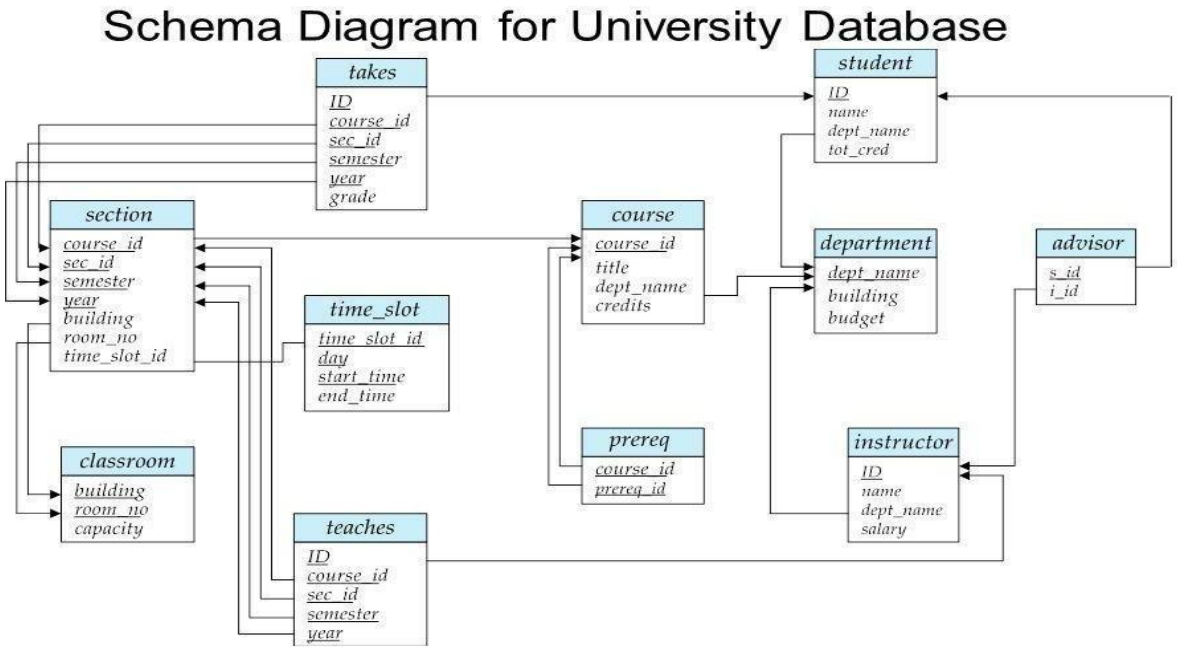


Figure 1.12 shows the schema diagram for our university organization

Conceptual Database Design - Entity Relationship(ER) Modeling:

Database Design Techniques

1. ER Modeling (Top down Approach)
2. Normalization (Bottom Up approach)

What is ER Modeling?

A graphical technique for understanding and organizing the data independent of the actual database implementation

We need to be familiar with the following terms to go further.

Entity

Anything that has an independent existence and about which we collect data. It is also known as entity type.

In ER modeling, notation for entity is given below.

Entity instance

Entity instance is a particular member of the entity type. Example for entity instance : A particular employee **Regular Entity**

An entity which has its own key attribute is a regular entity. Example for regular entity : Employee.

Weak entity

An entity which depends on other entity for its existence and doesn't have any key attribute of its own is a weak entity.

Example for a weak entity : In a parent/child relationship, a parent is considered as a strong entity and the child is a weak entity.

In ER modeling, notation for weak entity is given below.

Attributes

Properties/characteristics which describe entities are called attributes. In ER modeling, notation for attribute is given below.

Domain of Attributes

The set of possible values that an attribute can take is called the domain of the attribute. For example, the attribute day may take any value from the set {Monday, Tuesday ... Friday}. Hence this set can be termed as the domain of the attribute day.

Key attribute

The attribute (or combination of attributes) which is unique for every entity instance is called key attribute.

E.g the employee_id of an employee, pan_card_number of a person etc.If the key attribute consists of two or more attributes in combination, it is called a composite key.

In ER modeling, notation for key attribute is given below.

Simple attribute

If an attribute cannot be divided into simpler components, it is a simple attribute. Example for simple attribute : employee_id of an employee.

Composite attribute

If an attribute can be split into components, it is called a composite attribute.

Example for composite attribute : Name of the employee which can be split into First_name, Middle_name, and Last_name.

Single valued Attributes

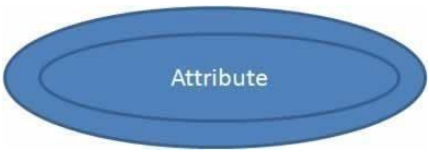
If an attribute can take only a single value for each entity instance, it is a single valued attribute. example for single valued attribute : age of a student. It can take only one value for a particular student.

Multi-valued Attributes

If an attribute can take more than one value for each entity instance, it is a multi-valued attribute. Multi- valued

example for multi valued attribute : telephone number of an employee, a particular employee may have multiple telephone numbers.

In ER modeling, notation for multi-valued attribute is given below.



Stored Attribute

An attribute which need to be stored permanently is a stored attribute Example for stored attribute : name of a student

Derived Attribute

An attribute which can be calculated or derived based on other attributes is a derived attribute.

Example for derived attribute : age of employee which can be calculated from date of birth and current date.

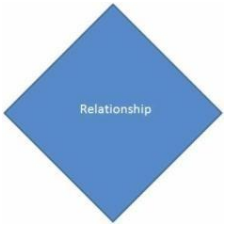
In ER modeling, notation for derived attribute is given below.

Relationships

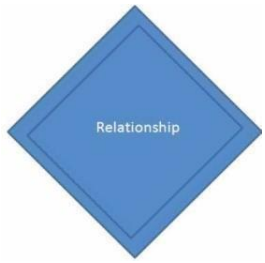
Associations between entities are called relationships

Example : An employee works for an organization. Here "works for" is a relation between the entities employee and organization.

In ER modeling, notation for relationship is given below.



However in ER Modeling, To connect a weak Entity with others, you should use a weak relationship notation as given below



Degree of a Relationship

Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary ,where the degree is 1, 2, and 3, respectively.

Example for unary relationship : An employee ia a manager of another employee Example for binary relationship : An employee works-for department. Example for ternary relationship : customer purchase item from a shop keeper **Cardinality of a Relationship**

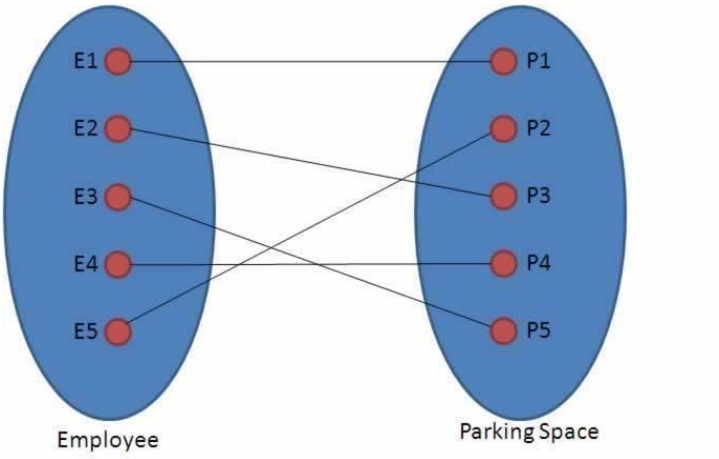
Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivities as given below.

1. One to one (1:1) relationship
2. One to many (1:N) relationship
3. Many to one (M:1) relationship
4. Many to many (M:N) relationship

The minimum and maximum values of this connectivity is called the cardinality of the relationship

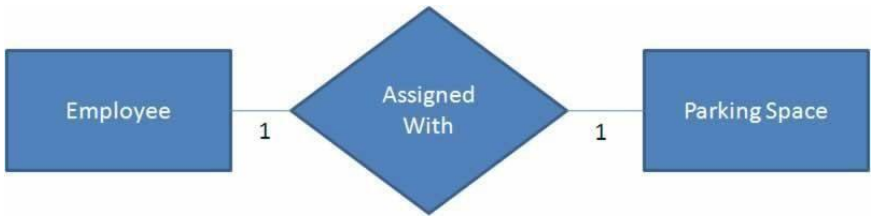
Example for Cardinality – One-to-One (1:1)

Employee is assigned with a parking space.



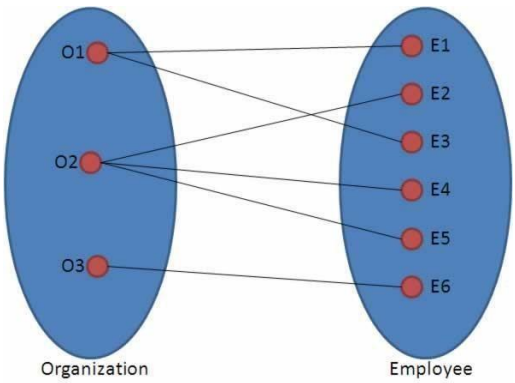
One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)

In ER modeling, this can be mentioned using notations as given below



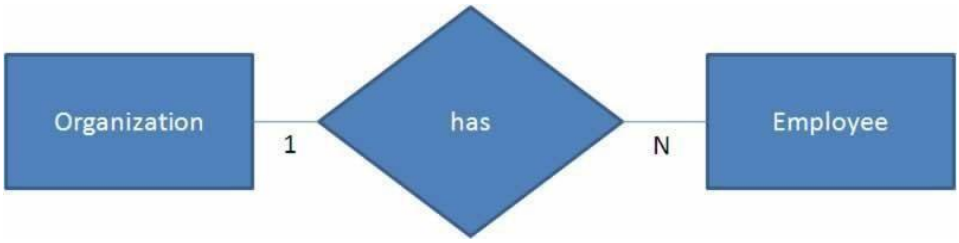
Example for Cardinality – One-to-Many (1:N)

Organization has employees



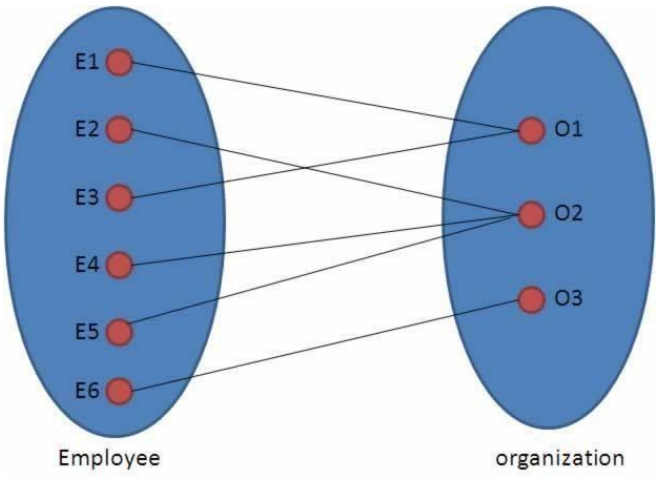
One organization can have many employees , but one employee works in only one organization. Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)

In ER modeling, this can be mentioned using notations as given below



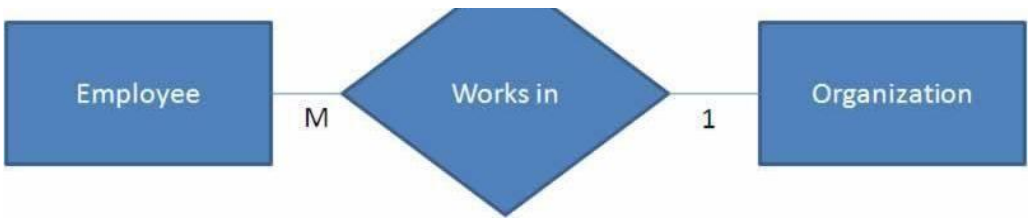
Example for Cardinality – Many-to-One (M :1)

It is the reverse of the One to Many relationship. employee works in organization



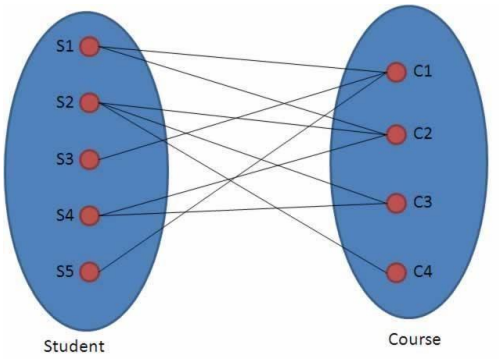
One employee works in only one organization But one organization can have many employees. Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

In ER modeling, this can be mentioned using notations as given below.

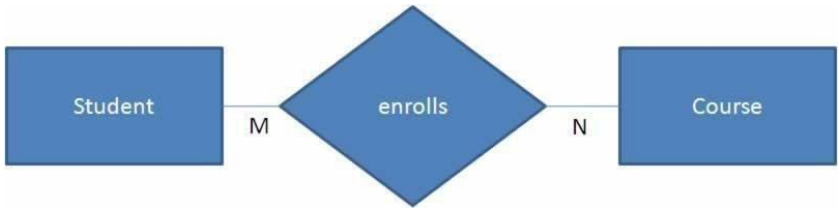


Cardinality – Many-to-Many (M:N)

Students enrolls for courses



One student can enroll for many courses and one course can be enrolled by many students.
Hence it is a M:N relationship and cardinality is Many-to-Many (M:N)
In ER modeling, this can be mentioned using notations as given below



Relationship Participation

1. Total

In total participation, every entity instance will be connected through the relationship to another instance of the other participating entity types

2. Partial

Example for relationship participation

Consider the relationship - Employee is head of the department.

Here all employees will not be the head of the department. Only one employee will be the head of the department. In other words, only few instances of employee entity participate in the above relationship. So employee entity's participation is partial in the said relationship.

Advantages and Disadvantages of ER Modeling (Merits and Demerits of ER Modeling)

Advantages

1. ER Modeling is simple and easily understandable. It is represented in business users language and it can be understood by non-technical specialist.
2. Intuitive and helps in Physical Database creation.
3. Can be generalized and specialized based on needs.
4. Can help in database design.

5. Gives a higher level description of the system.

Disadvantages

1. Physical design derived from E-R Model may have some amount of ambiguities or inconsistency.
2. Sometime diagrams may lead to misinterpretations.

UNIT-2
Relational Algebra and Calculus

PRELIMINARIES

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances, and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances.

We present a number of sample queries using the following schema:

Sailors (*sid:* integer, *sname:* string, *rating:* integer, *age:* real) Boats (*bid:* integer, *bname:* string, *color:* string)

Reserves (*sid:* integer, *bid:* integer, *day:* date)

The key fields are underlined, and the domain of each field is listed after the field name. Thus *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations will be referred to by name, or positionally, using the order in which they are listed above.

RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the re- lational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) rela-tion instances as arguments and returns a relation instance as the result.

Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query.

Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation. Con - sider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression,

$\sigma_{rating>8}(S2)$

The selection operator σ specifies the tuples to retain through a *selection condition*. In general, the selection condition is a boolean combination (i.e., an expression using the logical connectives

A and V) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators <, <=, =, =, >=, or >.

The projection operator π allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using π. The expression $\pi_{sname,rating}(S2)$

Suppose that we wanted to find out only the ages of sailors. The expression

$\pi_{age}(S2)$

a single tuple with *age=35.0* appears in the result of the projection. This follows from

the definition of a relation as a *set* of tuples. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

Set Operations

The following standard operations on sets are also available in relational algebra: *union* (*U*),

intersection (∩), *set-difference* (−), and *cross-product* (×).

- Union: $R \cup S$ returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *union-compatible*, and the schema of the result is defined to be identical to the schema of *R*.
- Intersection: $R \cap S$ returns a relation instance containing all tuples that occur in *both* *R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
- Set-difference: $R - S$ returns a relation instance containing all tuples that occur in *R* but not in *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.
- Cross-product: $R \times S$ returns a relation instance whose schema contains all the fields of *R* (in the same order as they appear in *R*) followed by all the fields of *S*

(in the same order as they appear in *S*). The result of $R \times S$ contains one tuple $\langle r, s \rangle$ (the concatenation of tuples *r* and *s*) for each pair of tuples $r \in R, s \in S$. The cross-product opertion is sometimes called Cartesian product.

sid	*sname*	*rating*	*age*
31	*Lubbe*	*8*	*55.5*
58	*Rusty*	*10*	*35.0*

Figure 4.9 $S1 \cap S2$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 $S1 - S2$

The result of the cross-product $S1 \times R1$ is shown in Figure 4.11 The fields in $S1$

$\times R1$ have the same domains as the corresponding fields in $R1$ and $S1$. In Figure 4.11 *sid* is listed in parentheses to

emphasize that it is not an inherited field name; only the corresponding domain is inherited.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	*Dustin*	*7*	*45.0*	22	*101*	*10/10/96*
22	*Dustin*	*7*	*45.0*	58	*103*	*11/12/96*
31	*Lubber*	*8*	*55.5*	22	*101*	*10/10/96*
31	*Lubber*	*8*	*55.5*	58	*103*	*11/12/96*
58	*Rusty*	*10*	*35.0*	22	*101*	*10/10/96*
58	*Rusty*	*10*	*35.0*	58	*103*	*11/12/96*

Figure 4.11 $S1 \times R1$

Renaming

We introduce a renaming operator ρ for this purpose. The expression $\rho(R(F), E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E, and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the *renaming list F*.

For example, the expression $\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$ returns a relation that contains the tuples shown in Figure 4.11 and has the followi ng schema: C(*sid1:* integer, *sname:* string, *rating:* integer, *age:* real, *sid2:* integer, *bid:* integer,*day:* dates).

Joins

The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products.joins have received a lot of attention, and there are several variants of the join operation.

Condition Joins

The most general version of the join operation accepts a *join condition c* and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

$R \overset{a}{\rhd} c\ S = \sigma c(R \times S)$

Thus $\overset{a}{\rhd}$ is defined to be a cross-product followed by a selection. Note that the condition *c* can (and typically *does*) refer to attributes of both *R* and *S*.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	*Dustin*	*7*	*45.0*	58	*103*	*11/12/96*
31	*Lubber*	*8*	*55.5*	58	*103*	*11/12/96*

Figure 4.12 $S1 \overset{a}{\rhd} S1.sid < R1.sid\ R1$

Equijoin

A common special case of the join operation $R \overset{a}{\rhd} S$ is when the *join condition* con-sists solely of equalities (connected by A) of the form *R.name*1 = *S.name*2, that is, equalities between two

fields in *R* and *S*. In this case, obviously, there is some redun-dancy in retaining both attributes in the result.

Natural Join

A further special case of the join operation $R \ a\triangleright \ S$ is an equijoin in which equalities are specified on *all* fields having the same name in *R* and *S*. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields.

Division

The division operator is useful for expressing certain kinds of queries, for example: –Find the names of sailors who have reserved all boats.‖ Understanding how to use the basic operators of the algebra to define division is a useful exercise.

(Q1) Find the names of sailors who have reserved boat 103.

This query can be written as follows:

$\pi sname((\sigma bid=103 Reserves) \ a\triangleright Sailors)$

We first compute the set of tuples in Reserves with *bid* = 103 and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances *R*2 and *S*3, it yields a relation

(Q2) Find the names of sailors who have reserved a red boat.

$\pi sname((\sigma color='red' Boats) \ a\triangleright \ Reserves \ a\triangleright \ Sailors$

This query involves a series of two joins. First we choose (tuples describing) red boats.

(Q3) Find the colors of boats reserved by Lubber.

$\pi color((\sigma sname='Lubber' Sailors) \ a\triangleright \ Reserves \ a\triangleright \ Boats)$

This query is very similar to the query we used to compute sailors who reserved red boats. On instances *B*1, *R*2, and *S*3, the query will return the colors green and red.

(Q4) Find the names of sailors who have reserved at least one boat.

$\pi sname(Sailors \ a\triangleright \ Reserves)$

(Q5) Find the names of sailors who have reserved a red or a green boat.

$\rho(T \ empboats, (\sigma color='red' Boats) \ U \ (\sigma color='green' Boats))$

$\pi sname(Tempboats \ a\triangleright Reserves \ a\triangleright Sailors)$

(Q6) Find the names of sailors who have reserved a red and a green boat

$\rho(T\ empboats2, (\sigma color='red'\ Boats) \cap (\sigma color='green'\ Boats))$

$\pi sname(Tempboats2 \triangleright Reserves \triangleright Sailors)$

However, this solution is incorrect —it instead tries to compute sailors who have re-served a boat that is both red and green.

$\rho(T\ empred, \pi sid((\sigma color='red'\ Boats) \triangleright Reserves))$

$\rho(T\ empgreen, \pi sid((\sigma color='green'\ Boats) \triangleright Reserves))$

$\pi sname((Tempred \cap Tempgreen) \triangleright Sailors)$

(Q7) Find the names of sailors who have reserved at least two boats.

$\rho(Reservations, \pi sid,sname,bid(Sailors \triangleright Reserves))\ \rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2, 5 \rightarrow sname2,6 \rightarrow bid2),Reservations \times Reservations)\ \pi sname1\sigma(sid1=sid2) \cap (bid1=bid2)Reservationpairs$

(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

$\pi sid(\sigma age>20Sailors) - \pi sid((\sigma color='red'\ Boats) \triangleright Reserves \triangleright Sailors)$

This query illustrates the use of the set-difference operator. Again, we use the fact that *sid* is the key for Sailors.

(Q9) Find the names of sailors who have reserved all boats.

The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

$\rho(T\ empsids, (\pi sid,bidReserves)/(\pi bidBoats))$

$\pi sname(Tempsids \triangleright Sailors)$

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$\rho(T\ empsids, (\pi sid,bidReserves)/(\pi bid(\sigma bname='Interlake'\ Boats)))$

$\pi sname(Tempsids \triangleright Sailors)$

RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed.

Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields.

(Q11) Find all sailors with a rating above 7.

{S I S E *Sailors* ^ *S. rating* > 7*}* with respect to the given database instance, *F* evaluates to (or simply _is‘) true if one of the

following holds:

- *F* is an atomic formula *R* □ *Rel,* and *R* is assigned a tuple in the instance of relation *Rel*.
- *F* is a comparison *R.a* op *S.b*, *R.a* op *constant*, or *constant* op *R.a*, and the tuples assigned to *R* and *S* have field values *R.a* and *S.b* that make the comparison true.
- *F* is of the form ¬*p*, and *p* is not true; or of the form *p* ^ *q*, and both *p* and *q* are true; or of the form *p V q*, and one of them is true, or of the form *p* □ *q* and *q* is true whenever4 *p* is true.
- *F* is of the form □*R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*), including the variable *R*,5 that makes the formula *p*(*R*) true.
- *F* is of the form □*R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*) that makes the formula *p*(*R*) true no matter what tuple is assigned to *R*.

(Q12) Find the names and ages of sailors with a rating above 7 .

{P | □*S* □ *Sailors*(*S.rating* > 7 □ *P.name = S.sname* □ *P.age = S.age*)*}*

This query illustrates a useful convention: *P* is considered to be a tuple variable with exactly two fields, which are called *name* and *age*, because these are the only fields of *P* that are mentioned and *P* does not range over any of the relations in the query; that is, there is no subformula of the form *P* □ *Relname*.

(Q13) Find the sailor name, boat id, and reservation date for each reservation

{P | □*R* □*Reserves* □*S* □ *Sailors*

(*R.sid = S.sid* □ *P.bid = R.bid* □ *P.day = R.day* □ *P.sname = S.sname*)*}*

(Q1) Find the names of sailors who have reserved boat 103.

{P | □*S* □ *Sailors* □*R* □ *Reserves*(*R.sid = S.sid* □ *R.bid* = 103 □*P.sname = S.sname*)*}*

This query can be read as follows: ―Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the *sid* field, and with *bid* = 103.‖

(Q2) Find the names of sailors who have reserved a red boat.

{P | □S □ Sailors □R □ Reserves(R.sid = S.sid □ P.sname = S.sname

□ □B □Boats(B.bid = R.bid □ B.color ='red'))}

This query can be read as follows: –Retrieve all sailor tuples *S* for which there exist tuples *R* in Reserves and *B* in Boats such that *S.sid = R.sid*, *R.bid = B.bid*, and *B.color ='red'*.‖

(Q7) Find the names of sailors who have reserved at least two boats. *{P |*

□S □ Sailors □R1 □ Reserves □R2 □ Reserves (S.sid = R1.sid

□ R1.sid = R2.sid □ R1.bid ≠ R2.bid □ P.sname = S.sname)}

(Q9) Find the names of sailors who have reserved all boats.

{P | □S □ Sailors □B □ Boats

(□R □ Reserves(S.sid = R.sid □R.bid = B.bid □P.sname = S.sname))}

(Q14) Find sailors who have reserved all red boats.

{S | S □ Sailors □ □B □ Boats

(B.color ='red' □(□R ∈ Reserves(S.sid = R.sid □ R.bid = B.bid)))}

Domain Relational Calculus

A domain variable is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers).

A DRC query has the form *{ 〈 x1, x2, . . . , xn 〉 | p(〈x1,x2,.. ., xn〉)}*, where each *xi* is either a *domain variable* or a constant and *p(〈x1,x2,.. ., xn〉)* denotes a DRC formula whose only free variables are thevari-ables among the *xi*, $1 \leq i \leq n$. The result of this query is the set of all tuples *〈x1, x2,.. .,xn〉* for which the formula evaluates to true.

(Q1) Find the names of sailors who have reserved boat 103.

{ 〈N 〉 | □I, T, A(〈I, N, T, A〉 □ Sailors

□□Ir, Br, D(〈Ir, Br, D〉 □ Reserves □ Ir = I □ Br = 103))*}* ***(Q2) Find the names of sailors who have reserved a red boat.***

{ 〈N 〉 | □I, T, A(〈I, N, T, A〉 □ Sailors

□□*I, Br, D*〉 □ *Reserves* □ □ 〈*Br, BN,'red'*〉 □ *Boats*)}

(Q7) Find the names of sailors who have reserved at least two boats.

{ 〈*N* 〉 | □*I, T, A*(〈*I, N, T, A*〉 □ *Sailors* □ □*Br*1, *Br*2, *D*1, *D*2(〈*I, Br*1, *D*1〉

□*Reserves* □ 〈*I, Br*2, *D*2〉 □ *Reserves* □ *Br*1 = *Br*2)

(Q9) Find the names of sailors who have reserved all boats.

{ 〈*N* 〉 | □*I, T, A*(〈*I, N, T, A*〉 □ *Sailors* □

□*B, BN, C*(¬(〈*B, BN, C*〉 □ *Boats*) □

(□ 〈*Ir, Br, D*〉 □ *Reserves*(*I = Ir* ¯ *Br = B*))))}

THE FORM OF A BASIC SQL QUERY

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual evaluation strategy*. A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand, rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

Sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0

Figure 5.1An Instance *S* 3 of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98

Figure 5.2 An Instance *R*2 of Reserves

bid	*bname*	*color*
101	*Interlake*	*blue*
102	*Interlake*	*red*
103	*Clipper*	*green*
104	*Marine*	*red*

(Q15) Find the names and ages of all sailors.

SELECT DISTINCT S.sname, S.age FROM Sailors S

The answer to this query with and without the keyword DISTINCT on instance *S*3 of Sailors is shown in Figures 5.4 and 5.5. The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35.

(Q11) Find all sailors with a rating above 7.

SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

(Q16) Find the sids of sailors who have reserved a red boat.

SELECT R.sid FROM Boats B, Reserves R WHERE B.bid = R.bid AND B.color = ‗red‘

(Q2) Find the names of sailors who have reserved a red boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND

R.bid = B.bid AND B.color = ‗red‘

(Q3) Find the colors of boats reserved by Lubber.

SELECT B.color FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid

AND R.bid = B.bid AND S.sname = ‗Lubber‘

(Q4) Find the names of sailors who have reserved at least one boat.

SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid

Expressions and Strings in the SELECT Command

SQL supports a more general version of the select-list than just a list of columns. Each item in a select-list can be of the form expression AS column name, where expression is any arithmetic or string expression over column names (possibly prefixed by range variables) and constants.

(Q5) Compute increments for the ratings of persons who have sailed two different boats on the same day.

SELECT S.sname, S.rating+1 AS rating FROM Sailors S, Reserves R1, Reserves R2 WHERE S.sid = R1.sid AND S.sid = R2.sid AND R1.day = R2.day AND R1.bid <> R2.bid

Also, each item in a qualification can be as general as expression1 = expression2.

SELECT S1.sname AS name1, S2.sname AS name2 FROM Sailors S1, Sailors S2 WHERE 2*S1.rating = S2.rating-1.

(Q6) Find the ages of sailors whose name begins and ends with B and has at least three characters.

SELECT S.age FROM Sailors S WHERE S.sname LIKE _B %B‘

The only such sailor is Bob, and his age is 63.5.

UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form pre-sented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.4 SQL also provides other set operations: IN (to

check if an element is in a given set),op ANY,op ALL(tocom-pare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning. We cover UNION, INTERSECT, and EXCEPT in this section. Consider the following query:

(Q1) Find the names of sailors who have reserved both a red and a green boat.

SELECT S.sname FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats B2 WHERE S.sid = R1.sid AND R1.bid = B1.bid AND S.sid = R2.sid AND R2.bid

= B2.bid AND B1.color=_red‘ AND B2.color = _green‘

(Q2) Find the sids of all sailors who have reserved red boats but not green boats.

SELECT S.sid FROM Sailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = _red‘ EXCEPT SELECT S2.sid FROM Sailors S2, Reserves R2, Boats B2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = _green‘.

NESTED QUERIES

A nested query is a querythat has another query embedded within it; the embedded query is called a subquery.

(Q1) Find the names of sailors who have reserved boat 103.

SELECT S.sname FROM Sailors S WHERE S.sid IN (SELECT R.sid FROM Reserves R

WHERE R.bid = 103)

(Q2) Find the names of sailors who have reserved a red boat.

SELECT S.sname FROM Sailors S WHERE S.sid IN (SELECT R.sid FROM Reserves R WHERE R.bid IN (SELECT B.bid FROM Boats B WHERE B.color = _red‘)

(Q3) Find the names of sailors who have* not *reserved a red boat.

SELECT S.sname FROM Sailors S WHERE S.sid NOT IN (SELECT R.sid FROM Reserves R WHERE R.bid IN (SELECT B.bid FROM Boats B WHERE B.color = _red‘).

Correlated Nested Queries

In the nested queries that we have seen thus far, the inner subquery has been completely independent of the outer query:

(Q1) Find the names of sailors who have reserved boat number 103.

SELECT S.sname FROM Sailors S WHERE EXISTS (SELECT * FROM Reserves R WHERE R.bid = 103 AND R.sid = S.sid)

Set-Comparison Operators

(Q1) Find sailors whose rating is better than some sailor called Horatio.

SELECT S.sid FROMSailors S WHERE S.rating > ANY (SELECT S2.rating FROM Sailors S2 WHERE S2.sname = _Horatio‘)

(Q2) Find the sailors with the highest rating .

SELECT S.sid FROM Sailors S WHERE S.rating >= ALL (SELECT S2.rating FROM Sailors S2)

More Examples of Nested Queries

(Q1) Find the names of sailors who have reserved both a red and a green boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = _red‘ AND S.sid IN (SELECT S2.sid FROM Sailors S2, Boats B2, Reserves R2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = _green‘)

Q9) Find the names of sailors who have reserved all boats.

SELECT S.sname FROM Sailors S WHERE NOT EXISTS ((SELECT B.bid FROM Boats B) EXCEPT (SELECT R.bid FROM Reserves R WHERE R.sid = S.sid))

AGGREGATE OPERATORS

We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM.

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

(Q1) Find the average age of all sailors.

SELECT AVG (S.age) FROM Sailors S

(Q2) Find the average age of sailors with a rating of 10.

SELECT AVG (S.age) FROM Sailors S WHERE S.rating = 10

SELECT S.sname, MAX (S.age) FROMSailors S

Q3) Count the number of sailors.

SELECT COUNT (*) FROM Sailors S

The GROUP BY and HAVING Clauses

we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance). ***(Q31) Find the age of the youngest sailor for each rating level.***

SELECT MIN (S.age)

FROM Sailors S

WHERE S.rating = *i*

Q32) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

SELECT S.rating, MIN (S.age) AS minageGROUP BY S.rating

HAVING COUNT (*) > 1

More Examples of Aggregate Queries

Q3) For each red boat, find the number of reservations for this boat.

SELECT B.bid, COUNT (*) AS sailorcount FROM Boats B, Reserves R

WHERE R.bid = B.bid AND B.color = ‗red‘ GROUP BY B.bid

SELECT B.bid, COUNT (*) AS sailorcount FROM Boats B, Reserves R

WHERE R.bid = B.bid GROUP BY B.bid HAVING B.color = ‗red‘

(Q4) Find the average age of sailors for each rating level that has at least two sailors.

SELECT S.rating, AVG (S.age) AS avgage

FROM Sailors S

GROUP BY S.rating

HAVING COUNT (*) > 1

(Q5) Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.

SELECT S.rating, AVG (S.age) AS avgage

FROM Sailors S

WHERE S. age >= 18

GROUP BY S.rating

HAVING 1 < (SELECT COUNT (*)FROM Sailors S2 WHERE S.rating = S2.rating

(Q6) Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two* such *sailors.

```
SELECT      S.rating, AVG ( S.age ) AS avgage
FROM Sailors S
WHERE S. age > 18
GROUP BY S.rating

HAVING 1 < ( SELECT COUNT (*)
FROM Sailors S2
WHERE S.rating = S2.rating AND S2.age >= 18 )
```

The above formulation of the query reflects the fact that it is a variant of Q35. The answer to Q36 on instance *S*3 is shown in Figure 5.16. It differs from the answer to Q35 in that there is no tuple for rating 10, since there is only one tuple with rating 10 and *age* ≥ 18.

```
SELECT      S.rating, AVG ( S.age ) AS avgage
FROM Sailors S
WHERE S. age > 18
GROUP BY S.rating
HAVING      COUNT (*) > 1
```

This formulation of Q36 takes advantage of the fact that the WHERE clause is applied before grouping is done; thus, only sailors with *age* > 18 are left when grouping is done. It is instructive to consider yet another way of writing this query:

```
SELECT Temp.rating, Temp.avgage
FROM ( SELECT S.rating, AVG ( S.age ) AS
avgage, COUNT (*) AS
ratingcount
FROM Sailors S WHERE S. age > 18 GROUP BY S.rating ) AS Temp
WHERE Temp.ratingcount > 1
```

NULL VALUES

we have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a *rating* column, what row should we insert for Dan? What is needed here is a special value that denotes *unknown*.

SQL provides a special column value called *null* to use in such situations. We use *null* when the column value is either *unknown* or *inapplicable*. Using our Sailor table definition, we might enter the row 〈 98, *Dan, null,* 39 〉 to represent Dan. The presence of *null* values complicates many issues, and we consider the impact of *null* values on SQL in this section.

Comparisons Using Null Values

Consider a comparison such as *rating = 8*. If this is applied to the row for Dan, is this condition true or false? Since Dan's rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is *null*; for example, we can say *rating* IS NULL, which would evaluate to true on the row representing Dan. We can also say *rating* IS NOT NULL, which would evaluate to false on the row for Dan.

Logical Connectives AND, OR, and NOT

Now, what about boolean expressions such as *rating* = 8 OR *age* < 40 and *rating* = 8 AND *age* < 40? Considering the row for Dan again, because *age* < 40, the first expression evaluates to true regardless of the value of *rating*, but what about the second? We can only say unknown.

INTRODUCTION TO VIEWS

A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view de nition. Consider the Students and Enrolled relations.

CREATE VIEW B-Students (name, sid, course)

AS SELECT S.sname, S.sid, E.cid

FROM Students S, Enrolled E

WHERE S.sid = E.sid AND E.grade = `B'

This view can be used just like a base table, or explicitly stored table, in de ning new queries or views.

DESTROYING/ALTERING TABLES AND VIEWS

If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows *and* remove the table de nition information), we can use the DROP TABLE command. For example, DROP TABLE Students RESTRICT destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword RESTRICT is replaced by CASCADE, Students is dropped and any ref-erencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be speci ed. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

ALTER TABLE modi es the structure of an existing table. To add a column called *maiden-name*

to Students, for example, we would use the following command:

ALTER TABLE Students

ADD COLUMN maiden-name CHAR(10)

TRIGGERS

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

Event: A change to the database that activates the trigger. Condition: A query or test that is run when the trigger is activated.

Action: A procedure that is executed when the trigger is activated and its con-dition is true.

A trigger *action* can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute new queries, and make changes to the database.

Examples of Triggers in SQL

The examples shown in Figure 5.19, written using Oracle 7 Server syntax for defining triggers, illustrate the basic concepts behind triggers. (The SQL:1999 syntax for these triggers is similar; we will see an example using SQL:1999 syntax shortly.) The trigger called *init count* initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called *incr count* increments the counter for each inserted tuple that satisfies the condition *age* < 18.

CREATE TRIGGER init count BEFORE INSERT ON Students /* Event */

DECLARE -

count INTEGER;

```
BEGIN
count := 0;
END
/* Action */
CREATE TRIGGER incr count AFTER INSERT ON Students /* Event */ WHEN (new.age < 18)  /* Condition; _new‘ is just-inserted tuple */ FOR EACH ROW
BEGIN /* Action; a procedure in Oracle‘s PL/SQL syntax */ count := count + 1;
END
```

(identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with *age* < 18. (The trigger in Figure 5.19 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

```
CREATE TRIGGER set count AFTER INSERT ON Students /*  Event */
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
INSERT  /* Action */
INTO StatisticsTable(ModifiedTable,  ModificationType,  Count) SELECT
_Students‘, _Insert‘, COUNT * FROM InsertedTuples I WHERE I.age < 18
```

Procedures:

"A procedures or function is a group or set of SQL and PL/SQL statements that perform a specific task."

A function and procedure is a named PL/SQL Block which is similar . The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

A procudure is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages. A procedure has a header and a body.

The header consists of the name of the procedure and the parameters or variables passed to the procedure.

The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

We can pass parameters to procedures in three ways :

Parameters Description

IN type These types of parameters are used to send values to stored procedures.

OUT type These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.

IN OUT type These types of parameters are used to send values and get values from stored procedures.

A procedure may or may not return any value.

Syntax:

CREATE [OR REPLACE] PROCEDURE procedure_name (<Argument> {IN, OUT, IN OUT} <Datatype>,…)

IS

Declaration section<variable, constant> ;

BEGIN

Execution section

EXCEPTION

Exception section

END

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

How to execute a Procedure?

There are two ways to execute a procedure :

From the SQL prompt : EXECUTE [or EXEC] procedure_name;

Within another procedure – simply use the procedure name : procedure_name;

Example:

create table named emp have two column id and salary with number datatype.

```
CREATE OR REPLACE PROCEDURE p1(id IN NUMBER, sal IN NUMBER)
AS
BEGIN
  INSERT INTO emp VALUES(id, sal);
DBMD_OUTPUT.PUT_LINE('VALUE INSERTED.');
END;
/
```

UNIT-III

NORMALIZATION

SCHEMA REFINEMENT

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

Problems Caused by Redundancy

Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

- **Redundant storage:** Some information is stored repeatedly.
- **Update anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

Consider a relation obtained by translating a variant of the Hourly Emps entity set from Chapter 2:

Hourly Emps(*ssn, name, lot, rating, hourly wages, hours worked*)

If we delete all tuples with a given rating value (e.g., we delete the tuples for Smethurst and Guldu) we lose the association between that *rating* value and its *hourly wage* value (a *deletion anomaly*).

Ideally, we want schemas that do not permit redundancy, but at the very least we want to be able to identify schemas that do allow redundancy. Even if we choose to accept a schema with some of these drawbacks, perhaps owing to performance considerations, we want to make an informed decision.

Use of Decompositions

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies (and, for that matter, other

ICs) can be used to identify such situations and to suggest refinements to the schema.

We can deal with the redundancy in Hourly Emps by decomposing it into two relations:

Hourly Emps2(*ssn*, *name*, *lot*, *rating*, *hours worked*) Wages(*rating*, *hourly wages*)

Unless we are careful, decomposing a relation schema can create

more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation).

FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a kind of IC that generalizes the concept of a *key*. Let *R* be a relation schema and let *X* and *Y* be nonempty sets of attributes in *R*. We

say that an instance *r* of *R* satisfies the FD *X ! Y* 1 if the following holds for every pair of tuples *t*1 and *t*2 in *r*:

If *t*1:*X* = *t*2:*X*, then *t*1:*Y* = *t*2:*Y* .

A primary key constraint is a special case of an FD. The attributes in the key play the role of *X*, and the set of all attributes in the relation plays the role of *Y*. Note, however, that the definition of an FD does not require that the set *X* be minimal; the additional minimality condition must be met for *X* to be a key. If *X ! Y* holds, where

Y is the set of all attributes, and there is some subset *V* of *X* such that *V ! Y* holds, then *X* is a *super key*; if *V* is a strict subset of *X*, then *X* is not a key.

In the rest of this chapter, we will see several examples of FDs that are not key constraints.

REASONING ABOUT FUNCTIONAL DEPENDENCIES

The discussion up to this point has highlighted the need for techniques that allow us to carefully examine and further re ne relations obtained through ER design (or, for that matter, through other approaches to conceptual design.

Given a set of FDs over a relation schema *R*, there are typically several additional

FDs that hold over *R* whenever all of the given FDs hold. As an example, consider:

Workers(*ssn*, *name*, *lot*, *did*, *since*)

15.4.1 Closure of a Set of FDs

The set of all FDs implied by a given set *F* of FDs is called the closure of F and is

denoted as *F* +. An important question is how we can infer, or compute, the closure of a given set *F* of FDs. The answer is simple and elegant. The following three rules, called Armstrong's Axioms, can be applied repeatedly to infer all FDs implied by a set *F* of FDs. We use *X*, *Y*, and *Z* to denote *sets* of attributes over a relation schema

R:

Reflexivity: If *X Y*, then *X ! Y*.

Augmentation: If *X ! Y*, then *XZ ! YZ* for any *Z*. Transitivity: If *X ! Y* and *Y ! Z*, then *X ! Z*.

Armstrong's Axioms are sound in that they generate only FDs in *F* + when applied to a set *F* of FDs. They are complete in that repeated application of these rules will

generate all FDs in the closure *F* +. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about *F*+:

Union: If *X ! Y* and *X ! Z*, then *X ! YZ*. Decomposition: If *X ! YZ*, then *X ! Y* and *X ! Z*.

These additional rules are not essential; their soundness can be proved using Arm-strong's Axioms.

use a more elaborate version of the Contracts relation:

Contracts (*contractid*, *supplierid*, *projectid*, *deptid*, *partid*, *qty*, *value*)

The following ICs are known to hold:

1. The contract id *C* is a key: *C ! CSJDPQV*.
2. A project purchases a given part using a single contract: JP ! C.
3. A department purchases at most one part from a supplier: SD ! P.

NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several normal forms have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

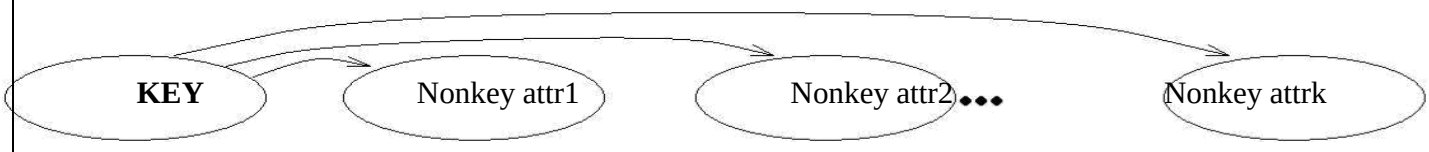
The normal forms based on FDs are first normal form (1NF), second normal form (2NF), third normal form (3NF), and Boyce-Codd normal form (BCNF). These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in first normal form if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our de nition of the relational model. Although some of the newer database systems are relaxing this requirement, in this chapter we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

Boyce-Codd Normal Form

Let R be a relation schema, X be a subset of the attributes of R, and let A be an attribute of R. R is in Boyce-Codd normal form if for every FD X ! A that holds over R, one of the following statements is true:

- *A 2 X*; that is, it is a trivial FD, or
- *X* is a super key.

Note that if we are given a set *F* of FDs, according to this de nition, we must consider each dependency *X ! A* in the closure *F +* to determine whether *R* is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in *F* is a super key (by computing the attribute closure and seeing if it includes all attributes of *R*).



FDs in a BCNF Relation

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy)

Thus, if a relation is in BCNF, every field of every tuple records a piece of information that cannot be inferred (using only FDs) from the values in all other elds in (all tuples of) the relation instance.

Third Normal Form

Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and *A* be an attribute of *R*. *R* is in third normal form if for every FD *X ! A* that holds over *R*, one of the following statements is true:

- *A 2 X*; that is, it is a trivial FD, or
- *X* is a super key, or
- *A* is part of some key for *R*.

The definition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF.

Partial dependencies are illustrated in Figure 15.9, and transitive dependencies are illustrated in Figure. Note that in Figure 15.10, the set *X* of attributes may or may not have some attributes in common with KEY; the diagram should be interpreted as indicating only that *X* is not a subset of KEY.

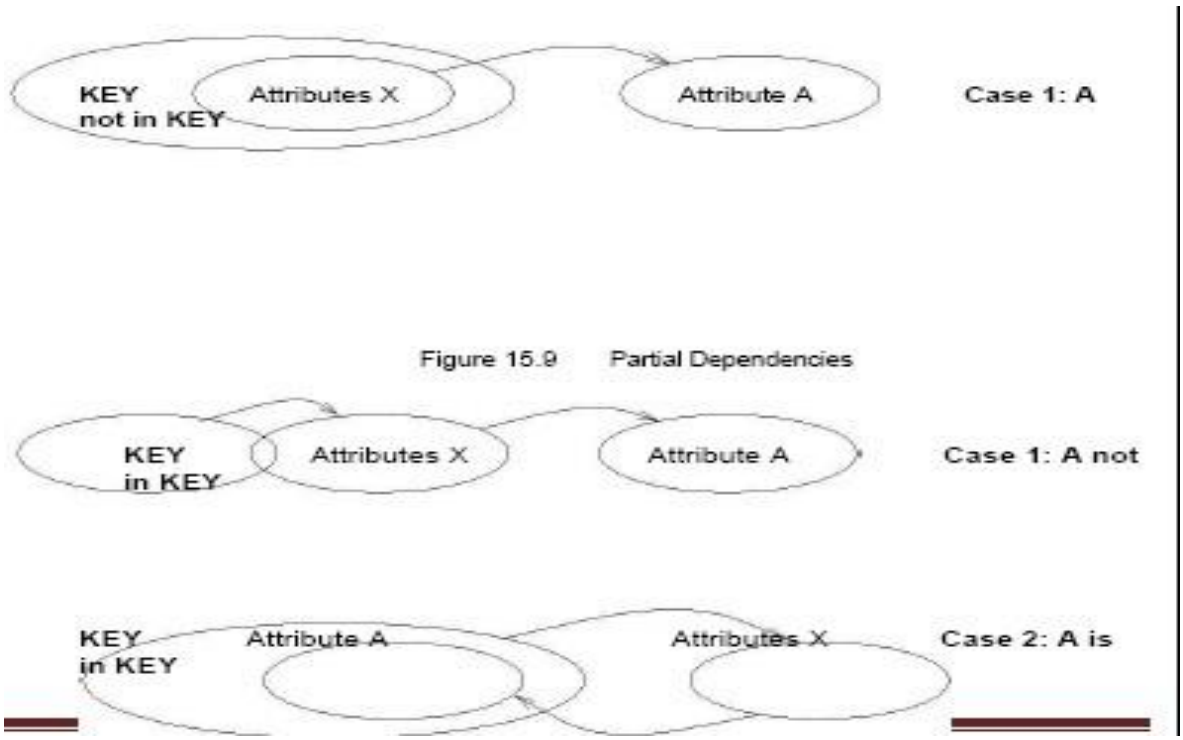


Figure 15.9 Partial Dependencies

Transitive Dependencies

The motivation for 3NF is rather technical. By making an exception for certain de-pendencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only decompositions that have certain desirable properties

DECOMPOSITIONS

As we have seen, a relation in BCNF is free of redundancy (to be precise, redundancy that can be detected using FD information), and a relation schema in 3NF comes close. If a relation schema is not in one of these normal forms, the FDs that cause a violation can give us insight into the potential problems..

A decomposition of a relation schema *R* consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of *R* and together include all attributes in *R*.

Lossless-Join Decomposition

Let *R* be a relation schema and let *F* be a set of FDs over *R*. A decomposition of *R* into two schemas with attribute sets *X* and *Y* is said to be a lossless-join decomposition with respect to F if for every instance *r* of *R* that satis es the

dependencies in *F*, $_X (r) ./ _Y (r) = r$.

All decompositions used to eliminate redundancy must *be lossless.*

The following simple test is very useful:

Let *R* be a relation and *F* be a set of FDs that hold over *R*. The decomposition of *R* into relations with attribute sets R_1 and R_2 is lossless if and only if *F* + contains either the FD R_1

$\ R_2 ! R_1$ or the FD $R_1 \ R_2 ! R_2$.

Dependency-Preserving Decomposition

Consider the Contracts relation with attributes *CSJDPQV* from Section 15.4.1. The given FDs are *C ! CSJDPQV, JP ! C,* and *SD ! P*. Because *SD* is not a key the dependency *SD ! P* causes a violation of BCNF.

Let *R* be a relation schema that is decomposed into two schemas with attribute sets

X and *Y*, and let *F* be a set of FDs over *R*. The projection of F on *X* is the set of

+

FDs in the closure *F* (not just *F* !) that involve only attributes in *X*. We will denote

+

the projection of *F* on attributes *X* as F_X . Note that a dependency *U ! V* in *F*is in

F_X only if *all* the attributes in *U* and *V* are in *X*.

The decomposition of relation schema *R* with FDs *F* into schemas with attribute sets

X and *Y* is dependency-preserving if (F_X [F_Y) + = *F* +. That is, if we take the

dependencies in F_X and F_Y and compute the closure of their union, we get back all dependencies in the closure of *F*. Therefore, we need to enforce only the

dependencies in F_X and F_Y ; all FDs in *F* + are then sure to be satisfied. To enforce

F_X , we need to examine only relation *X* (on inserts to that relation). To enforce F_Y , we need to examine only relation *Y*.

NORMALIZATION

Having covered the concepts needed to understand the role of normal forms and de-compositions in database design, we now consider algorithms for converting relations to BCNF or 3NF. If a relation schema is not in BCNF, it is possible to obtain a lossless-join decomposition into a collection of BCNF relation schemas.

Unfortunately, there may not be any dependency-preserving decomposition into a collection of BCNF relation schemas

Decomposition into BCNF

We now present an algorithm for decomposing a relation schema *R*

into a collection of BCNF relation schemas:

1. Suppose that *R* is not in BCNF. Let *X R, A* be a single attribute in

R, and *X ! A*

be an FD that causes a violation of BCNF. Decompose *R*

into *R* −> *A* and *X*□*A*.

2. If either *R* −> *A* or *X*□*A* is not in BCNF, decompose them further by a recursive application of this algorithm.

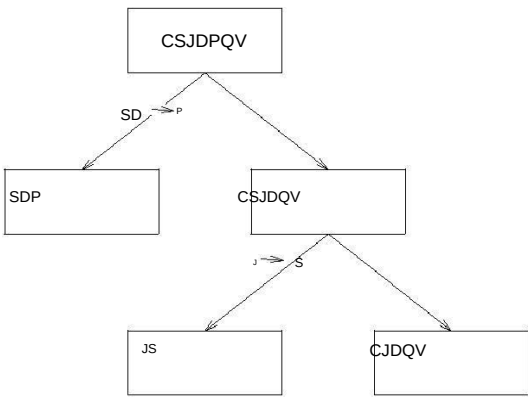
R −> *A* denotes the set of attributes other than *A* in *R,* and *X*□*A* denotes the union of attributes in *X* and *A*. Since *X ! A* violates BCNF, it is not a trivial dependency; further, *A* is a single attribute. Therefore, *A* is not in *X*; that is, *X* *A* is empty. Thus,each decomposition carried out in Step is lossless-join.

The set of dependencies associated with $R -> A$ and XA is the projection of F onto their attributes. If one of the new relations is not in BCNF, we decompose it further in Step. Since a decomposition results in relations with strictly fewer attributes, this process will terminate, leaving us with a collection of relation schemas that are all in BCNF.

Consider the Contracts relation with attributes *CSJDPQV* and key *C*. We are given FDs *JP ! C* and *SD ! P*. By using the dependency *SD ! P* to guide the decomposition, we get the two schemas *SDP* and *CSJDQV*. *SDP* is in BCNF. Suppose that we also have the constraint that each project deals with a single supplier: *J ! S*. This means that the schema *CSJDQV* is not in BCNF. So we decompose it further into *JS* and *CJDQV*. *C ! JDQV* holds over *CJDQV*; the only other FDs that hold are those obtained from this FD by augmentation, and therefore all FDs contain a key in the left side. Thus, each of the

schemas *SDP*, *JS*, and *CJDQV* is in BCNF, and this collection of schemas also represents a lossless-join decomposition of *CSJDQV*.

The steps in this decomposition process can be visualized as a tree, as shown in Figure. The root is the original relation *CSJDPQV*, and the leaves are the BCNF relations that are the result of the decomposition algorithm, namely, *SDP*, *JS*, and *CSDQV*. Intuitively, each internal node is replaced by its children through a single decomposition step that is guided by the FD shown just below the node.



Redundancy in BCNF Revisited

The decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV* is not dependency-preserving. Intuitively, dependency *JP ! C* cannot be enforced without a join. One way to deal with this situation is to add a relation with attributes *CJP*.

This is a subtle point: Each of the schemas *CJP*, *SDP*, *JS*, and *CJDQV* is in BCNF, yet there is some redundancy that can be predicted by FD information. In particular, if we join the relation

instances for *SDP* and *CJDQV* and project onto the attributes *CJP*, we must get exactly the instance stored in the relation with schema *CJP*.

Minimal Cover for a Set of FDs

A minimal cover for a set *F* of FDs is a set *G* of FDs such that:

1. Every dependency in *G* is of the form *X ! A*, where *A* is a single attribute.
2. The closure *F +* is equal to the closure *G+*.
3. If we obtain a set *H* of dependencies from *G* by deleting one or more dependencies, or by deleting attributes from a dependency in *G*, then *F + 6= H+*.

Intuitively, a minimal cover for a set *F* of FDs is an equivalent set of dependencies that is *minimal* in two respects: (1) Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute.

(2) Every dependency in it is required in order for the closure to be equal to *F +*. As an example, let *F* be the set of dependencies:

A ! B, ABCD ! E, EF ! G, EF ! H, and *ACDF ! EG*.

First, let us rewrite *ACDF ! EG* so that every right side is a single attribute: *ACDF ! E* and *ACDF ! G*.

Next consider *ACDF ! G*. This dependency is implied by the following FDs:

A ! B, ABCD ! E, and *EF ! G*.

Therefore, we can delete it. Similarly, we can delete *ACDF ! E*. Next consider *ABCD*

! E. Since *A ! B* holds, we can replace it with *ACD ! E*. (At this point, the reader should verify that each remaining FD is minimal and required.) Thus, a minimal cover for *F* is the set:

A ! B, ACD ! E, EF ! G, and *EF ! H*.

The preceding example suggests a general algorithm for obtaining a minimal cover of a set *F* of FDs:

1. Put the FDs in a standard form: Obtain a collection *G* of equivalent FDs with a single attribute on the right side (using the decomposition axiom).
2. Minimize the left side of each FD: For each FD in *G*, check each attribute in the left side to see if it can be deleted while preserving equivalence to *F +*.

3. Delete redundant FDs: Check each remaining FD in *G* to see if it can be deleted while preserving equivalence to *F* +.

Dependency-Preserving Decomposition into 3NF

Returning to the problem of obtaining a lossless-join, dependency- preserving decom-position into 3NF relations, let *R* be a relation with a set *F* of FDs that is a

minimal cover, and let $R_1; R_2; : : : ; R_n$ be a lossless-join decomposition of *R*. For 1 *i n*, suppose that each R_i is in 3NF and let F_i denote the projection of *F* onto the attributes of R_i. Do the following:

Identify the set *N* of dependencies in *F* that are not preserved, that is, not included in the closure of the union of F_is.

For each FD *X ! A* in *N* , create a relation schema *XA* and add it to the decom-position of *R*.

Obviously, every dependency in *F* is preserved if we replace *R* by the R_is plus the schemas of the form *XA* added in this step. The R_is are given to be in 3NF. We can show that each of the schemas *XA* is in 3NF as follows: Since *X ! A* is in the minimal cover *F*, *Y ! A* does not hold for any *Y* that is a strict subset of *X*. Therefore, *X* is a key for *XA*.

As an optimization, if the set *N* contains several FDs with the same left side, say, *X !*

$A_1; X ! A_2; : : : ; X ! A_n$, we can replace them with a single equivalent FD $X ! A_1 : : :$

A_n. Therefore, we produce one relation schema $XA_1 : : : A_n$, instead of several

schemas $XA_1; : : : ; XA_n$, which is generally preferable.

Comparing this decomposition with the one that we obtained earlier in this section, we find that they are quite close, with the only difference being that one of them has *CDJPQV* instead of *CJP* and *CJDQV*. In general, however, there could be significant differences. Database designers typically use a conceptual design methodology (e.g., ER design) to arrive at an initial database design. Given this, the approach of repeated decompositions to rectify instances of redundancy is likely to be the most natural use of FDs and normalization techniques. However, a designer can also consider the alternative designs suggested by the synthesis approach.

Multivalued Dependencies

Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*. The meaning of a tuple is that teacher *T* can teach course *C*, and book

B is a recommended text for the course. There are no FDs; the key is *CTB*.

However, the recommended texts for a course are independent of the instructor.

The instance shown in Figure 15.13 illustrates this situation.

course	*teacher*	*book*
Physics1 01	Green	Mechanic s
Physics1 01	Green	Optics
Physics1 01	Brown	Mechanic s
Physics1 01	Brown	Optics
Math301	Green	Mechanic s
Math301	Green	Vectors
Math301	Green	Geometry

BCNF Relation with Redundancy That Is Revealed by MVDs

There are three points to note here:

The relation schema *CTB* is in BCNF; thus we would not consider decomposing

it further if we looked only at the FDs that hold over *CTB.*

There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.

The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB.*

This table suggests another way to think about MVDs: If *X !! Y*

holds over *R*, then *Y*

Z (*X*=*x*(*R*)) = *Y* (*X*=*x*(*R*)) *Z* (*X*=*x*(*R*)) in every legal instance of *R*, for any value *x* that appears in the *X* column of *R*. In other words, consider groups of tuples in *R* with the same *X*-value, for each

X-value. In each such group consider the projection onto the attributes *YZ*. This projection must be equal to the cross-product of the projections onto *Y* and *Z*. That is, for a given *X*-value, the *Y*-values and *Z*-values are independent. (From this de nition it is easy to see that *X !! Y* must hold whenever *X ! Y* holds. If the FD *X ! Y* holds, there is exactly one *Y*-value for a given *X*-value, and the conditions in the MVD de nition hold trivially. The converse does not hold, as Figure 15.14 illustrates.)

Returning to our *CTB* example, the constraint that course texts are independent of instructors can be expressed as *C !! T*. In terms of the de nition of MVDs, this constraint can be read as follows:

\If (there is a tuple showing that) *C* is taught by teacher *T*, and (there is a tuple showing that) *C* has book *B* as text, then (there is a tuple showing that) *C* is taught by *T* and has text *B*.

Given a set of FDs and MVDs, in general we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three Armstrong Axioms plus have additional rules. Three of the additional rules involve only MVDs:

- MVD Complementation: If *X !! Y*, then *X !! R − XY* .
- MVD Augmentation: If *X !! Y* and *W Z*, then *WX !! YZ*.
- MVD Transitivity: If *X !! Y* and *Y !! Z*, then *X !!* (*Z − Y*).

As an example of the use of these rules, since we have *C !! T* over *CTB*, MVD complementation allows us to infer that *C !! CT B − CT* as well, that is, *C !! B*. The remaining two rules relate FDs and MVDs:

Replication: If *X ! Y*, then *X !! Y*.

Coalescence: If *X !! Y* and there is a *W* such that *W \ Y* is empty, *W ! Z*, and *Y*

Z, then *X ! Z*.

Observe that replication states that every FD is also an MVD.

Fourth Normal Form

Fourth normal form is a direct generalization of BCNF. Let *R* be a relation schema, *X* and *Y* be nonempty subsets of the attributes of *R*, and *F* be a set of dependencies that includes both FDs and MVDs. *R* is said to be in fourth normal form (4NF) if for every MVD *X !! Y* that holds over *R*, one of the following statements is true:

- *Y X* or *XY* = *R*, or
- *X* is a Superkey.

In reading this definition, it is important to understand that the de nition of a key has not changed the key must uniquely determine all attributes through FDs alone. *X !! Y* is a trivial MVD if *Y X R* or *XY = R*; such MVDs always hold.

The relation *CTB* is not in 4NF because *C !! T* is a nontrivial MVD and *C* is not a key. We can eliminate the resulting redundancy by decomposing *CTB* into *CT* and *CB*; each of these relations is then in 4NF.

To use MVD information fully, we must understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions detected using only FD information!|under which we can safely ignore MVD information. That is, using MVD information in addition to the FD information will not reveal any redundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.

If a relation schema is in BCNF, and at least one of its keys consists of a single attribute, it is also in 4NF.

An important assumption is implicit in any application of the preceding result: *The set of FDs identified thus far is indeed the set of all FDs that hold over the relation*. This assumption is important because the result relies on the relation being in BCNF, which in turn depends on the set of FDs that hold over the relation.

Figure shows three tuples from an instance of *ABCD* that satisfies the given MVD *B*

!! *C*. From the definition of an MVD, given tuples *t*1 and *t*2, it follows

B	*C*	*A*	*D*	
b	*c*1	*a*1	*d*1	| tuple *t*1
b	*c*2	*a*2	*d*2	| tuple *t*2
b	*c*1	*a*2	*d*2	| tuple *t*3

Three Tuples from a Legal Instance of *ABCD*

that tuple *t*3 must also be included in the instance.

Consider tuples *t*2 and *t*3. From the given FD *A ! BCD* and the fact that these tuples have the same *A*-value, we can deduce that *c*1 = *c* 2. Thus, we see that the FD *B ! C* must hold over *ABCD* whenever the FD *A ! BCD* and the MVD *B !! C* hold. If *B ! C* holds, the relation *ABCD* is not in BCNF (unless additional FDs hold that make *B* a key)!

Join Dependencies

A join dependency is a further generalization of MVDs. A join dependency (JD)

*./ fR*1*; : : : ; R ng* is said to hold over a relation *R* if *R*1*; : : : ; Rn*

is a lossless-join decomposition of *R*.

An MVD *X !! Y* over a relation *R* can be expressed as the join dependency *./ f*XY, X(R−Y)*g*. As an example, in the *CTB* relation, the MVD *C !! T* can be expressed as the join dependency *./ f*CT, CB*g*.

Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

Fifth Normal Form

A relation schema *R* is said to be in fth normal form (5NF) if for every JD *./ fR*1*; : : : ; Rng* that holds over

R, one of the following statements is true:

- Ri = R for some i, or
- The JD is implied by the set of those FDs over R in which the left side is a key for R.

The second condition deserves some explanation, since we have not presented inference rules for FDs and JDs taken together. Intuitively, we must be able to show that the decomposition of R into fR1; : : : ; Rng is lossless-join whenever the key dependencies (FDs in which the left side is a key for R) hold. ./ fR1; : : : ; Rng is a trivial JD if Ri = R for some i; such a JD always holds. The following result, also due to Date and Fagin, identifies conditions again, detected using only FD information under which we can safely ignore JD information.

If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF.

The conditions identified in this result are sufficient for a relation to be in 5NF, but not necessary. The result can be very useful in practice because it allows us to conclude that a relation is in 5NF without ever identifying the MVDs and JDs that may hold over the relation.

UNIT-IV

TRANSACTION MANAGEMENT

What is a Transaction?

A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. If you have any concept of Operating Systems, then we can say that a transaction is analogous to processes.

Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. That is, we may say that these transactions bring the database from an image which existed before the transaction occurred (called the **Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

The Four Properties of Transactions

Every transaction, for whatever purpose it is being used, has the following four properties. Taking the initial letters of these four properties we collectively call them the **ACID Properties**. Here we try to describe them and explain them.

Atomicity: This means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected.

Say for example, we have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

Read A;

A = A – 100;

Write A; Read B;

B = B + 100;

Write B;

Fine, is not it? The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Rs 900/- in A and Rs 1100/- in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Rs 900/- in A, but the same Rs

1000/- in B. It would be said that Rs 100/- evaporated in thin air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Rs 1000/-, as if the failed transaction had never occurred.

Consistency: If we execute a particular transaction in isolation or together with other transaction, (i.e. presumably in a multi-programming environment), the transaction will yield the same expected result.

To give better performance, every database management system supports the execution of multiple transactions at the same time, using CPU Time Sharing. Concurrently executing transactions may have to deal with the problem of sharable resources, i.e. resources that multiple transactions are trying to read/write at the same time. For example, we may have a table or a record on which two transaction are trying to read or write at the same time. Careful mechanisms are created in order to prevent mismanagement of these sharable resources, so that there should not be any change in the way a transaction performs. A transaction which deposits Rs 100/- to account A must deposit the same amount whether it is acting alone or in conjunction with another transaction that may be trying to deposit or withdraw some amount at the same time.

Isolation: In case multiple transactions are executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that they should not create any anomaly in the value stored at the sharable resource.

There are several ways to achieve this and the most popular one is using some kind of locking mechanism. Again, if you have the concept of Operating Systems, then you should remember the semaphores, how it is used by a process to make a resource busy before starting to use it, and how it is used to release the resource after the usage is over. Other processes intending to access that same resource must wait during this time. Locking is almost similar. It states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

Durability: It states that once a transaction has been complete the changes it has made should be permanent.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has

made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.

If you study carefully, you can understand that Atomicity and Durability is essentially the same thing, just as Consistency and Isolation is essentially the same thing.

Transaction States

There are the following six states in which a transaction may exist:

Active: The initial state when the transaction has just started execution.

Partially Committed: At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.

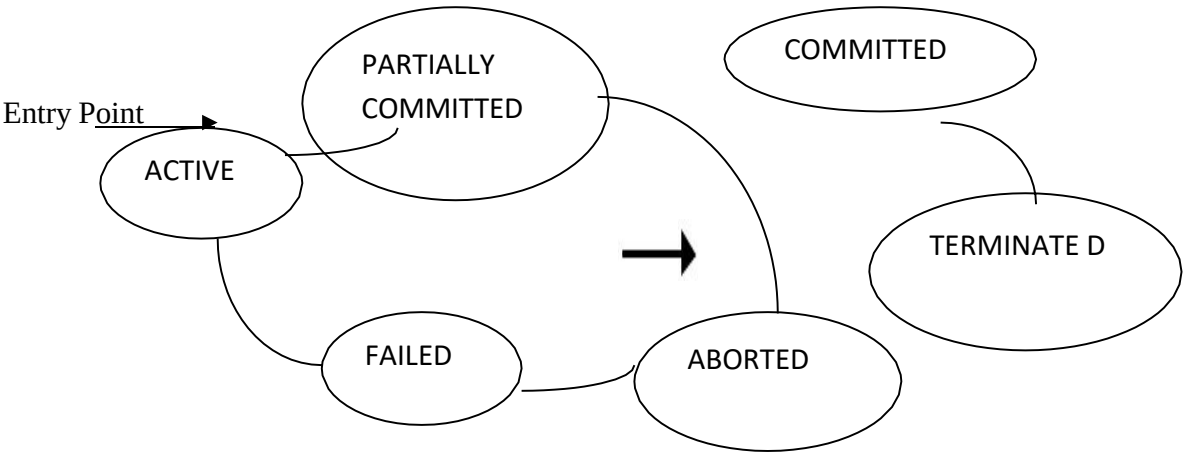
Failed: If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.

Aborted: When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.

Committed: If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.

Terminated: Either committed or aborted, the transaction finally reaches this state.

The whole process can be described using the following diagram:



Concurrent Execution

A schedule is a collection of many transactions which is implemented as a unit. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

- **Serial:** The transactions are executed one after another, in a non-preemptive manner.
- **Concurrent:** The transactions are executed in a preemptive, time shared method.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.

In concurrent schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.

Let us consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

T1

Read A;

A = A – 100;

Write A; Read B;

B = B + 100;

Write B;

T2

Read A;

Temp = A * 0.1; Read C;

C = C + Temp; Write C;

T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

T1	T2
Read A;	
A = A – 100;	
Write A;	
	Read A;
	Temp = A * 0.1; Read C;
	C = C + Temp; Write C;
Read B;	
B = B + 100;	
Write B;	

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Rs 100/- from A and writes the new value of Rs 900/- into A. T2 reads the value of A, calculates the value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.

It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

T1	T2
Read A;	
A = A – 100;	
	Read A;
	Temp = A * 0.1; Read C;
	C = C + Temp;
	Write C;
Write A; Read B;	
B = B + 100;	
Write B;	

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Rs 1000/- each, then the result of this schedule should have left Rs 900/- in A, Rs 1100/- in B and add Rs 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Rs 900/- has been updated in A. T2 reads the old value of A, which is still Rs 1000/-, and deposits Rs 100/- in C. C makes an unjust gain of Rs 10/- out of nowhere.

Serializability

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions

will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.

2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.

3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create two different schedules

S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.

2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.

3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

Let us consider a schedule *S* in which there are two consecutive instructions, *I* and *J* , of transactions *Ti* and *Tj* , 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:

- I = read(Q), J = read(Q). The order of I and J does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.
- I = read(Q), J = write(Q). If I comes before J , then Ti does not read the value of Q that is written by Tj in instruction J . If J comes before I, then Ti reads the value of Q that is written by Tj. Thus, the order of I and J matters.
- I = write(Q), J = read(Q). The order of I and J matters for reasons similar to those of the previous case.

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

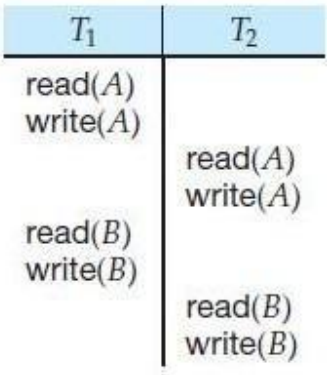


Fig: Schedule 3—showing only the read and write instructions.

We say that I and J **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3in Figure above. The write(A) instruction of $T1$ conflicts with the read(A) instruction of $T2$. However, the write(A) instruction of $T2$ does not conflict with the read(B) instruction of $T1$, because the two instructions access different data items.

Transaction Characteristics

Every transaction has three characteristics: *access mode*, *diagnostics size*, and *isolation level*. The **diagnostics size** determines the number of error conditions that can be recorded.

If the **access mode** is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. For transactions with READ ONLY access mode, only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure given below. In this context, *dirty read* and *unrepeatable read* are defined as usual. **Phantom** is defined to be the possibility that a transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself.

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 19.3.1), and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that *T* reads only the changes made by committed transactions, and that no value read or written by *T* is changed by any other transaction until *T* is complete. However, *T* could experience the phantom phenomenon; for example, while *T* examines all

Sailors records with *rating=1*, another transaction might add a new such Sailors record, which is missed by *T*.

A REPEATABLE READ transaction uses the same locking protocol as a SERIALIZABLE transaction, except that it does not do index locking, that is, it locks only individual objects, not sets of objects.

READ COMMITTED ensures that *T* reads only the changes made by committed transactions, and that no value written by *T* is changed by any other transaction until *T* is complete. However, a value read by *T* may well be modified by another transaction while *T* is still in progress, and *T* is, of course, exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself - a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects, and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance.

For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level, or even the READ UNCOMMITTED level, because a few incorrect or missing values will not significantly affect the result if the number of sailors is large. The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY

When a transaction is started, the default is SERIALIZABLE and READ WRITE.

PRECEDENCE GRAPH

A precedence graph, also named conflict graph and serializability graph, is used in the context of concurrency control in databases.

The precedence graph for a schedule S contains:

A node for each committed transaction in S

An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions.

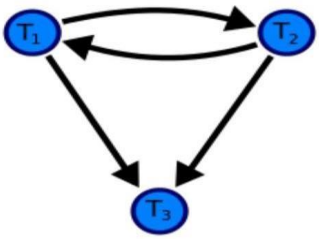
A precedence graph of the schedule D, with 3 transactions. As there is a cycle (of length 2; with two edges) through the committed transactions T1 and T2, this schedule (history) is not Conflict serializable.

The drawing sequence for the precedence graph:-

1. For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph. So the precedence graph contains T_1, T_2, T_3
2. For each case in S where T_i executes a write_item(X) then T_j executes a read_item(X), create an edge (T_i --> T_j) in the precedence graph. This occurs nowhere in the above example, as there is no read after write.

Precedence graph

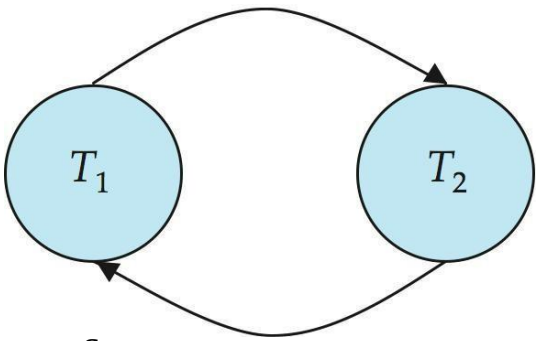
$$D = \begin{bmatrix} T1 & T2 & T3 \\ & R(B) & \\ R(C) & W(A) & \\ W(C) & & \\ R(D) & & \\ & & W(B) \\ W(D) & & W(A) \end{bmatrix}$$



3. For each case in S where T_i executes a read_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edge from T_1 to T_2.
4. For each case in S where T_i executes a write_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edges from T_2 to T_1, T_1 to T_3, and T_2 to T_3.
5. The schedule S is conflict serializable if the precedence graph has no cycles. As T_1 and T_2 constitute a cycle, then we cannot declare S as serializable or not and serializability has to be checked using other methods.

TESTING FOR CONFLICT SERIALIZABILITY

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- To test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm.Cycle-detection algorithms exist which take order n2 time, where n is the number of vertices in the graph. (Better algorithms take order n + e where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. That is, a linear order consistent with the partial order of the graph.
- For example, a serializability order for the schedule (a) would be one of either (b) or (c)



A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph.

RECOVERABLE SCHEDULES

- Recoverable schedule — if a transaction Tj reads a data item previously written by a transaction Ti , then the commit operation of Ti must appear before the commit operation of Tj.
- The following schedule is not recoverable if T9 commits immediately after the read(A) operation.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T8 should abort, T9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

CASCADING ROLLBACKS

- Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

- If T10 fails, T11 and T12 must also be rolled back.

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

- Can lead to the undoing of a significant amount of work

CASCADELESS SCHEDULES

- Cascadeless schedules — for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- Example of a schedule that is NOT cascadeless

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

CONCURRENCY SCHEDULE

- A database must provide a mechanism that will ensure that all possible schedules are both:
- Conflict serializable.
- Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability after it has executed is a little too late!
- Tests for serializability help us understand why a concurrency control protocol is correct
- Goal – to develop concurrency control protocols that will assure serializability.

WEEK LEVELS OF CONSISTENCY

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

- E.g., a read-only transaction that wants to get an approximate total balance of all accounts
- E.g., database statistics computed for query optimization can be approximate (why?)
- Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

LEVELS OF CONSISTENCY IN SQL

- Serializable — default
- Repeatable read — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- Read committed — only committed records can be read, but successive reads of record may return different (but committed) values.
- Read uncommitted — even uncommitted records may be read.
- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

TRANSACTION DEFINITION IN SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
- Commit work commits current transaction and begins a new one.
- Rollback work causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
- Implicit commit can be turned off by a database directive
- E.g. in JDBC, connection.setAutoCommit(false);

RECOVERY SYSTEM

Failure Classification:

- Transaction failure :
- Logical errors: transaction cannot complete due to some internal error condition
- System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash: a power failure or other hardware or software failure causes the system to crash.

- Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted as result of a system crash
- Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure: a head crash or similar disk failure destroys all or part of disk storage
- Destruction is assumed to be detectable: disk drives use checksums to detect failures

RECOVERY ALGORITHMS

- Consider transaction Ti that transfers $50 from account A to account B
- Two updates: subtract 50 from A and add 50 to B
- Transaction Ti requires updates to A and B to be output to the database.
- A failure may occur after one of these modifications have been made but before both of them are made.
- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
- Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
- Actions taken during normal transaction processing to ensure enough information exists to recover from failures
- Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

STORAGE STRUCTURE

- Volatile storage:
- does not survive system crashes
- examples: main memory, cache memory
- Nonvolatile storage:
- survives system crashes
- examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
- but may still fail, losing data
- Stable storage:
- a mythical form of storage that survives all failures
- approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
- copies can be at remote sites to protect against disasters such as fire or flooding.

- Failure during data transfer can still result in inconsistent copies. Block transfer can result in
- Successful completion
- Partial failure: destination block has incorrect information
- Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
- Execute output operation as follows (assuming two copies of each block):
    - Write the information onto the first physical block.
    - When the first write successfully completes, write the same information onto the second physical block.
    - The output is completed only after the second write successfully completes.
- Copies of a block may differ due to failure during output operation. To recover from failure:
- First find inconsistent blocks:
- Expensive solution: Compare the two copies of every disk block.
- Better solution:
- Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
- Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
- Used in hardware RAID systems
- If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

DATA ACCESS

- Physical blocks are those blocks residing on the disk.
- System buffer blocks are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
- input(B) transfers the physical block B to main memory.
- output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Each transaction Ti has its private work-area in which local copies of all data items accessed and updated by it are kept.
- Ti's local copy of a data item X is denoted by xi.
- BX denotes block containing X

- Transferring data items between system buffer blocks and its private work-area done by:
- read(X) assigns the value of data item X to the local variable xi.
- write(X) assigns the value of local variable xi to data item {X} in the buffer block.
- Transactions
- Must perform read(X) before accessing X for the first time (subsequent reads can be from local copy)
- The write(X) can be executed at any time before the transaction commits
- Note that output(BX) need not immediately follow write(X). System can perform the output operation when it seems fit.

Lock-Based Protocols

A lock is a mechanism to control concurrent access to a data item Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.

2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

1) A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

2) Any number of transactions can hold shared locks on an item,

but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

3) If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example of a transaction performing locking:

T_2: **lock-S**(*A*); **read** (*A*); **unlock**(*A*); **lock-S**(*B*);

read (*B*); **unlock**(*B*); **display**(*A*+*B*)

Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Consider the partial schedule

T_3	T_4
lock-x (*B*)	
read (*B*)	
B := *B* - 50	
write (*B*)	
	lock-s (*A*)
	read (*A*)
	lock-s (*B*)
lock-x (*A*)	

Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.

Such a situation is called a **deadlock**.

1. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.
2. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
3. **Starvation** is also possible if concurrency control manager is badly designed. For example:
    a. A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
    b. The same transaction is repeatedly rolled back due to deadlocks.
4. Concurrency control manager can be designed to prevent starvation.

THE TWO-PHASE LOCKING PROTOCOL

1.This is a protocol which ensures conflict-serializable schedules.

2.Phase 1: Growing Phase

a.transaction may obtain locks

b.transaction may not release locks

3. Phase 2: Shrinking Phase

a.transaction may release locks

b.transaction may not obtain locks

4. The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

5. Two-phase locking *does not* ensure freedom from deadlocks

6. Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.

7. **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

8. There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

9. However, in the absence of extra information (e.g., ordering of access to data), two- phase locking is needed for conflict serializability in the following sense:

Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

TIMESTAMP-BASED PROTOCOLS

1. Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).

2. The protocol manages concurrent execution such that the time-stamps determine the serializability order.

3. In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:

a.W-timestamp(*Q*) is the largest time-stamp of any transaction that executed write(*Q*) successfully.

b.R-timestamp(*Q*) is the largest time-stamp of any transaction that executed read(*Q*) successfully.

4. The timestamp ordering protocol ensures that any conflicting **read** and **write**

operations are executed in timestamp order.

5.Suppose a transaction T_i issues a **read**(*Q*)

1. If TS(T_i) □ **W**-timestamp(*Q*), then T_i needs to read a value of *Q* that was already overwritten.

   n Hence, the **read** operation is rejected, and T_i is rolled back.

2. If TS(T_i)□ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS(T_i)).

6. Suppose that transaction T_i issues **write**(*Q*).

1. If TS(T_i) < R-timestamp(*Q*), then the value of *Q* that T_i is producing was needed previously, and the system assumed that that value would never be produced.

   n Hence, the **write** operation is rejected, and T_i is rolled back.

2. If TS(T_i) < W-timestamp(*Q*), then T_i is attempting to write an obsolete value of *Q*. n Hence, this **write** operation is rejected, and T_i is rolled back.

3. Otherwise, the **write** operation is executed, and W-timestamp(*Q*) is set to TS(T_i).

Thomas' Write Rule

1. We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol i.e., Timestamp ordering Protocol . Let us consider schedule 4 of Figure below, and apply the timestamp-ordering protocol. Since *T*27 starts before *T*28, we shall assume that TS(*T*27) < TS(*T*28). The read(*Q*) operation of *T*27 succeeds, as does the write(*Q*) operation of *T*28. When *T*27 attempts its write(*Q*) operation, we find that TS(*T*27) <

W-timestamp(Q), since Wtimestamp(Q) = TS($T28$). Thus, the write(Q) by $T27$ is rejected and transaction $T27$ must be rolled back.

2. Although the rollback of $T27$ is required by the timestamp-ordering protocol, it is unnecessary. Since $T28$ has already written Q, the value that $T27$ is attempting to write is one that will never need to be read. Any transaction Ti with TS(Ti) < TS($T28$) that attempts a read(Q)will be rolled back, since TS(Ti)<W-timestamp(Q).

3. Any transaction Tj with TS(Tj) > TS($T28$) must read the value of Q written by $T28$, rather than the value that $T27$ is attempting to write. This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp- ordering protocol.

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

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this:
Suppose that transaction Ti issues write(Q).

1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.

2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and setsW-timestamp(Q) to TS(Ti).

VALIDATION-BASED PROTOCOLS

Phases in Validation-Based Protocols

1) Read phase. During this phase, the system executes transaction Ti. It reads the values of the various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.

2) Validation phase. The validation test is applied to transaction Ti. This determines whether Ti is allowed to proceed to the write phase without causing a violation of serializability.

If a transaction fails the validation test, the system aborts the transaction.

3) Write phase. If the validation test succeeds for transaction Ti, the temporary local variables that hold the results of any write operations performed by Ti are copied to the database. Read-only transactions omit this phase.

MODES IN VALIDATION-BASED PROTOCOLS

1. Start(Ti)
2. Validation(Ti)
3. Finish

MULTIPLE GRANULARITY.

multiple granularity locking (MGL) is a locking method used in database management systems (DBMS) and relational databases.

In MGL, locks are set on objects that contain other objects. MGL exploits the hierarchical nature of the contains relationship. For example, a database may have files, which contain pages, which further contain records. This can be thought of as a tree of objects, where each node contains its children. A lock on such as a shared or exclusive lock locks the targeted node as well as all of its descendants.

Multiple granularity locking is usually used with non-strict two-phase locking to guarantee serializability. The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:

- Transaction *Ti* must observe the lock-compatibility function of Figure above.
- Transaction *Ti* must lock the root of the tree first, and can lock it in anymode.
- Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q*
- locked in either IX or IS mode.
- Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.
- Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).
- Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of
- *Q* locked.

UNIT-V

RECOVERY AND ATOMICITY

RECOVERY AND ATOMICITY

FAILURE WITH LOSS OF NON-VOLATILE STORAGE

What would happen if the non-volatile storage like RAM abruptly crashes? All transaction, which are being executed are kept in main memory. All active logs, disk buffers and related data is stored in non-volatile storage.

When storage like RAM fails, it takes away all the logs and active copy of database. It makes recovery almost impossible as everything to help recover is also lost. Following techniques may be adopted in case of loss of non-volatile storage.

- A mechanism like checkpoint can be adopted which makes the entire content of database be saved periodically.
- State of active database in non-volatile memory can be dumped onto stable storage periodically, which may also contain logs and active transactions and buffer blocks.
- <dump> can be marked on log file whenever the database contents are dumped from non-volatile memory to a stable one.

RECOVERY:

- When the system recovers from failure, it can restore the latest dump.
- It can maintain redo-list and undo-list as in checkpoints.
- It can recover the system by consulting undo-redo lists to restore the state of all transaction up to last checkpoint.

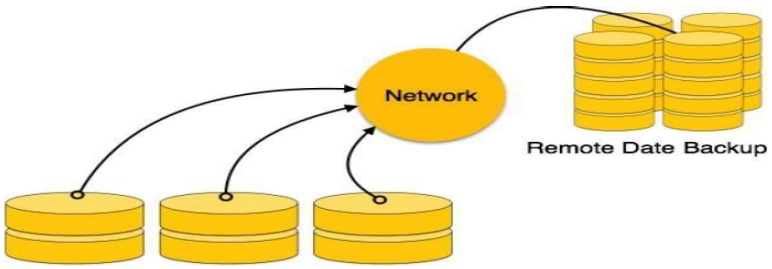
DATABASE BACKUP & RECOVERY FROM CATASTROPHIC FAILURE

Remote backup, described next, is one of the solutions to save life. Alternatively, whole database backups can be taken on magnetic tapes and stored at a safer place. This backup can later be restored on a freshly installed database and bring it to the state at least at the point of backup.

REMOTE BACKUP

Remote backup provides a sense of security and safety in case the primary location where the database is located gets destroyed. Remote backup can be offline or real-time and online. In case it is offline it is maintained manually.

[Image: Remote Data Backup]



DBMS DATA RECOVERY

CRASH RECOVERY

Though we are living in highly technologically advanced era where hundreds of satellite monitor the earth and at every second billions of people are connected through information technology, failure is expected but not every time acceptable.

FAILURE CLASSIFICATION

To see where the problem has occurred we generalize the failure into various categories, as follows:

Transaction failure When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt. Reason for transaction failure could be:

- **Logical errors:** where a transaction cannot complete because of it has some code error or any internal error condition
- **System errors:** where the database system itself terminates an active transaction because DBMS is not able to execute it or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability systems aborts an active transaction.

SYSTEM CRASH

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supply, failure of underlying hardware or software failure. Examples may include operating system errors.

DISK FAILURE: In early days of technology evolution, it was a common problem where hard disk drives or storage drives used to fail frequently. Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage

STORAGE STRUCTURE

We have already described storage system here. In brief, the storage structure can be divided in various categories:

- **Volatile storage:** As name suggests, this storage does not survive system crashes and mostly placed very closed to CPU by embedding them onto the chipset itself for examples: main memory, cache memory. They are fast but can store a small amount of information.
- **Nonvolatile storage:** These memories are made to survive system crashes. They are huge in data storage capacity but slower in accessibility. Examples may include, hard disks, magnetic tapes, flash memory, non-volatile (battery backed up) RAM.

RECOVERY AND ATOMICITY

When a system crashes, it many have several transactions being executed and various files opened for them to modifying data items. As we know that transactions are made of various operations, which are atomic in nature.

- It should check the states of all transactions, which were being executed.
- A transaction may be in the middle of some operation; DBMS must ensure the atomicity of transaction in this case. It should check whether the transaction can be completed now or needs to be rolled back.
- No transactions would be allowed to left DBMS in inconsistent state. There are two types of techniques, which can help DBMS in recovering as well as maintaining the atomicity of transaction:
- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where are the changes are done on a volatile memory and later the actual database is updated.

LOG-BASED RECOVERY

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to actual modification and stored on a stable storage media, which is failsafe.

Log based recovery works as follows:

- The log file is kept on stable storage media
- When a transaction enters the system and starts execution, it writes a log about it <Tn, Start>
- When the transaction modifies an item X, it write logs as follows: <Tn, X, V1, V2> It reads Tn has changed the value of X, from V1 to V2.

When transaction finishes, it logs: <Tn, commit> Database can be modified using two approaches:

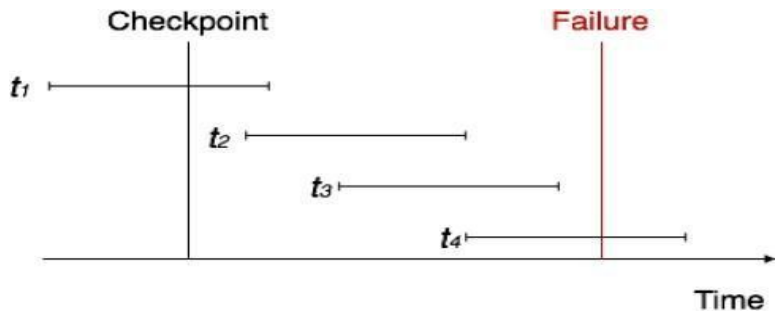
1. **Deferred database modification:** All logs are written on to the stable storage and database is updated when transaction commits.
2. **Immediate database modification:** Each log follows an actual database modification. That is, database is modified immediately after every operation.

RECOVERY WITH CONCURRENT TRANSACTIONS

When more than one transactions are being executed in parallel, the logs are interleaved. At the time of recovery it would become hard for recovery system to backtrack all logs, and then start recovering. To ease this situation most modern DBMS use the concept of 'checkpoints'.

Checkpoint Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. At time passes log file may be too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in storage disk. Checkpoint declares a point before which the DBMS was in consistent state and all the transactions were committed.

Recovery When system with concurrent transaction crashes and recovers, it does behave in the following manner:



BUFFER MANAGEMENT

1. Database buffers are generally implemented in virtual memory in spite of some drawbacks:

   a. When operating system needs to evict a page that has been modified, the page is written to swap space on disk.

   b. When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!

- Known as dual paging problem.

c.Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should

- Output the page to database instead of to swap space (making sure to output log records first), if it is modified
- Release the page from the buffer, for the OS to use

Dual paging can thus be avoided, but common operating systems do not support such functionality.

FUZZY CHECKPOINTING

a.To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing

b.Fuzzy checkpointing is done as follows:

1. Temporarily stop all updates by transactions
2. Write a <**checkpoint** *L*> log record and force log to stable storage
3. Note list *M* of modified buffer blocks
4. Now permit transactions to proceed with their actions
5. Output to disk all modified buffer blocks in list *M*

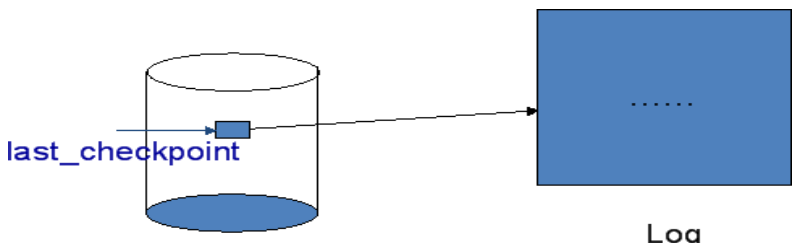
H blocks should not be updated while being output

H Follow WAL: all log records pertaining to a block must be output before the block is output

6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

7. When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**

   a. Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.



FAILURE WITH LOSS OF NONVOLATILE STORAGE

a. So far we assumed no loss of non-volatile storage

b. Technique similar to checkpointing used to deal with loss of non-volatile storage

1. Periodically **dump** the entire content of the database to stable storage

2. No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place

Output all log records currently residing in main memory onto stable storage.

- Output all buffer blocks onto the disk.
- Copy the contents of the database to stable storage.
- Output a record <**dump**> to log on stable storage.

RECOVERING FROM FAILURE OF NON-VOLATILE STORAGE

a. To recover from disk failure

1. restore database from most recent dump.

2. Consult the log and redo all transactions that committed after the dump

b. Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**

1. Similar to fuzzy checkpointing