DATA BASE SYSTEMS

LECTURE NOTES

B.TECH
(II YEAR – II SEM)
(2018-19)

Prepared by:
Mr M.Venu, Assistant Professor

Department of Electrical & Electronics Engineering

MALLA REDDY COLLEGE
OF ENGINEERING &TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
Recognized under 2(f) and 12 (B) of UGC ACT 1956
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Kompally), Secunderabad – 500100, Telangana State, India

UNIT 1

Database: File Processing System Vs DBMS, History, Characteristic-Abstraction levels, Architecture of a database, Functional components of a DBMS, DBMS Languages-Database users and DBA.

File Processing System Vs DBMS

1. A database management system coordinates both the physical and the logical access to the data, whereas a file-processing system coordinates only the physical access.

2. A database management system is designed to allow flexible access to data (i.e. queries), whereas a file-processing system is designed to allow predetermined access to data (i.e. compiled programs).

3. A database management system is designed to coordinate multiple users accessing the same data at the same time. A file-processing system is usually designed to allow one or more programs to access different data files at the same time. In a file-processing system, a file can be accessed by two programs concurrently only if both programs have read-only access to the file.

4. Redundancy is control in DBMS, but not in file system.

5. Unauthorized access is restricted in DBMS but not in the file system.

6. DBMS provide backup and recovery whereas data lost in file system can't be recovered.

7. DBMS provide multiple user interfaces. Data is isolated in file system.

DBMS	File Processing System
Minimal data redundancy problem in DBMS	Data Redundancy problem exits
Data Inconsistency does not exist	Data Inconsistency exist here
Accessing database is easier	Accessing is comparatively difficult
The problem of data isolation is not found in database	Data is scattered in various files and files may be of different format, so data isolation problem exists
Transactions like insert, delete, view, updating, etc are possible in database	In file system, transactions are not possible
Concurrent access and recovery is possible in database	Concurrent access and recovery is not possible
Security of data	Security of data is not good
A database manager (administrator) stores the relationship in form of structural tables	A file manager is used to store all relationships in directories in file systems.

History of Database

1950s and early 1960s:

- Data processing using magnetic tapes for storage
- Tapes provided only sequential access
- Punched cards for input

Late 1960s and 1970s:

- Hard disks allowed direct access to data
- Hierarchical and network data models in widespread use
  - IBM's DL/I (Data Language One)
  - CODAYSL's DBTG (Data Base Task Group) model
    - → the basis of current DBMSs
- Ted Codd defines the relational data model
  - IBM Research develops System R prototype
  - UC Berkeley develops Ingres prototype
- Entity-Relationship Model for database design

1980s:

- Research relational prototypes evolve into commercial systems

    - DB2 from IBM is the first DBMS product based on the relational model
    - Oracle and Microsoft SQL Server are the most prominent commercial DBMS products based on the relational model
  - SQL becomes industrial standard
  - Parallel and distributed database systems
  - Object-oriented database systems (OODBMS)
    - Goal: store object-oriented programming objects in a database without having to transform them into relational format
    - In the end, OODBMS were not commercially successful due to high cost of relational to object-oriented transformation and a sound underlying theory, but they still exist
  - Object-relational database systems allow both relational and object views of data in the same database

Late 1990s:

- Large decision support and data-mining applications
- Large multi-terabyte data warehouses
- Emergence of Web commerce

Early 2000s:

- XML and XQuery standards
- Automated database administration

Later 2000s:

- Web databases (semi-structured data, XML, complex data types)
- Cloud computing
- Giant data storage systems (Google BigTable, Yahoo PNuts, Amazon Web Services, …)

Characteristics of a Database

Stores any kind of Data

A database management system should be able to store any kind of data. It should not be restricted to the employee name, salary and address. Any kind of data that exists in the real world can be stored in DBMS because we need to work with all kinds of data that is present around us.

Support ACID Properties

Any DBMS is able to support ACID (Accuracy, Completeness, Isolation, and Durability) properties. It is made sure is every DBMS that the real purpose of data should not be lost while performing transactions like delete, insert an update. Let us take an example; if an employee name is updated then it should make sure that there is no duplicate data and no mismatch of student information.

Represents complex relationship between data

Data stored in a database is connected with each other and a relationship is made in between data. DBMS should be able to represent the complex relationship between data to make the efficient and accurate use of data.

Backup and recovery

There are many chances of failure of whole database. At that time no one will be able to get the database back and for sure company will be in a big loss. The only solution is to take backup of database and whenever it is needed, it can be stored back. All the databases must have this characteristic.

Structures and described data

A database should not contains only the data but also all the structures and definitions of the data. This data represent itself that what actions should be taken on it. These descriptions include the structure, types and format of data and relationship between them.

Data integrity

This is one of the most important characteristics of database management system. Integrity ensures the quality and reliability of database system. It protects the unauthorized access of database and makes it more secure. It brings only the consistence and accurate data into the database.

Concurrent use of database

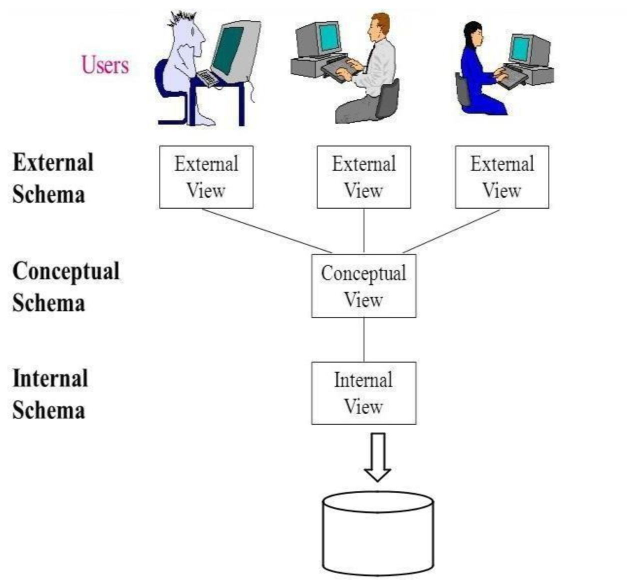
There are many chances that many users will be accessing the data at the same time. They may require altering the database system concurrently. At that time, DBMS supports them to concurrently use the database without any problem.

Abstraction levels

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:



Database Disk

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*
• *student with fields ID, name, dept_name and tot_cred*

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

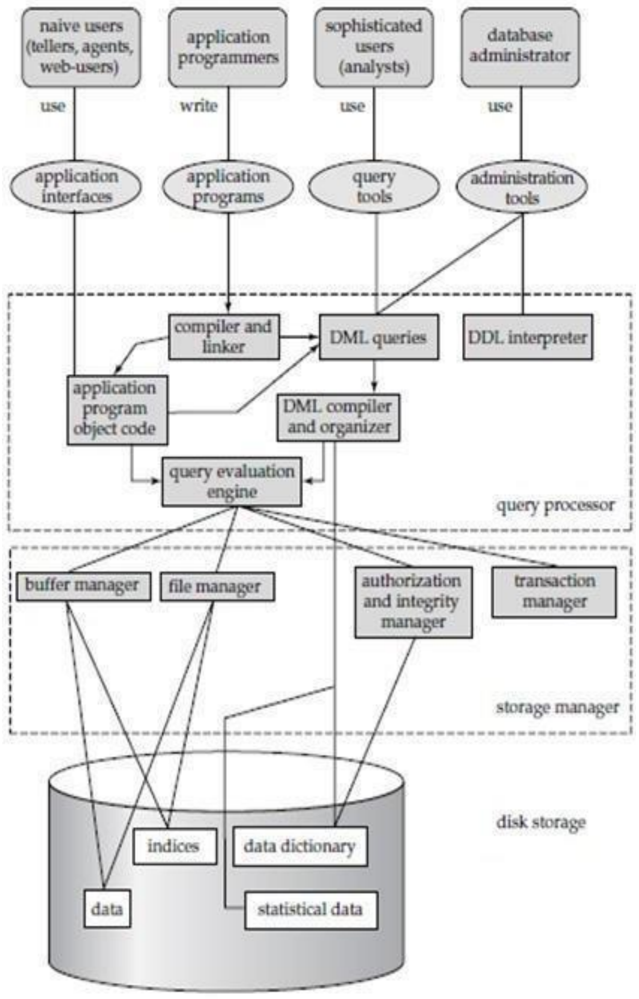
At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well.

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.

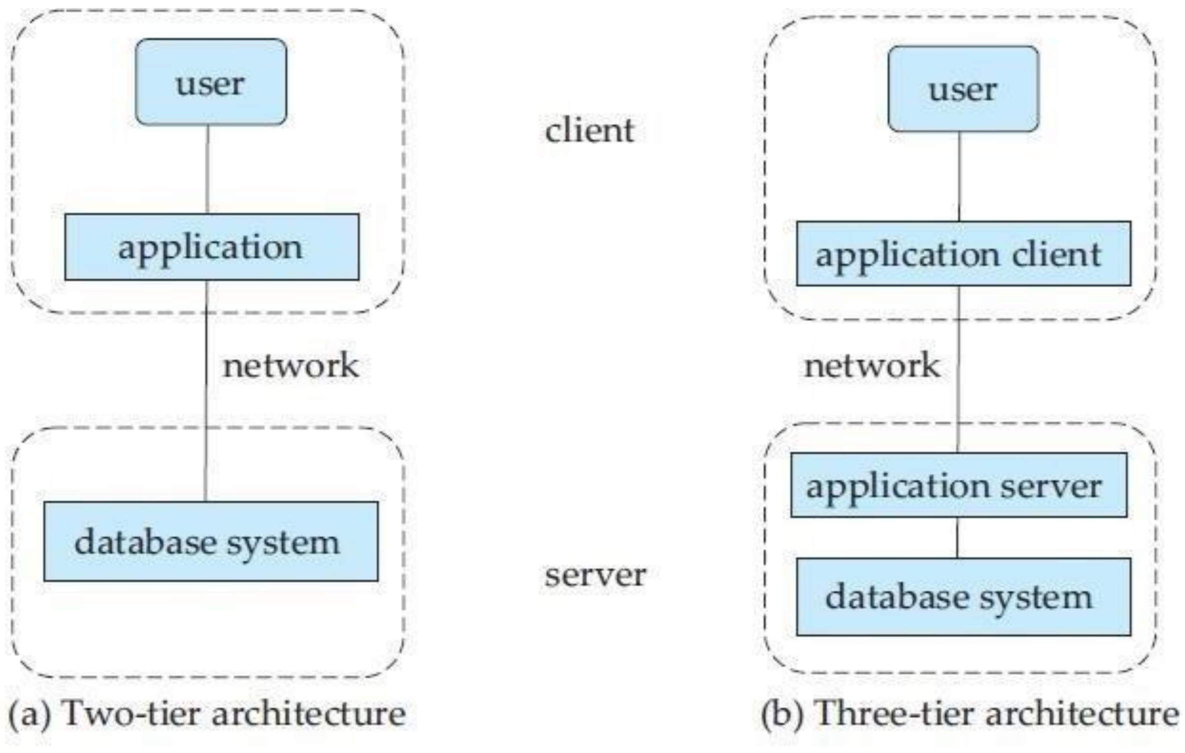
Architecture of a Database

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.



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.



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 low level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager.

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.

Functional components of a DBMS

1. **File Manager** manages the allocation space on disk storage and the data structures used to represent info stored on other media. In most applications (99.9%) the file is the central element. All applications are designed with the specific goal: generation and use of information. A typical file system layered architecture is the following.

User Program
Sequential Indexed Random Lists
Logical I/O
Basic File System Structure
Device Drivers (Disk,tape,etc)
Controllers
Actual Device

2. **Buffer Manager** among other tasks, it transfers blocks between disk (or other devices) and Main Memory (MM). A DMA (Direct Memory Access) is a form of I/O that controls the exchange of blocks between MM and a device. When a processor receives a request for a transfer of a block, it sends it to the DMA which transfers the block uninterrupted.

3. **Query Parser** translates statements in a query language, whether embedded or not, into a lower level language. (See RL language example from CPS510). This parser is also a *strategy selector:* i.e., finding the best and most efficient way (faster?) of executing the query.

4. **Authorization and Integrity Manager** checks for the authority of the users to access and modify info, as well as integrity constraints (keys, etc).

5. **Recovery Manager** ensures that the database is and remains in a consistent (sound) state after any kind of failure.

6. **Concurrency Controller** enforces Mutual Exclusion by ensuring that concurrent interactions with the data base proceed without conflict (deadlocks, etc).

Components of a Database

User: - Users are the one who really uses the database. Users can be administrator, developer or the end users.

Data or Database: - As we discussed already, data is one of the important factor of database. A very huge amount of data will be stored in the database and it forms the main source for all other components to interact with each other. There are two types of data. One is user data. It contains the data which is responsible for the database, i.e.; based on the requirement, the data will be stored in the various tables of the database in the form of rows and columns. Another data is Metadata. It is known as 'data about data', i.e.; it stores the information like how many tables, their names, how many columns and their names, primary keys, foreign keys etc. basically these metadata will have information about each tables and their constraints in the database.

DBMS: - This is the software helps the user to interact with the database. It allows the users to insert, delete, update or retrieve the data. All these operations are handled by query languages like MySQL, Oracle etc.

Database Application: - It the application program which helps the users to interact with the database by means of query languages. Database application will not have any idea about the underlying DBMS.

DBMS Languages

To read data, update and store information in DBMS, some languages are used. Database languages in DBMS are given as below.

- DDL – Data Definition Language
- DML – Data Manipulation Language
- DCL – Data Control Language
- TCL – Transaction Control Language

1. **Data Definition Language (DDL)**

DDL stands for data definition language and used to define database patterns or structures. DDL is a syntax which is same as syntax of computer programming language for defining patterns of database.

Few examples of it are:

- CREATE – used to create objects in database
- ALTER – alter the pattern of database
- DROP – helps in detecting objects
- TRUNCATE – erase all records from table
- COMMENT – adding of comments to data dictionary
- RENAME – useful in renaming an object

CREATE statement or command is used to create a new database. In structured query language the create command creates an object in a relational database management system. The commonly used create command is as follows

- CREATE TABLE [name of table] ([definitions of column]) [parameters of table]

DROP statement destroys or deletes database or table. In structured query language, it also deletes an object from relational database management system. Typically used DROP statement is

- DROP type of object name of object

ALTER statement enhance the object of database. In structured query language it modifies the properties of database object. The ALTER statement is

- ALTER type of object name of object

RENAME statement is used to rename a database. It’s statement is as follows

- RENAME TABLE old name of table to new name of table.

2. **Data manipulation language (DML)**

It has statements which are used to manage the data within the pattern of objects. Some of the samples of the statements are as follows:

- SELECT – useful in holding data from a database
- INSERT – helps in inserting data in to a table

- UPDATE – used in updating the data
- DELETE – do the function of deleting the records
- MERGE – this do the UPSERT operation i.e. insert or update operation
- CALL – this calls a structured query language or a java subprogram
- EXPLAIN PLAN – has the parameter of explaining data
- LOCK TABLE – this ha the function of controlling concurrency

These syntax elements are similar to the syntax elements used in computer programming language. Performing the operation of reading of queries is also a component of data manipulation language. Other forms of data manipulation languages (DML) are used by IMS, CODASYL databases.

DML also include the structured query language (SQL) data modifying statements, they modify the saved data but not the pattern of objects. The initial word of the DML statements has functional capability.

The query statement SELECT is grouped with data statements of structured query language (SQL). In practice there is no such difference and it is viewed to be a portion of DML.

Data manipulation languages contribute to have distinct relishes between database sellers. They are divided as:

- Procedural programming
- Declarative programming

Initially data manipulation languages were only used in computer programs, but with the coming of structured query languages it is also used in the database executors.

3. Data Control Language (DCL)

Data Control Language (DCL) is syntax similar to the programming language, which was used to retrieve the stored or saved data. Examples of the commands in the data control language (DCL) are:

- GRANT – this permits particular users to perform particular tasks
- REVOKE – it blocks the previously granted untrue permissions

The operations which has the authorization of REVOKE are CONNECT, INSERT, USAGE, EXECUTE, DELETE, UPDATE and SELECT.

The execution of DCL is transactional; it also has the parameter of rolling back. But the execution of data control language in oracle database does not have the feature of rolling back.

4. **Transaction Control Language (TCL)**

Transaction Control Language (TCL) has commands which are used to manage the transactions or the conduct of a database. They manage the changes made by data manipulation language statements and also group up the statements in o logical management.

Some examples of it are:

- COMMIT – use to save work
- SAVE POINT – helps in identifying a point in the transaction, can be rolled back to the identified point
- ROLL BACK – has the feature of restoring the database to the genuine point, since from the last COMMIT
- SET TRANSACTION – have parameter of changing settings like isolation level and roll back point

COMMIT command permanently save the transaction in to database.

- It's syntax is: Commit;

ROLL BACK command uses the save point command to jump to save point in transaction.

- It' s syntax is: rollback to name-save point;

SAVE POINT command is used to save a transaction temporarily.

- It's syntax is: Save point name-save point;

Database Users

Database administrators – DBA is responsible for authorizing access to the database, for coordinating and monitoring its use, and acquiring software and hardware resources as needed.

Database designers – identify data to be stored in the database and choosing appropriate structures to represent and store the data. Most of these functions are done before the database is implemented and populated with the data. It is the responsibility of the database designers to communicate with all prospective users to understand their requirements and come up with a design that meets these requirements. Database designers interact with all potential users and develop views of the database that meet the data and processing requirements of these groups. The final database must support the requirements of all user groups.

End Users

- **Casual End Users** – occasionally access, may need different information each time. Use query language to specify requests.
- **Naïve or parametric end users** – main job is to query and update the database using standard queries and updates. These canned transactions have been carefully programmed and tested. Examples?
- **Sophisticated end users** – engineers, scientists, analysts who implement applications to meet their requirements.
- **Stand alone users** – maintain personal databases using ready made packages.

DBA

A database administrator's (DBA) primary job is to ensure that data is available, protected from loss and corruption, and easily accessible as needed. Below are some of the chief responsibilities that make up the day-to-day work of a DBA. DSP deliver an outsourced DBA service in the UK, providing Oracle Support and SQL Server Support; whilst mindset and toolset may be different, whether a database resides on-premise or in a Public / Private Cloud, the role of the DBA is not that different.

1. Software installation and Maintenance

A DBA often collaborates on the initial installation and configuration of a new Oracle, SQL Server etc database. The system administrator sets up hardware and deploys the operating system for the database server, then the DBA installs the database software and configures it for use. As updates and patches are required, the DBA handles this on-going maintenance. And if a new server is needed, the DBA handles the transfer of data from the existing system to the new platform.

2. Data Extraction, Transformation, and Loading

Known as ETL, data extraction, transformation, and loading refers to efficiently importing large volumes of data that have been extracted from multiple systems into a data warehouse environment. This external data is cleaned up and transformed to fit the desired format so that it can be imported into a central repository.

3. Specialized Data Handling

Today's databases can be massive and may contain unstructured data types such as images, documents, or sound and video files. Managing a very large database (VLDB) may require higher-level skills and additional monitoring and tuning to maintain efficiency.

4. Database Backup and Recovery

DBAs create backup and recovery plans and procedures based on industry best practices, then make sure that the necessary steps are followed. Backups cost time and money, so the DBA may have to persuade management to take necessary precautions to preserve data.

System admins or other personnel may actually create the backups, but it is the DBA's responsibility to make sure that everything is done on schedule.

In the case of a server failure or other form of data loss, the DBA will use existing backups to restore lost information to the system. Different types of failures may require different recovery strategies, and the DBA must be prepared for any eventuality. With technology change, it is becoming ever more typical for a DBA to backup databases to the cloud, Oracle Cloud for Oracle Databases and MS Azure for SQL Server.

5. Security

A DBA needs to know potential weaknesses of the database software and the company's overall system and work to minimize risks. No system is one hundred per cent immune to attacks, but implementing best practices can minimize risks.In the case of a security breach or irregularity, the DBA can consult audit logs to see who has done what to the data. Audit trails are also important when working with regulated data.

6. Authentication

Setting up employee access is an important aspect of database security. DBAs control who has access and what type of access they are allowed. For instance, a user may have permission to see only certain pieces of information, or they may be denied the ability to make changes to the system.

7. Capacity Planning

The DBA needs to know how large the database currently is and how fast it is growing in order to make predictions about future needs. Storage refers to how much room the database takes up in server and backup space. Capacity refers to usage level. If the company is growing quickly and adding many new users, the DBA will have to create the capacity to handle the extra workload.

8. Performance Monitoring

Monitoring databases for performance issues is part of the on-going system maintenance a DBA performs. If some part of the system is slowing down processing, the DBA may need to make configuration changes to the software or add additional hardware capacity. Many types of monitoring tools are available, and part of the DBA's job is to understand what they need to track to improve the system. 3rd party organizations can be ideal for outsourcing this aspect, but make sure they offer modern DBA support.

9. Database Tuning

Performance monitoring shows where the database should be tweaked to operate as efficiently as possible. The physical configuration, the way the database is indexed, and how queries are handled can all have a dramatic effect on database performance. With effective monitoring, it is possible to proactively tune a system based on application and usage instead of waiting until a problem develops.

10. Troubleshooting

DBAs are on call for troubleshooting in case of any problems. Whether they need to quickly restore lost data or correct an issue to minimise damage, a DBA needs to quickly understand and respond to problems when they occur.

UNIT- 2

Database Design and ER Diagram:

The database design process can be divided into six steps. The ER model is most relevant to the first three steps:

(i) **Requirements Analysis:** The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database.

(ii) **Conceptual Database Design:** The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model, and is discussed in the rest of this chapter.

(iii) **Logical Database Design:** We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will only consider relational DBMS's, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.

Beyond the ER Model

ER modeling is sometimes regarded as a complete approach to designing a logical database schema. This is incorrect because the ER diagram is just an approximate description of the data, constructed through a very subjective evaluation of the information collected during requirements analysis. The remaining three steps of database design are briefly described below:

(iv) **Schema Refinement:** The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory.

(v) **Physical Database Design:** In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps.

(vi) **Security Design:** In this step, we identify different user groups and different **roles** played by various users (e.g., the development team for a product, the customer support representatives, and the product manager). For each role and user group, we must identify the parts of the database that they must be able to access and the parts of the database that they should *not* be allowed to access, and take steps to ensure that they can access only the necessary parts.

Entities, Attributes & Entity Sets:

An ***entity*** is an object that exists and is distinguishable from other objects.

– Example: student, department, employee and branch

Entities have ***attributes,*** which defines the property of an entity

– Example: student has *names* and *roll no.*
– There are different types of attributes which are categorized as follows:

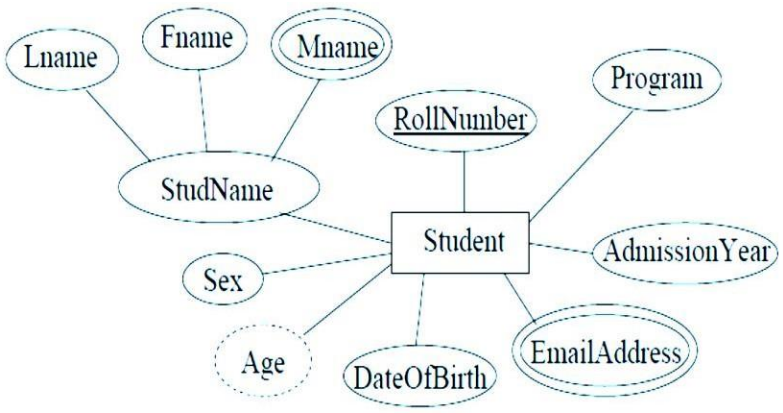
i) **Simple Attributes:** having atomic or indivisible values. For e.g. *Dept*–a string, *PhoneNumber*–an eight digit number.

ii) **Composite Attributes:** having several components in the value. For e.g.: *Qualification* with components (*DegreeName, Year, UniversityName*).

iii) **Derived Attributes:** Attribute value is dependent on some other attribute. For e.g.:*Age* depends on *DateOfBirth*. So age is a derived attribute.

iv) **Single-valued Attributes:** having only one value rather than a set of values.
For e.g. *PlaceOfBirth*–single string value.

v) **Multi-valued Attributes:** having a set of values rather than a single value.
For



e.g., C*oursesEnrolled*attribute for student, *EmailAddress* attribute for student, *PreviousDegree* attribute for student.

Diagrammatic representation of an Entity and different types of attributes:

i) ***entity* -rectangle *attribute*** – ellipse connected to rectangle
ii) ***multi-valued attribute*** – double ellipse
iii) ***composite attribute*** – ellipse connected to ellipse
iv) ***derived attribute*** – dashed ellipse

Domains of Attributes

Each attribute takes values from a set called its *domain.*

For example,

studentAge – {17,18, …, 55}

HomeAddress–character strings of length 35.

Domain of composite attributes –cross product of domains of component attributes.

Domain of multi-valued attributes –set of subsets of values from the basic domain

An ***entity set*** is a set of entities of the same type that share the same properties.

– Example: set of all students, employees etc.

Relationships and Relationship sets

When two or more entities are associated with each other, we have an instance of a *Relationship*.

E.g.: *student* Ramesh *enrolls* in Discrete Mathematics *course*
Relationship *enrolls* has *Student* and *Course* as the *participating* entity sets.

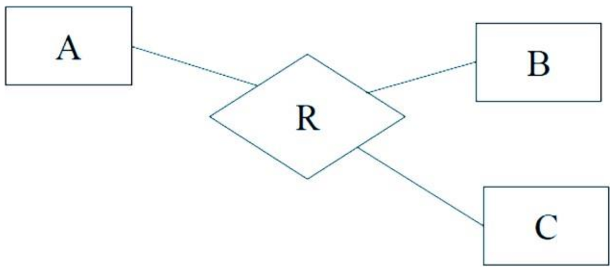
Degree of a relationship

Degree: The number of participating entities.

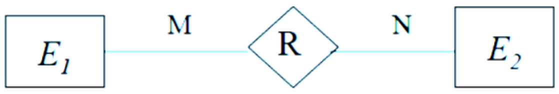
• Degree 2: A relationship having 2 entities attached, it is called *binary relationship*.
• Degree 3: A relationship having 3 entities attached, it is called *ternary relationship*
• Degree n: A relationship having 2 entities attached, it is called *n-ary relationship*
• Binary relationships are very common and widely used.

Diagrammatic Notation for Relationships

Relationship is represented using a diamond shaped box. Rectangle of each participating entity is connected by a line to this diamond. Name of the relationship is written in the box

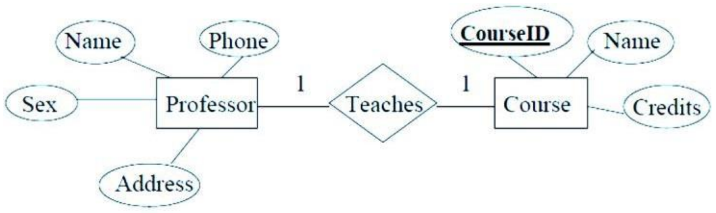

A
R
B
C

Binary Relationships and Cardinality Ratio

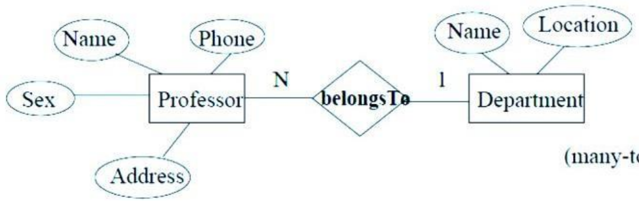


Cardinality Ratios

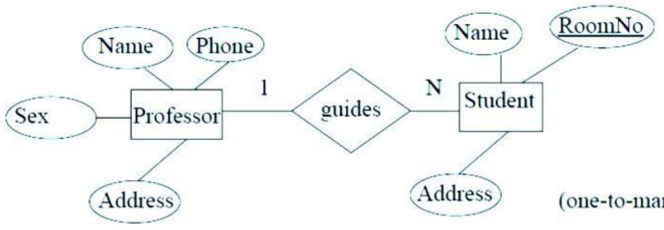
- *One-to-One:* An *E1* entity may be associated with at most one *E2* entity and similarly an *E2* entity may be associated with at most one *E1* entity.



- *One-to-Many & Many-to-One:* An *E1* entity may be associated with many *E2* entities whereas an *E2* entity may be associated with at most one *E*1 entity.

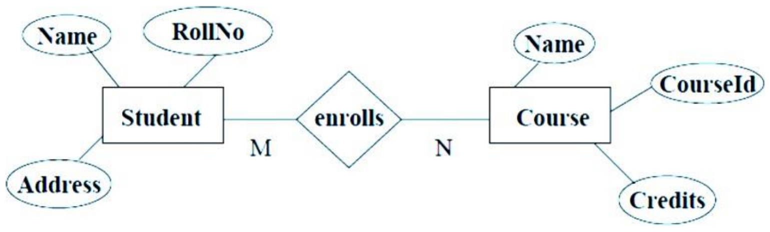


(many-to-one)



(one-to-many)

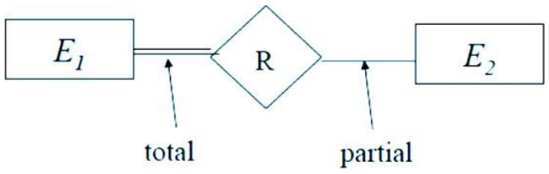
- *Many-to-Many:* Many *E1* entities may be associated with a single *E2* entity and a single *E1* entity may be associated with many *E2* entities



Participation Constraints

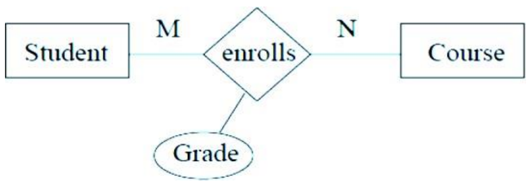
An entity set may participate in a relation either *totally*or *partially*.

- *Total participation*: Every entity in the set is involved in some association (or tuple) of the relationship.
- *Partial participation*: Not all entities in the set are involved in association (or tuples) of the relationship.



Attributes for Relationship Types

Relationship types can also have attributes.



Grade gives the letter grade (S,A,B, etc.) earned by the student for a course. It is neither an attribute of *student* nor that of *course*.

<u>Design Issues of ER model:</u>

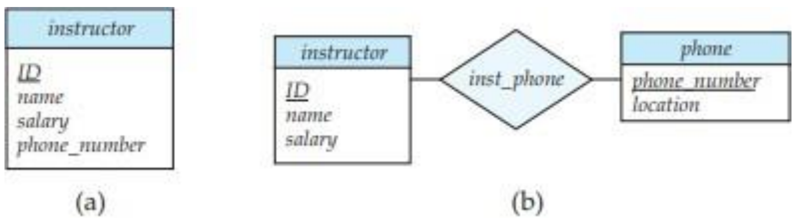
The notions of an entity set and a relationship set are not precise, and it is possible to define a set of entities and the relationships among them in a number of different ways.

Use of Entity Sets versus Attributes

Consider the entity set instructor with the additional attribute phone number It can easily be argued that a phone is an entity in its own right with attributes phone number and location; the location may be the office or home where the phone is located, with mobile (cell) phones perhaps represented by the value "mobile." If we take this point of view, we do not add the attribute phone number to the instructor. Rather, we create:

• A phone entity set with attributes phone number and location.

• A relationship set inst phone, denoting the association between instructors and the phones that they have.



Use of Entity Sets versus Relationship Sets

It is not always clear whether an object is best expressed by an entity set or a relationship set.

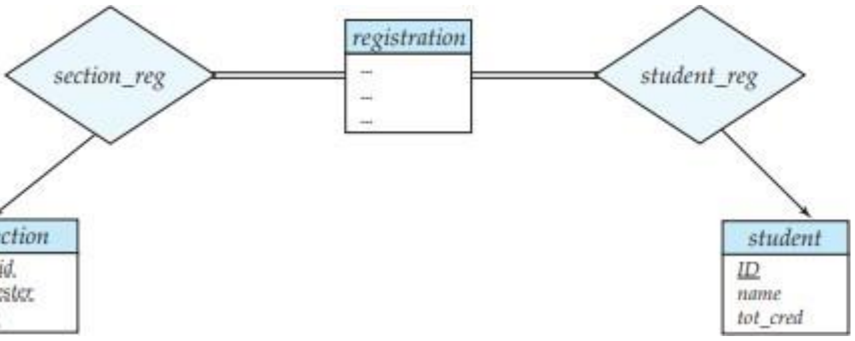
we used the takes relationship set to model the situation where a student takes a (section of a) course. An alternative is to imagine that there is a course-registration record for each course that each student takes. Then, we have an entity set to represent the course-registration record. Let us call that entity set registration. Each registration entity is related to exactly one student and to exactly one section, so we have two relationship sets, one to relate courseregistration records to students and one to relate course-registration records to sections. we show the entity sets section and studentfrom with the takes relationship set replaced by one entity set and two relationship sets:

• registration, the entity set representing course-registration records.

• section reg, the relationship set relating registration and course.

• student reg, the relationship set relating registration and student.

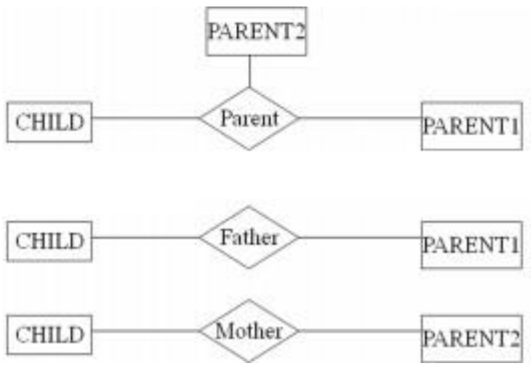
Note that we use double lines to indicate total participation by registration entities.



One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships

Binary vs. n-ary relationship sets

Relationships in databases are often binary. Some relationships that appear to be non-binary could actually be better represented by several binary relationships



It is always possible to replace a non-binary relationship set by a number of distinct binary relationship sets. For example, consider a ternary relationship R associated with three entity sets A, B and C. We can replace the relationship set R by an entity set E and create three relationship sets as:

• RA, relating E and A

• RB, relating E and B

• RC, relating E and C

If the relationship set R had any attributes, these are assigned to entity set E. A special identifying attribute is created for E

Placement of Relationship Attributes

The cardinality ratio of a relationship can affect the placement of relationship attributes:

• One-to-Many: Attributes of 1:M relationship set can be repositioned to only the entity set on the many side of the relationship

• One-to-One: The relationship attribute can be associated with either one of the participating entities

• Many-to-Many: Here, the relationship attributes can not be represented to the entity sets; rather they will be represented by the entity set to be created for the relationship set.

constraints/key constraints:-

Entity integrity constraints / key constraints specifies the condition and restricts the data that can be stored, only in one table

Definitions :

Key constraint: a key constraint is a statement that a certain minimal fields of a relation has a unique identifier for all tuples. Actually key constraint is the general term, the term candidate key is used for satisfying the constraints according to a key constraints.

Candidate key: A set of fields that uniquely identifies a tuple according to a key constraint is called a candidate key for the relation

Super key: A super key is a super set of a candidate key. A super key is a set of fields that each contains a candidate key

Specifying key constraint in sql:

In sql , we can declare that columns of a table from a candidate key using two statements.

- ➔ Unique key
- ➔ Primary key

Example:

Sql>create table student(sid char(10) primary key ,name varchar(20),

login char(10) UNIQUE, age int ,GPA float);

UNIQUE Key:- The purpose of a UNIQUE Key is to ensure that the information in the column is UNIQUE,i.e

- ✓ The data held across the column must be UNIQUE.
- ✓ Any column can be left blank eith NULL value.

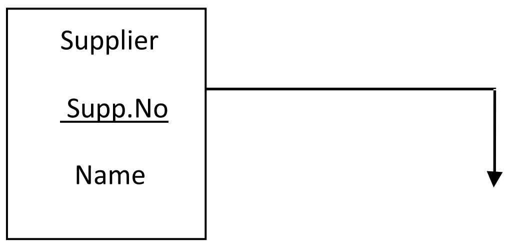
PRIMARY Key:- The purpose of a PRIMARY Key is to ensure that information in the column is UNIQUE and must be compulsory entered.i.e

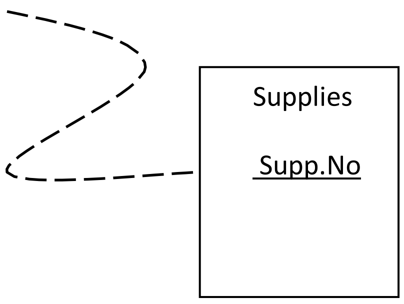
- ✓ The data held across the column must be UNIQUE.
- ✓ Even one column also can't be left blank.

➔ Referentional Integrity constraints/foreign key constraints:-

Referentional Integrity constraints checks the conditions to satisfied in more than one relation(Two or more tables)connected with some relationship.

Let us take an ex.





If a relation *supplies* includes a foreign key(*SUPP No*) matching a Primary key(*Supplier.Supp No*) in some other relation(*Supplier*), then every value of the foreign key in that relation(*Supplies*) must be equal to the primary key of relation(*Supplier*).

Specifying foreign key constraints in sql:

In SQL, we can declare that columns of a table form a referential integrity constraint using *Foreign key.*

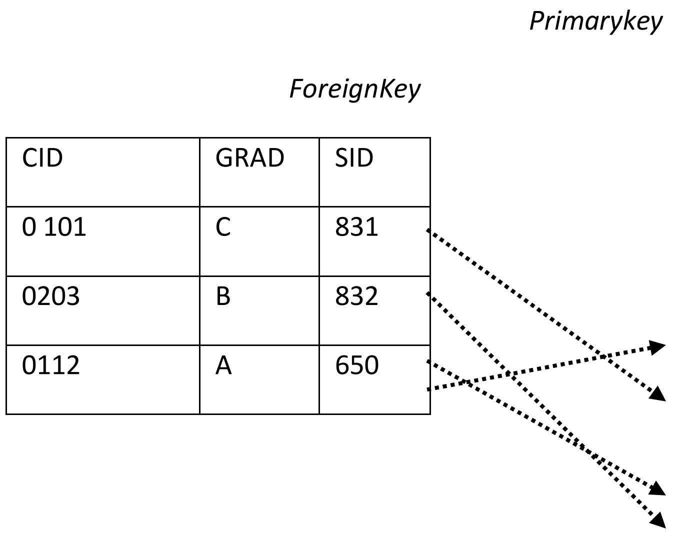
Foreign keys represent relationship between tables. A foreign key is a column whose values are derived from the primary key or UNIQUE Key of some other table.

The table in which the foreign key is defined is called a foreign table or detail table. The table that defines the primary key and is referenced by the foreign key is called the primary or Master Table.

Let us create the new table "*Enrolled*" connected with the previous example S*tudent* table.

Sql>create table enrolled(sid varchar(10), cid varchar(10) primary key, grade varchar(10),

foreignkey (sid) references students);



CID	GRAD	SID
0 101	C	831
0203	B	832
0112	A	650

0105	B	666

Enrolled

Sid	Name	Login	Age	gpa
000	Ghj	Ghj@cs	18	3.6
666	Abc	Abc@cs	19	3.3
688	Sad	Sad@db	20	4.6
831	Bad	Bad@se	21	4.8
650	Good	Good@db	18	3.1
832	xyz	xyz@cs	19	3.8

Students

Introduction to relational model:

The relational model was introduced by Dr.E.F.Codd in 1970.The relational model represents data in the form of two dimensional tables. The organization of data into relational tables is known as the logical view of the database.

Characteristics of Relational Model:

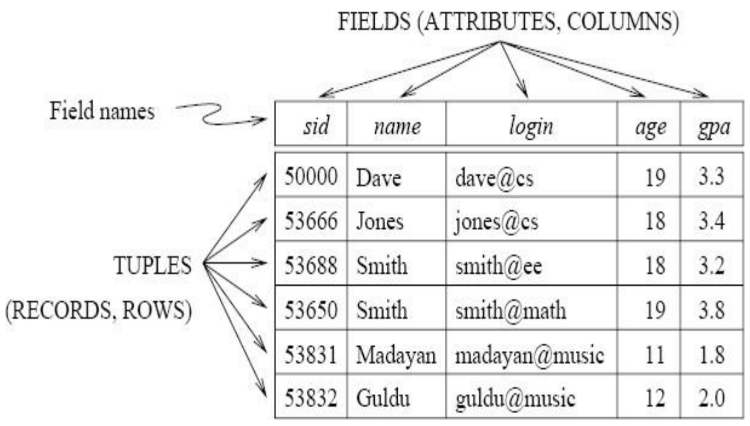
- A relational table eliminates all parent child relationships or instead represented all data in the database as sample row/column tables of data values
- A relation as similar to a table with rows/columns of data values
- Each table as an independent entry and there as no physical relationships between tables
- Relational model of data management is based on set theory
- The user interface used with relational models as non procedural because only what needs to be done as specified and not how it has to be done

Fundamental concepts of relations:

Relation:- A Relation can be thought of as a set of records in the form of two-dimensional table containing rows and columns of data

A relation consists of two things: a relation schema and instance relation.

Relation schema: The relation schema contains the basic information of a table. This information includes the name of the table, the names of the columns and the data types associated with each column



sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Cardinality

Degree

Relation instance: An instance of a relation is a set of types in which each tuple has the same no. of fields as the relation schema

Relational database schema: A relational database schema is a collection of relation schemas, describing one or more relations

Relation cardinality: The Relation cardinality is the no. of tuples in the relation

Relation degree: The Relation degree as the no.of columns in the relation

Tuples/records: The rows of the table as also known as records or tuples

Field/attributes: The columns of the table as also known as fields/attributes

Tabular Representation of Various ER Schemas

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*.

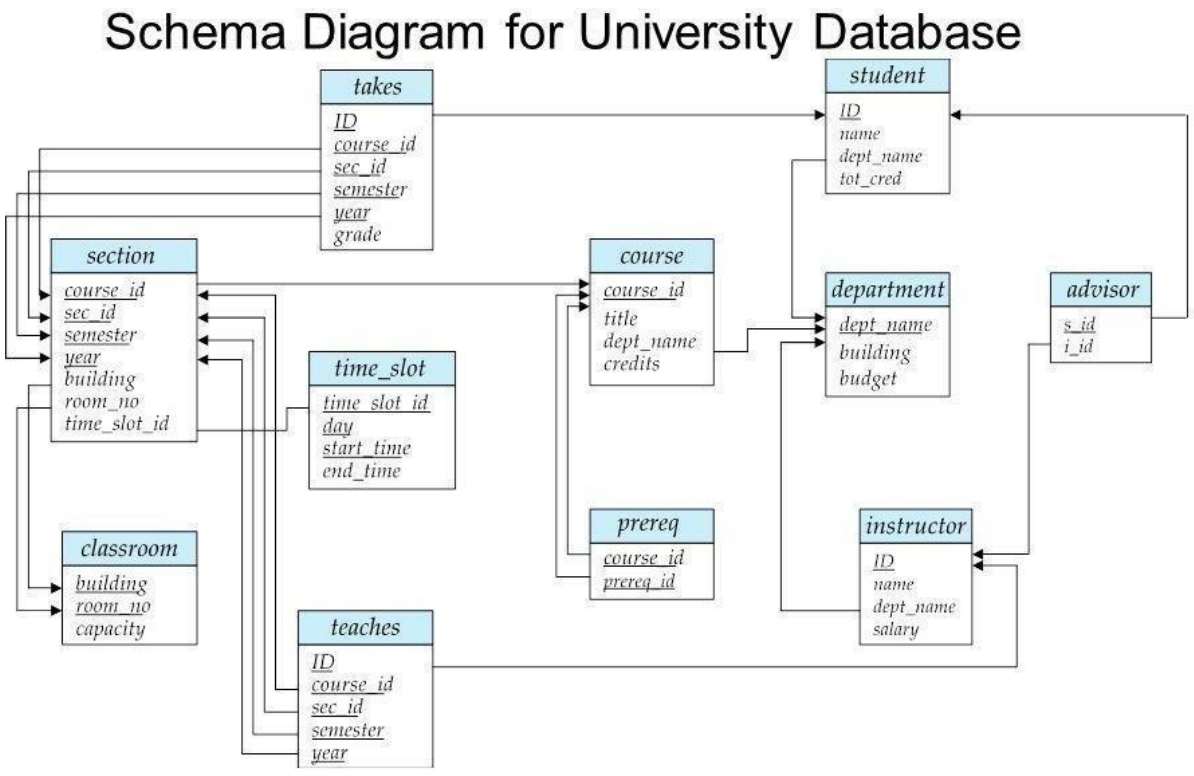
Database Schema

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.

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.

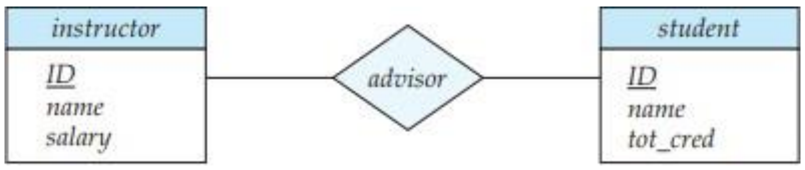


Schema diagram for the university database.

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.

ER Diagram Notations

An E-R diagram consists of the following major components:



• Rectangles divided into two parts represent entity sets. The first part, which in this textbook is shaded blue, contains the name of the entity set. The second part contains the names of all the attributes of the entity set.

• Diamonds represent relationship sets.

• Undivided rectangles represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.

• Lines link entity sets to relationship sets.

• Dashed lines link attributes of a relationship set to the relationship set.

• Double lines indicate total participation of an entity in a relationship set.

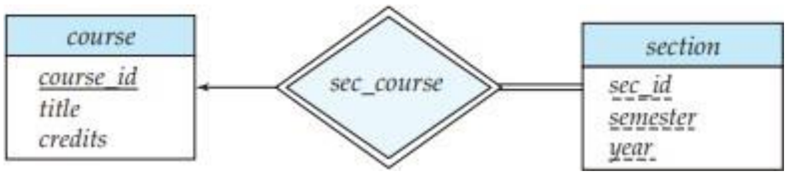
• Double diamonds represent identifying relationship sets linked to weak entity sets

Weak Entity Set-

Consider a section entity, which is uniquely identified by a course identifier, semester, year, and section identifier. Clearly, section entities are related to course entities. Suppose we create a relationship set sec course between entity sets section and course. Now, observe that the information in sec course is redundant, since section already has an attribute course id, which identifies the course with which the section is related. One option to deal with this redundancy is to get rid of the relationship sec course; however, by doing so the relationship between section and course becomes implicit in an attribute, which is not desirable.

The notion of weak entity set formalizes the above intuition. An entity set that does not have sufficient attributes to form a primary key is termed a weak entity set. An entity set that has a primary key is termed a strong entity set.

For a weak entity set to be meaningful, it must be associated with another entity set, called the identifying or owner entity set. Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be existence dependent on the identifying entity set. The identifying entity set is said to own the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the identifying relationship



had a primary key. However, conceptually, a section is still dependent on a course for its existence, which is made explicit by making it a weak entity set.

In E-R diagrams, a weak entity set is depicted via a rectangle, like a strong entity set, but there are two main differences:

• The discriminator of a weak entity is underlined with a dashed, rather than a solid, line.

• The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.

Views

Introduction to views

- The dynamic result of one or more relational operations operating on the base relations to produce another relation is called view. A view is a virtual relation that does not necessarily exist in the database. But can be produced upon request by a particular user at the time of request.
- A view is object that gives the user a logical view of data from an underlying table or tables. You can restrict what users can view by allowing them to see only a few columns from a table.

Purpose of views

- The view mechanism is desirable for several reasons.
- It simplifies queries.
- It can be queried as a base table.
- It provides a powerful and flexible security mechanism by hiding parts of the database from certain users.
- It permits users to access data in a way that is customized to their needs, so that the same data can be seen by different users in different ways at the same time.

Updating views

- All updates to a base relation should be immediately reflected in all views that a single base relation and containing either the primary key or a candidate key of the base relation.
- Updates are not allowed through views involving multiple base relations.
- Updates are not allowed through views involving aggregation or grouping operations.

Creating views

Syntax:

```
CREATE VIEW  viewname as
 SELECT columnname,cloumnname
 FROM tablenmae
 WHERE columnname=expression list;
```

Examples: create view on book table which contains two fields title, and author name.

```
SQL> create view V_book as select title, author_name from book;

View created.

SQL>select * from V_book;
```

Output:

Title	**Author_name**
Oracle	Arora
DBMS	Basu
DOS	Sinha
ADBMS	Basu
Unix	Kapoor

Selecting Data from a view

Example: Display all the title of book written by author 'Basu'.

SQL> select title from V_Book Where author_name='Basu';

Output:

Title
DBMS
ADBMS

Updatable Views

Views can also be used for data manipulation i.e., the user can perform Insert, Update and the Delete operations on the view. The views on which data manipulation can be done are called Updatable views, views that do not allow data manipulation are called Read only Views. When you give a view name in the update, insert or delete statement, the modification to the data will be passed to the underlying table.

For the view to be updatable, it should meet following criteria:

- The view must be created on a single table.
- The primary key column of the table should be included in the view.
- Aggregate functions cannot be used in the select statement.
- The select statement used for creating a view should not included Distinct, Group by or Having clause.
- The select statement used for creating a view should not include sub queries.
- It must not use constant, string or values expression like total/5.

Destroying/Altering Tables and Views

Altering Table

The definition of the table is changed using ALTER TABLE statement. The ALTER TABLE is used to add, delete or modify columns in an existing table explained below:

1) **ALTER TABLE…..ADD…..**
   This is used to add some extra columns into an existing table. The generalized format is given below.

```
ALTER TABLE relation_name

ADD(new field1 datatype(size),

new field2 datatype(size)'………
new fieldn datatype(size));
```

Example:

ADD customer phone and fax number in the customer relation.

SQL> ALTER TABLE Customer

ADD(cust_ph_no varchar(15),cust_fax_no varchar(15));

Table created.

2) ALTER TABLE …..MODIFY

This form is used to change the width as well as data type of existing relations. the generalized syntax of this from is shown below.

ALTER TABLE relation_name MODIFY(field$_1$ new data type(size), field$_2$ new data type(size), ... field$_n$ new data type(size));

- **Example:**

Modify the data type of the publication year as numeric data type.
SQL> ALTER TABLE Book
MODIFY(pub_year number(4));
Table created.

Restrictions of the Alter Table

Using the alter table clause you perform the following tasks:
Change the name of the table.
Change the name of the column.
Drop a column.
Decrease the size of a column if table data exists.

3) ALETR TABLE……DELETE

To delete a column in a table , use the following syntax:

```
ALTER TABLE table_name

DROP COLUMN column_name
```

- **Example**: Drop customer fax number from the customer table.

SQL> ALTER TABLE customer

DROP COLUMN cust_fax_no;

DELETING TABLE

The tables are deleted permanently from the database using DROP TABLE command. We remove all the data from the table using TRUNCATE TABLE command. It is explained below:

1) DROP TABLE

This command is used to delete a table. The generalized syntax if this form is given below:

```
DROP TABLE relation_name
```

- **Example**: write the command for deleting special_customer relation.

SQL DROP TABLE Special_customer;

TABLR dropped.

2) Truncate a table

Truncating a table is removing all records from the table. The structure of the table stays intact. The SQL language has a DELETE statement which can be used to remove one or more (or all) rows from a table. Truncation releases storage space occupied by the table, but deletion does not. The **syntax:**

```
TRUNCATE TABLE table_name;
```

- **Example:**

SQL> TRUNCATE TABLE student;

Deleting view

A view can be dropped by using the DROP VIEW command.

Syntax:

```
DROP VIEW viewname;
```

- **Example:**

DROP VIEW V_Book;

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     /*action*/

             Count:=0;

     END
```

```
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
  - BEGIN
        count := 0;
    END
```

Unit 3

SQL: Overview, The Form of Basic SQL Query -UNION, INTERSECT, and EXCEPT– join operations: equi join and non equi join-Nested queries - correlated and uncorrelated- Aggregate Functions-Null values, GROUPBY- HAVING Clause.

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
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

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
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

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’
```

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:

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

Thus $\bowtie$ 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 $S1 \bowtie_{S1.sid<R1.sid} R1$

Equijoin

A common special case of the join operation $R \bowtie S$ is when the *join condition* consists solely of equalities (connected by $\wedge$) 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 redundancy in retaining both attributes in the result.

Natural Join

A further special case of the join operation $R \bowtie S$ 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.

Non Equi Join

The SQL NON EQUI JOIN uses comparison operator instead of the equal sign like **>, <, >=, <=** along with conditions.

```
SELECT *

FROM table_name1, table_name2

WHERE table_name1.column [> | < | >= | <= ] table_name2.column;
```

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
FROM   Sailors 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' )
```

Noncorrelated

There are two kind of subquery in SQL one is called non-correlated and other is called correlated subquery. In non correlated subquery, **inner query doesn't depend on outer query** and can run as stand alone query.Subquery used along-with IN or NOT IN sql clause is good examples of Noncorrelated subquery in SQL. Let's a **noncorrelated subquery example** to understand it better

NonCorrelated subquery are used along-with IN and NOT IN clause. here is an example of subquery with IN clause in SQL.

SQL query: Find all stocks from United States and India

```
mysql> SELECT COMPANY FROM Stock WHERE LISTED_ON_EXCHANGE IN (SELECT RIC FRO
M Market WHERE COUNTRY='United States' OR COUNTRY= 'INDIA');
+-------------------------+
| COMPANY                 |
+-------------------------+
| Google Inc              |
| Goldman Sachs GROUP Inc |
| InfoSys                 |
+-------------------------
```

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)

FROM    Sailors S
```

Q3) Count the number of sailors.

```
SELECT COUNT (*)
FROM    Sailors S
```

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.

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
```

UNIT 4

DEPENDENCIES AND NORMAL FORMS

Normalization – Introduction, functional dependencies, First, Second, and third normal forms – dependency preservation, Boyce/Codd normal form.

Higher Normal Forms - Introduction, Multi-valued dependencies and Fourth normal form, Join dependencies and Fifth normal form

NORMALIZATION

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.

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.

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.

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.

Normalization:

It is a process for evaluating and correcting table structures to minimize data redundancies, there by reducing the likelihood of data anomalies.

Normalization works through a series of stages called normal forms. The first three stages are described as first normal form (1NF), second normal form (2NF) and third normal form (3NF).

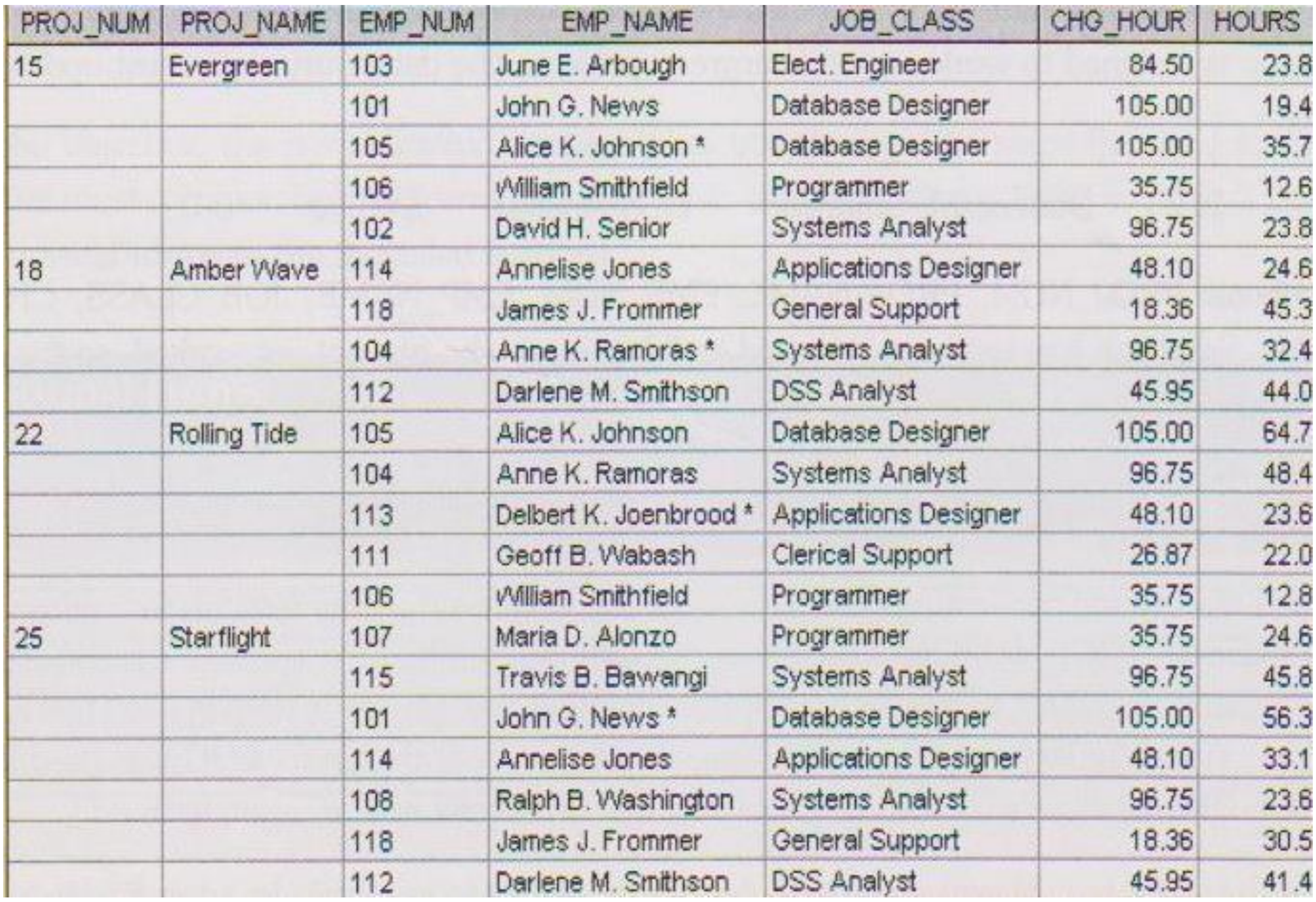
From a structural point of view, 2NF is better than 1NF and 3NF is better than 2NF.

Denormalization:

Produces a lower normal form, which is a 3NF will be converted to a 2NF through denormalization. A successful design must also consider end-user demand for fast performance. Therefore, you will occasionally be expected to denormalize some portions of database design in order to meet performance requirements.

The need for normalization

In following example:



PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June E. Arbough	Elect. Engineer	84.50	23.8
		101	John G. News	Database Designer	105.00	19.4
		105	Alice K. Johnson *	Database Designer	105.00	35.7
		106	William Smithfield	Programmer	35.75	12.6
		102	David H. Senior	Systems Analyst	96.75	23.8
18	Amber Wave	114	Annelise Jones	Applications Designer	48.10	24.6
		118	James J. Frommer	General Support	18.36	45.3
		104	Anne K. Ramoras *	Systems Analyst	96.75	32.4
		112	Darlene M. Smithson	DSS Analyst	45.95	44.0
22	Rolling Tide	105	Alice K. Johnson	Database Designer	105.00	64.7
		104	Anne K. Ramoras	Systems Analyst	96.75	48.4
		113	Delbert K. Joenbrood *	Applications Designer	48.10	23.6
		111	Geoff B. Wabash	Clerical Support	26.87	22.0
		106	William Smithfield	Programmer	35.75	12.8
25	Starflight	107	Maria D. Alonzo	Programmer	35.75	24.6
		115	Travis B. Bawangi	Systems Analyst	96.75	45.8
		101	John G. News *	Database Designer	105.00	56.3
		114	Annelise Jones	Applications Designer	48.10	33.1
		108	Ralph B. Washington	Systems Analyst	96.75	23.6
		118	James J. Frommer	General Support	18.36	30.5
		112	Darlene M. Smithson	DSS Analyst	45.95	41.4

We see in that example, the structure of data set does not conform to the requirements of table nor does it handle data very well.

Consider the following deficiencies:

1. The project number (PROJ_NUM) is apparently intended to be primary key or at least a part of a PK, but it contains nulls.
2. The table entries invite data inconsistencies. For example the JOB_CLASS value "Elect. Engineer" might be entered as "Elect. Eng."
3. The table displays data redundancies. Those data redundancies yield the following anomalies:
   a. Update anomalies. Modifying the JOB_CLASS for employee number 105 requires (potentially) many alterations, one for each EMP_NUM=105.
   b. Insertion anomalies. Just to complete a row definition, a new employee must be assigned to a project. If the employee is not assigned, a phantom project must be created to complete the employee data entry.

c. Deletion anomalies. Suppose that only one employee is associated with a given project, if that employee leaves the company and the employee data are deleted , the project information will also be deleted .to prevent the loss of the project information ,a fictitious employee must be created just to save the project information.

The Normalization Process:

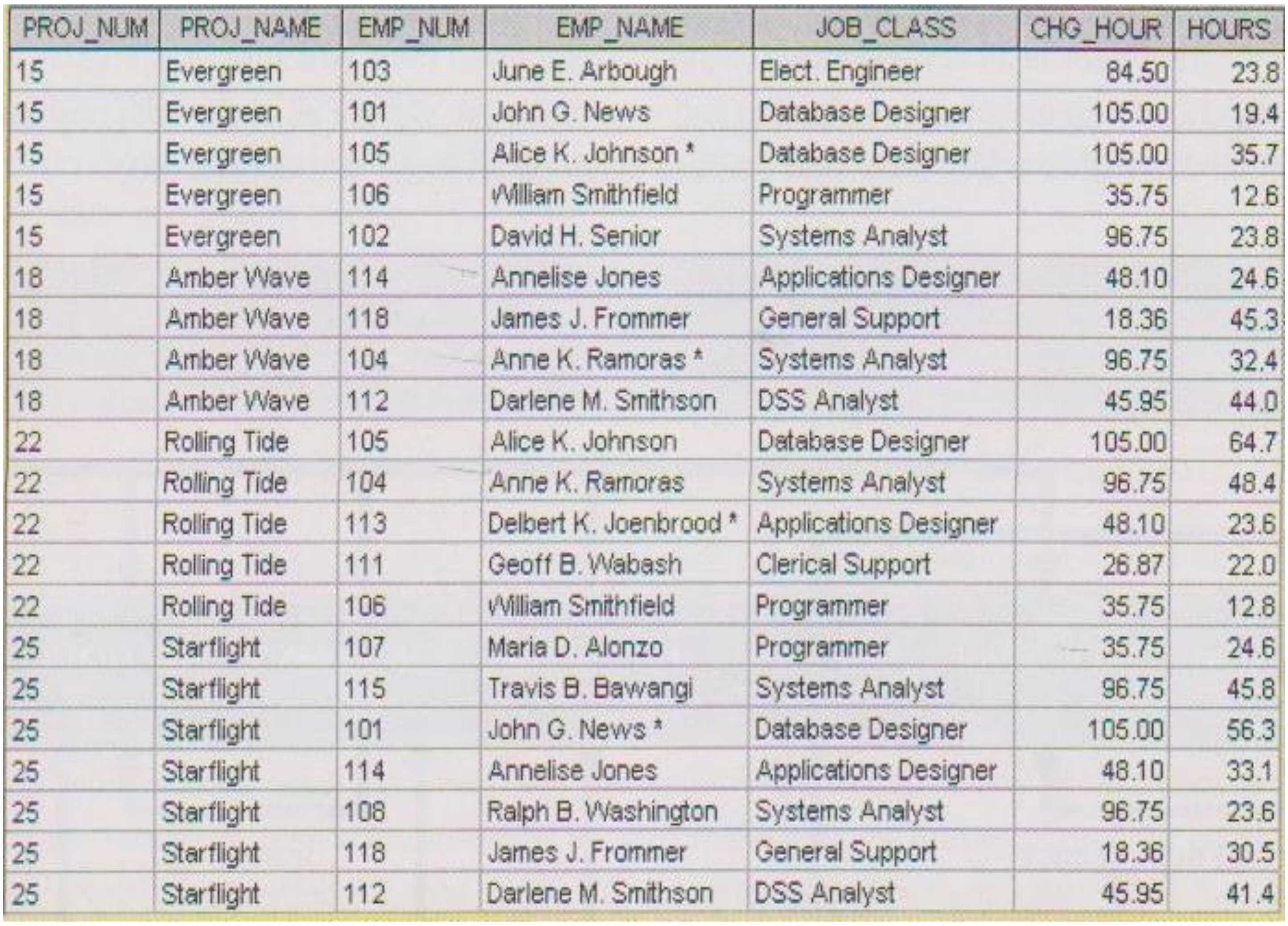
We will learn how to use normalization to produce a set of normalized tables to store the data that will be used to generate the required information. The objective of normalization is to ensure that each table conforms to the concept of well-formed relations, that is, tables that have the following characteristics:

- Each table represents a single subject. For example, a course Table will contain only data that directly pertains to courses. Similarly, a student table will contain only student data.
- No data item will be unnecessarily stored in more than one table (in short, tables have minimum controlled redundancy). The reason for this requirement is to ensure that the data are update in only one place.
- All nonprime attributes in a table are dependent on the primary key. The reason for this requirement is to ensure that the data are uniquely identifiable by a primary key value.
- Each table is void of insertion, update or deletion anomalies. This is to ensure the integrity and consistency of the data.

Conversion to First Normal Form (1NF)

Step 1: Eliminate the Repeating Groups

Start by presenting the data in tabular format, where each cell has a single value and there are no repeating groups. A **repeating group** derives its name from the fact that a group of multiple entries of the same type can exist for any single key attributes occurrence. To eliminate the repeating groups, eliminate the nulls by making sure that each repeating group attribute contains an appropriate data value.



PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June E. Arbough	Elect. Engineer	84.50	23.8
15	Evergreen	101	John G. News	Database Designer	105.00	19.4
15	Evergreen	105	Alice K. Johnson *	Database Designer	105.00	35.7
15	Evergreen	106	William Smithfield	Programmer	35.75	12.6
15	Evergreen	102	David H. Senior	Systems Analyst	96.75	23.8
18	Amber Wave	114	Annelise Jones	Applications Designer	48.10	24.6
18	Amber Wave	118	James J. Frommer	General Support	18.36	45.3
18	Amber Wave	104	Anne K. Ramoras *	Systems Analyst	96.75	32.4
18	Amber Wave	112	Darlene M. Smithson	DSS Analyst	45.95	44.0
22	Rolling Tide	105	Alice K. Johnson	Database Designer	105.00	64.7
22	Rolling Tide	104	Anne K. Ramoras	Systems Analyst	96.75	48.4
22	Rolling Tide	113	Delbert K. Joenbrood *	Applications Designer	48.10	23.6
22	Rolling Tide	111	Geoff B. Wabash	Clerical Support	26.87	22.0
22	Rolling Tide	106	William Smithfield	Programmer	35.75	12.8
25	Starflight	107	Maria D. Alonzo	Programmer	35.75	24.6
25	Starflight	115	Travis B. Bawangi	Systems Analyst	96.75	45.8
25	Starflight	101	John G. News *	Database Designer	105.00	56.3
25	Starflight	114	Annelise Jones	Applications Designer	48.10	33.1
25	Starflight	108	Ralph B. Washington	Systems Analyst	96.75	23.6
25	Starflight	118	James J. Frommer	General Support	18.36	30.5
25	Starflight	112	Darlene M. Smithson	DSS Analyst	45.95	41.4

Step 2: Identify the primary key:

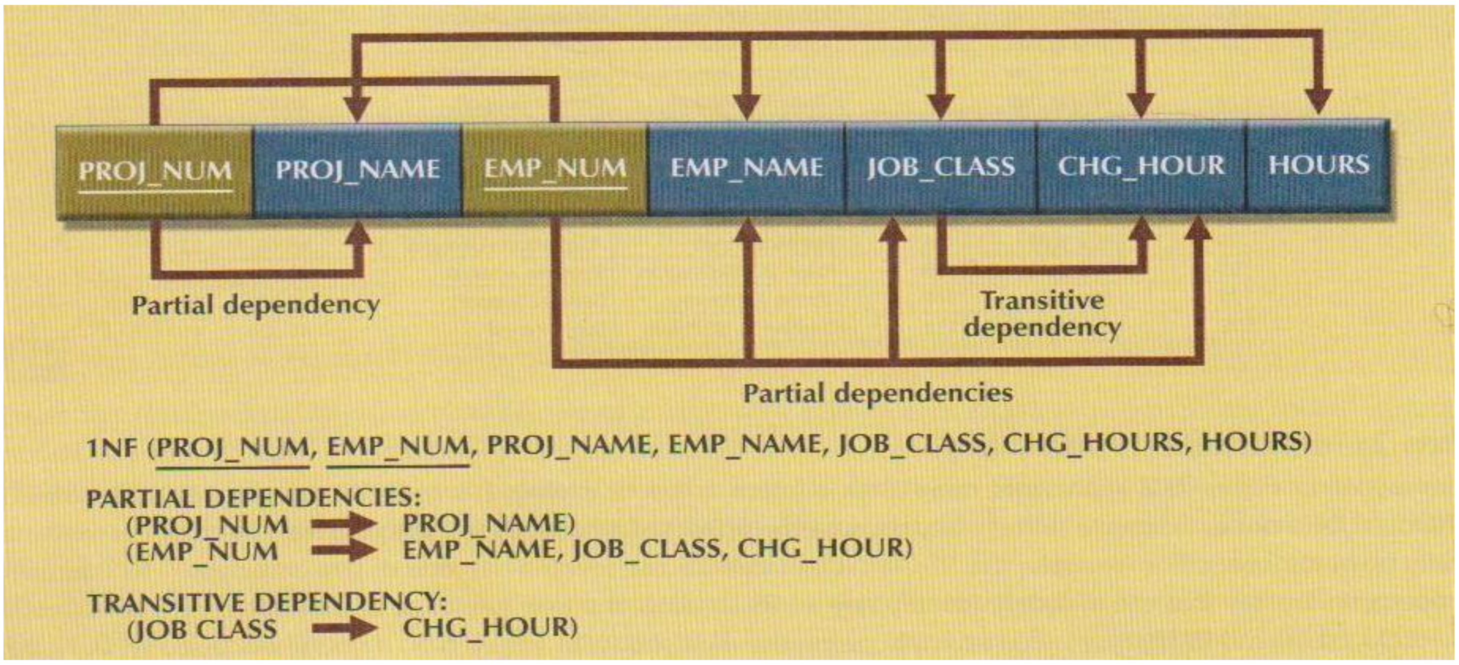
Even causal observers will not that PROJ-NUM is not an adequate primary key because the project number does not uniquely identify all of the remaining entity (row) attributes. To maintain a proper primary key that will uniquely identify any attribute value, the new key must be compost of a combination of a PROJ_NUM and EMP_NUM

Step 3: Identify All Dependencies:

The identification of the PK in Step 2 means that you have already identified the following dependency:

- PROJ_NUM, EMP_NUM → PROJ_NAME, EMP_NAME, JOB_CLASS, CHG_HOUR, HOURS

- PROJ_NUM → PROJ_NAME
- EMP_NUM → EMP-NAME, JOB-CLASS, CHG-HOUR
- JOB_CLASS → CHG_HOUR



Partial dependency a dependency based only a part of a composite primary key.

Transitive dependency is a dependency of one nonprime attribute on another nonprime attribute.

The term first normal form (1NF) describes the tabular format in which:

- All of the key attributes are defined.
- There are no repeating groups in the table. in other words, each row/column intersection contains one and only one value, not a set of values.
- All attributes are dependent on the primary key.

The problem with the 1NF table structure is that it contains partial dependencies. While partial dependencies are sometimes used for performance reasons, they should be used with caution.

Conversion to Second Normal Form (2NF)

Converting to 2NF is done only when the 1NF has a composite primary key. if the 1NF has a single attribute primary key, then the table is automatically in 2NF. The 1NF-to-2NF conversion is simple starting with:

Step 1: Write Each Key Component on a Separate Line

Write each key component on a separate line; then write the original (composite) key on the last line.

- PROJ_NUM
- EMP_NUM
- PROJ_NUM   EMP_NUM

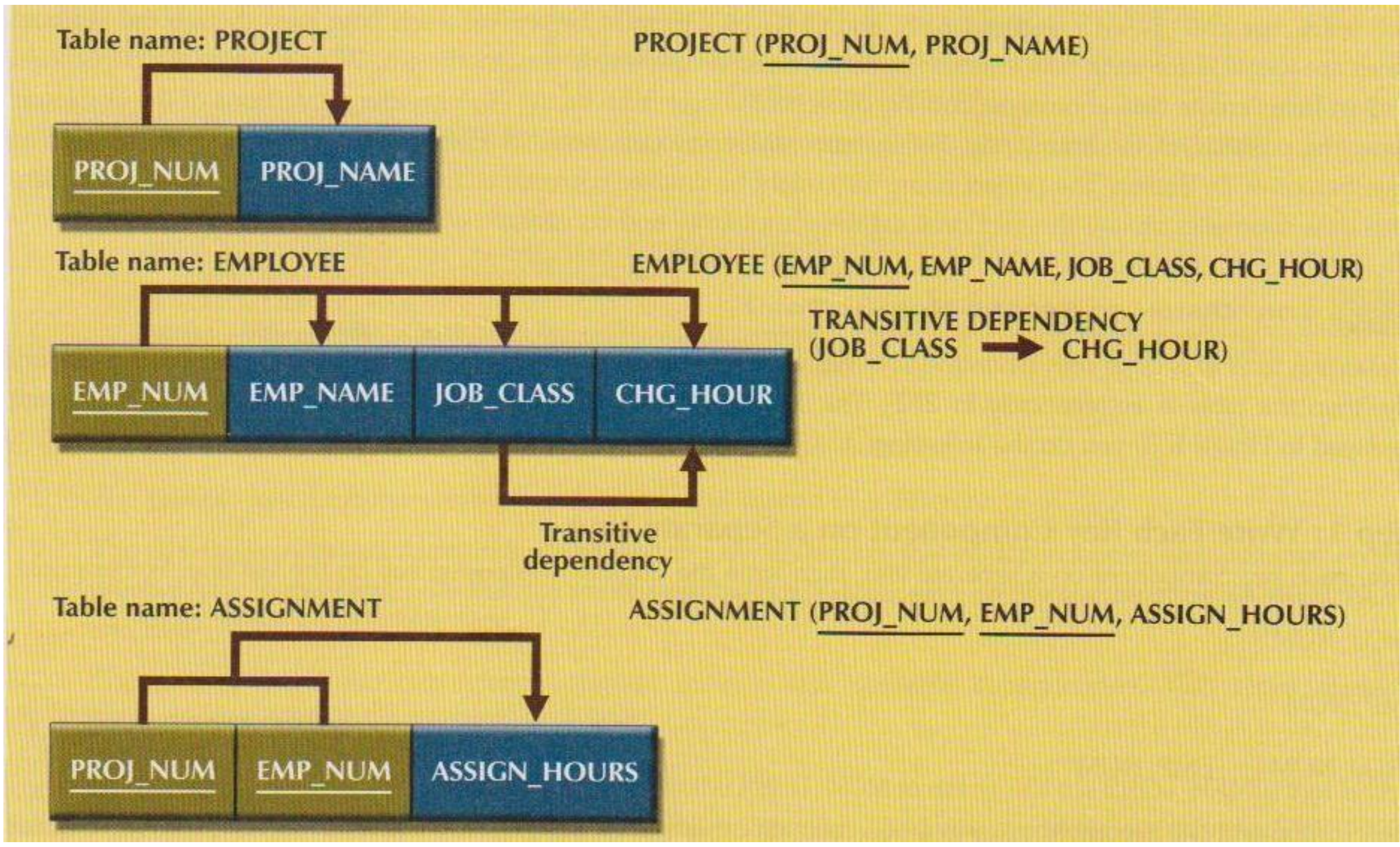
Each component will become the key in a new table. In other words, the original table is now divided in to three tables:

- (PROJECT, EMPLOYEE, and ASSIGNMENT).

Step 2: Assign Corresponding Dependent Attributes

Use dependency diagram to determine those attributes that are dependent on other attributes.

- PROJECT (**<u>PROJ_NUM</u>**, PROJ_NAME)
- EMPLOYEE (**<u>EMP_NUM</u>,** EMP_NAME, JOB_CLASS, CHG_HOUR)
- ASSIGNMENT (**<u>PROJ_NUM</u>**, **<u>EMP_NUM</u>**, ASSIGN_HOURS)



A table is in second normal form (2NF) when:

- it is in 1NF, And
- It includes no partial dependencies; that is, no attribute is dependent on only portion of the primary key. Note that is still possible for a table in 2NF to exhibit transitive dependency; that is, one or more attributes may be functionally dependent on non key attributes.

Conversion to Third Normal (3NF):

Step 1: Identify the Dependent Attributes

For every transitive dependency, write its determinant as PK for a new table.

- JOB_CLASS

Step 2: Identify the Dependent Attributes

Identify the attributes that are dependent on each determinant identified in Step 1 and identify the dependency.

- JOB_CLASS →CHG_HOUR

Name the table to reflect its contents and function. In this case, JOB seems appropriate.

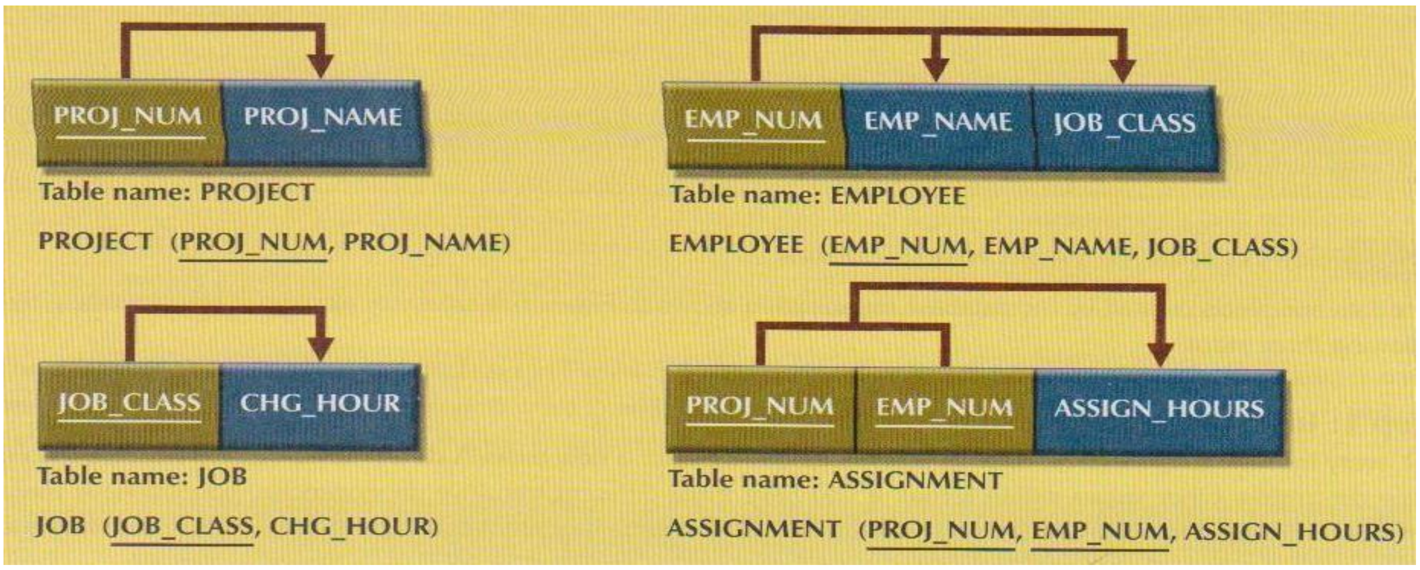
Step 3: Remove the Dependent Attributes from Transitive Dependencies

Eliminate all dependent attributes in the transitive relationship(s) from each of the tables that have such a transitive relationship.

- EMP_NUM → EMP_NAME, JOB_CLASS

Note that the JOB_CLASS remains in the EMPLOYEE table to save as FK.

After the 3NF conversion has been completed, your database contains four tables:



A table is in 3NF when:

- It is in 2NF
- It contains no transitive dependencies

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.

Boyce–Codd normal form (**BCNF**) is a normal form used in database normalization. It is a slightly stronger version of the third normal form (3NF). BCNF was developed in 1975 by Raymond F. Boyce and Edgar F. Codd to address certain types of anomalies not dealt with by 3NF as originally defined.[1]

If a relational schema is in BCNF then all redundancy based on functional dependency has been removed, although other types of redundancy may still exist. A relational schema $R$is in Boyce–Codd normal form if and only if for every one of its dependencies $X \rightarrow Y$, at least one of the following conditions hold:[2]

- $X \rightarrow Y$ is a trivial functional dependency (Y ⊆ X)
- X is a superkey for schema R

Only in rare cases does a 3NF table not meet the requirements of BCNF. A 3NF table that does not have multiple overlapping candidate keys is guaranteed to be in BCNF.[3]Depending on what its functional dependencies are, a 3NF table with two or more overlapping candidate keys may or may not be in BCNF.

An example of a 3NF table that does not meet BCNF is:

Today's Court Bookings

Court	Start Time	End Time	Rate Type
1	09:30	10:30	SAVER
1	11:00	12:00	SAVER
1	14:00	15:30	STANDARD
2	10:00	11:30	PREMIUM-B
2	11:30	13:30	PREMIUM-B
2	15:00	16:30	PREMIUM-A

- Each row in the table represents a court booking at a tennis club. That club has one hard court (Court 1) and one grass court (Court 2)
- A booking is defined by its Court and the period for which the Court is reserved
- Additionally, each booking has a Rate Type associated with it. There are four distinct rate types:
  - SAVER, for Court 1 bookings made by members
  - STANDARD, for Court 1 bookings made by non-members
  - PREMIUM-A, for Court 2 bookings made by members
  - PREMIUM-B, for Court 2 bookings made by non-members

The table's superkeys are:

- S_1 = {Court, Start Time}
- S_2 = {Court, End Time}
- S_3 = {Rate Type, Start Time}
- S_4 = {Rate Type, End Time}
- S_5 = {Court, Start Time, End Time}

- S_6 = {Rate Type, Start Time, End Time}
- S_7 = {Court, Rate Type, Start Time}
- S_8 = {Court, Rate Type, End Time}
- S_T = {Court, Rate Type, Start Time, End Time}, the trivial superkey

Note that even though in the above table *Start Time* and *End Time* attributes have no duplicate values for each of them, we still have to admit that in some other days two different bookings on court 1 and court 2 could *start at the same time* or *end at the same time*. This is the reason why {Start Time} and {End Time} cannot be considered as the table's superkeys.

However, only S_1, S_2, S_3 and S_4 are candidate keys (that is, minimal superkeys for that relation) because e.g. $S_1 \subset S_5$, so S_5 cannot be a candidate key.

Recall that 2NF prohibits partial functional dependencies of non-prime attributes (i.e., an attribute that does not occur in ANY candidate key. See candidate keys), and that 3NFprohibits transitive functional dependencies of non-prime attributes on candidate keys.

In **Today's Court Bookings** table, there are no non-prime attributes: that is, all attributes belong to some candidate key. Therefore the table adheres to both 2NF and 3NF.

The table does not adhere to BCNF. This is because of the dependency Rate Type → Court in which the determining attribute Rate Type - on which Court depends - (1.) is neither a candidate key nor a superset of a candidate key and (2.) Court Type is no subset of Rate Type.

Dependency Rate Type → Court is respected since a Rate Type should only ever apply to a single Court.

The design can be amended so that it meets BCNF:

Rate Types

Rate Type	Court	Member Flag
SAVER	1	Yes
STANDARD	1	No

PREMIUM-A	2	Yes
PREMIUM-B	2	No

Today's Bookings

Member Flag	Court	Start Time	End Time
Yes	1	09:30	10:30
Yes	1	11:00	12:00
No	1	14:00	15:30
No	2	10:00	11:30
No	2	11:30	13:30
Yes	2	15:00	16:30

The candidate keys for the Rate Types table are {Rate Type} and {Court, Member Flag}; the candidate keys for the Today's Bookings table are {Court, Start Time} and {Court, End Time}. Both tables are in BCNF. When {Rate Type} is a key in the Rate Types table, having one Rate Type associated with two different Courts is impossible, so by using {Rate Type} as a key in the Rate Types table, the anomaly affecting the original table has been eliminated.

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*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
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 ve 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) *./ f*R_1*; : : : ; R* $_{ng}$ is said to hold over a relation *R* if R_1*; : : : ;* R_n

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 *./ f*R_1*; : : : ;* R_{ng} that holds over *R*, one of the following statements is true:

- $R_i = R$ 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 *f*R_1*; : : : ;* R_{ng} is lossless-join whenever the key dependencies (FDs in which the left side is a key for *R*) hold. *./ f*R_1*; : : : ;* R_{ng} is a trivial JD if $R_i = R$ 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 – 5

Transaction concept, transaction state, Properties of a Transaction, concurrent executions, serializability, recoverability, implementation of isolation, Testing for serializability.

Transaction Concept:

Collection of operations that form a single logical unit of work are called transactions

Properties of transaction:

To ensure integrity of data, the database system should maintain following properties:

Atomicity: Either all operations of the transaction are reflected properly in the database or none are.

Consistency: Execution of a transaction in isolation preserves the consistency of the database.

Isolation: Even though multiple transactions execute concurrently, each transaction is unaware of other transactions executing concurrently in the system.

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

Transactions access data using two operations:

i) **Read(x),** which transfers the data item x from the database to a local buffer belonging to the transaction that executed the read operation

ii) **Write(x),** which transfers the data item x from the local buffer to the transaction that executed write back to the data base.

Ex:

Let T_i be a transaction that transfers Rs 50 from account A to account B,

This transaction can be defined as:

```
T_i :   read (A)
        A: =A-50
        Write (A)
        Read  (B)
        B: =B+50
        Write (B)
```

ACID Requirements:

i) **Consistency:** This Consistency requirement is that the sum of A and B unchanged by the execution of the transaction.

ii) **Atomicity:** Suppose, before execution of the transaction T_i the values of accounts A and B are Rs 1000, and Rs 2000, respectively. Now suppose during execution of transaction of transaction $T_{i,\ a}$ Failure occurs. Consider the failure occurs after write (A) operation but before write (B).In this case, the values of accounts A and

B reflected in the database are Rs 950 and Rs 2000. Thus, the sum A+B is no longer preserved. To avoid this, the atomicity should be ensured.

To ensure atomicity database system keeps track of the old values of any data on which a transaction performs a write. If transaction does not complete its execution, the database system restores the old values. Atomicity is handles by **transaction management** component

iii) **Durability:** The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

Ensuring durability is the responsibility of a component called the recovery management.

iv) **Isolation:** If several transactions are executed concurrently, their operation may interleave in some undesirable way, resulting in an inconsistent state.

For example, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Further more if second transaction then performs updates on A and B, the database may be left in inconsistent state even after both transactions have completed.

To avoid the problem of concurrent execution, transactions should be executed in isolation. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that could have been obtained if transactions are executed serially i.e. one after other.

Ensuring the isolation property is the responsibility of concurrent control component.

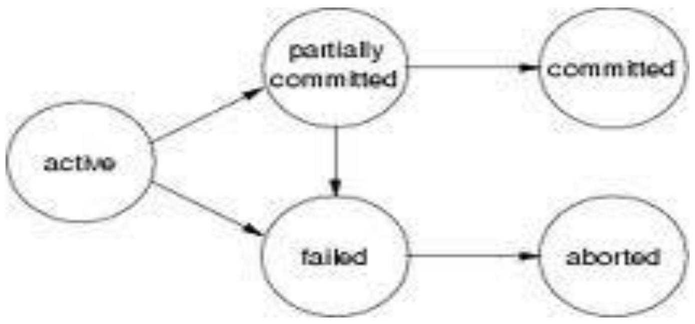
Transaction state:

In the absence of failures, all transactions completes execution successfully. However, it is also possible that transaction does not complete its execution successfully. Such a transaction is termed as aborted. It is necessary to undone the changes done by the aborted transaction to ensure the atomicity property. Once the changes done by the aborted transaction have been undone, we say that the transaction has been rolled back. A transaction that completes its execution successfully is said to be committed.

A transaction must be in one of the following states:

- **Active,** the initial state; the transaction stays in this state while it is executing.
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed,** after successful completion.

Fig shows the state diagram of a transaction. A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. When the last statement of the actual output is written out in disk from main memory, the transaction enters the committed state.



A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution. Such a transaction must be rolled back. Then, it enters the aborted state. At this point, ***the system has two options:***

i) It can **restart** the transaction, if the transaction was aborted as a result of some hardware or software error that was created through the internal logic of the transaction.

ii) It can **kill** the program, if the transaction was aborted because of internal logical error that can be corrected only by rewriting the application program.

Concurrent executions:

Transaction processing systems usually allow multiple transactions to run concurrently.

i) Improved throughput and resource utilization:

"**Throughput is number of transactions executed in a given amount of time**."

A transaction consists of many steps. Some involve **I/O** activity; others involve **CPU** activity. The CPU and the disks in a computer system can operate in parallel. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. If one transaction is reading or writing data on disk, another can be running in the CPU. All of this increases throughput of the system correspondingly, the processor and disk utilization also increases. Thus the processor and disk spend less time idle.

ii) Reduced waiting time :

If transaction runs serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to un-predictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to run them concurrently showing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. It also reduces the average response time.

Example: let **T1** and **T2** are two transactions. Transaction **T1** transfers Rs 50 from account **A** to account **B**. it is defined as:

T1:

```
read(A);
A:=A-50;
Write(A);
Read(B);
B:=B+50;
Write(B);
```

Transaction T2 transfer 10 percent of the balance from account A to account B. It is defined as:

T2:

```
read(A);
Temp:=A*0.1;
```

```
        A:=A-temp;
        Write(A);
        Read(B);
        B:=B+temp;
        Write(B);
```

Suppose the current values of accounts A and B are Rs1000 and Rs2000, respectively. Suppose the two transactions are executed in the order T1 followed by T2. This execution sequence appears in fig.

The final values of accounts A and B, after the execution of schedule 1 in fig are $855 and $2145, respectively. Thus the sum A+B is preserved.

If the two transaction are executed in the order T2 followed by T1, then also the corresponding execution sequence is that of fig. After execution of schedule 2, the sum A+B is preserved and the final values of accounts A and B are Rs 850 and Rs 2150 respectively.

T1	**T2**
read(A)	
A:=A-50	
Write(A)	
Read(B)	
B:=B+50	
Write(B)	
	read(A)
	Temp:=A*0.1
	A:=A-temp
	Write(A)
	Read(B)
	B:=B+temp
	Write(B)

Fig: Schedule 1-a serial schedule in which T1 is followed by T2

The execution sequences which represents the chronological order in which instructions are executed in the systems, are called **schedules.**

After execution of this schedule, we arrive at the same state as the one in which the transactions are executed serially in the order T1 followed by T2. The sum A+B is preserved.

T1	T2
read(A)	
A:=A-50	
Write(A)	
	read(A)
	Temp:=A*0.1
	A:=A-temp
	Write(A)
read(B)	
B:=B+50	
Write(B)	
	read(B)
	B:=B+temp
	Write(B)

Not all concurrent executions result in a correct state. Consider a schedule shown in fig. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are Rs 950 and Rs 2100, respectively. This final state is an inconsistent state.

Serializability

Need of serializability

Concurrent execution have following problems:

1) **Lost update**: The update of one transaction is overwritten by another transaction.

   Example: Suppose T1 credits \$ 100 to account A and T_2 debits \$ 50 from account A. The initial of A= 500. If credit and debit are applied correctly, then the final correct value of the account should be 550, if we run T_1 and T_2 concurrently as follows:

Time

T_1**(credit)**	T_2**(Debit)**

read(A) {A=500}
A:A+100 {A=600}
Write(A) {A=600}
	read(A) {A=500}
	A:A-50 {A=450}
	Write (A) {A=450}

Final value of A=450. The credit of T_1 is missing (lost update) from the account.

2) **Dirty read**: Reading of a non-existent value of A by T_2. If T1 update A which isthen read by T_2, then if T_1 aborts T_2 will have read a value of A which never existed.

Time

T_1 **(Credit)**	T_2 **(Debit)**

read(A) {A=500}
A:A+100 {A=600}
Write(A) {A=600}
 	
T1 failed to complete
	
	read(A) {A=500}
	A:A+100 {A=600}
	Write (A) {A=600}

T1 modified A=600. T2 read A=600. But T_1 failed and its effect is removed from the database, so A is restored to its old value, i.e. A=500.A=600 is a nonexistent value but read(reading dirty data) by T_2.

3) **Unrepeatable read**: if T_2 reads A, which is then altered by T_1 and T_1 commits. When T2 rereads A it will find different values of A in its second read

Time

T_1**(credit)**	T_2**(Debit)**

read(A) {A=500}
A:A+100 {A=600}
Write(A) {A=600}
	read(A) {A=500}
	A:A-50 {A=450}
	Write (A) {A=600}

In this execution T_1 reads A=500, T_2 reads A=500. T_1 modifies A to 600. When T_2 rereads A it gets A=600. This should not be the case. T_2 in the same execution should get only one value of A (500 or 600 and not both).

In serial execution these problem (dirty, read, unrepeatable read , and lost update) would not arise since serial execution does not share data items. This means we can use the results of serial execution as a measure of correctness and concurrent execution for improving resource utilization. We need *serialization* of concurrent transaction.

Serialization of concurrent transactions: process of managing the execution of a set of transactions in such a way that their concurrent execution produces the same end result as if they were run serially.

Definition: serializable Schedule

Given an interleaved execution of a set of n transaction; the following conditions hold for each transaction in the set.

- All transactions are correct in the sense that if any one of the transactions is executed by itself on a consistent database, the resulting database will be consistent.
- The transactions are logically correct and that no two transactions are interdependent.

The given interleaved execution of these transactions is said to be **serializable** if it produces the same result as some serial execution of the transactions.

Conflict and view serializable schcedule:

There are two types of serializability:

- Conflict serializability
- View serializability

Conflict serializable schedule:

Let us consider a schedule **S** in which there are two consecutive instructions $\mathbf{I_i}$ and $\mathbf{I_j}$ of transactions $\mathbf{T_i}$ and $\mathbf{T_j}$ respectively. If $\mathbf{I_i}$ and $\mathbf{I_j}$ refers to different data items, then we can swap $\mathbf{I_i}$ and $\mathbf{I_j}$ without affecting the results of any instruction in the schedule. however, if $\mathbf{I_i}$ and $\mathbf{I_j}$ refer to the same data item **Q**,then the order of the two steps may matter.there are four cases to consider:

1. **$\mathbf{I_i}$=read(Q), $\mathbf{I_j}$=read(Q)**. The order of $\mathbf{I_i}$ and $\mathbf{I_j}$ does not matter, since the same value of **Q** is read by $\mathbf{T_i}$ and $\mathbf{T_j}$ regardless of the order.
2. **$\mathbf{I_i}$=read(Q), $\mathbf{I_j}$=write(Q)**.If $\mathbf{I_i}$ comes before $\mathbf{I_j}$, then $\mathbf{T_i}$ does not read the value of **Q** that is written by $\mathbf{T_j}$ in instruction $\mathbf{I_j}$. If $\mathbf{I_j}$ comes before $\mathbf{I_i}$,then $\mathbf{T_i}$ reads the values of **Q** that is written by $\mathbf{T_j}$.thus the order of $\mathbf{I_i}$ and $\mathbf{I_j}$ matters.
3. **$\mathbf{I_i}$=write(Q), $\mathbf{I_j}$=read(Q).** The order of $\mathbf{I_i}$ and $\mathbf{I_j}$ matters, reason is same as previous case.
4. **$\mathbf{I_i}$=write(Q), $\mathbf{I_j}$=write(Q)**.Since both instructions are write operations, the order of these instructions does affect either $\mathbf{T_i}$ or $\mathbf{T_j}$. however , the value obtained by the next **read(Q)** instruction of **S** is affected, since the result of only the latter of the two write instructions is preserved in the database.

Thus, only in the case where both $\mathbf{I_i}$ and $\mathbf{I_j}$ are read instructions, the order of execution does not matter.

We say that $\mathbf{I_i}$ and $\mathbf{I_j}$ **conflict** if they are operations by different transactions on the same data item and at least one of these instructions is a **write** operation.

Consider the following schedule1

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule 1-Showing only the read and write operations

The **write(A)** of **T1** conflict with **read(A)** of **T2**. However, **write(A)** of **T2** does not conflict with **read(B)** of **T1**,hence we can swap these instructions to generate a new **schedule 2** as shown in Fig. regardless of initial system state, **schedule 1 and 2 generates same result**.

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
Read(B)	
	Write(A)
Write(B)	
	Read(B)
	Write(B)

Schedule 2- schedule 1 after swapping a pair of instructions

We can continue to swapping nonconflict instructions:

- ✓ Swap the read(B) instruction of T1 with read(A) instruction of T2.
- ✓ Swap the write(B) instruction of T1 with write(A) instruction of T2.
- ✓ Swap the write(B) instruction of T1 with read(A) instruction of T2.

The final result of these swaps is shown in fig, which is serial schedule.

T_1	T_2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Schedule 3- A serial schedule that is equivalennt to schedule 1

<u>View serializable schedule:</u>

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

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

The concept of **view equivalence** leads to the concept of **view serializability**.

Consider the **schedule 4** shown in fig. It is **view equivalent** to the serial schedule < T_i, T_i, Ti> since one **read(Q)** instruction reads the initial value of **Q** and **T_5** performs the final write(Q).

Every conflict serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable.

T_3	T_4	T5
Read(Q)		
	Write(Q)	
Write(Q)		
		Write(Q)

schedule 4- a view serializable schedule

In schedule 4, transaction **T_4** and **T_5** performs **write(Q)** operations without having performed a **read(Q)** operation.writes of this sort are called **blind writes.** View serilizable schedule with blind writes is not conflict serializable.

Testing of serializability:

Testing of serializability is done by using a directed graph, called **precedence graph,** constructed from schedule. This graph consists of a pair G=(V,E), where **V** is a set of vertices and **E** is a set of edges. The set of vertices consists of all transactions in schedule. The set of edges consists of all edges **$T_i \rightarrow T_j$** for which one of three conditions hold:

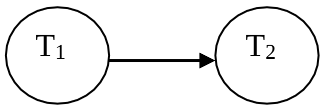
- T_i executes write(Q) before T_j executes read(Q).
- T_i executes read(Q) before T_j executes write(Q)
- T_i executes write(Q) before T_j executes write(Q)

Example:

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule 1

The precedence graph for the above schedule1 is shown in fig. It contains a single edge **$T_1 \rightarrow T_2$** , since all the instructions of T_1 are executed before the first instruction of T_2 is executed.

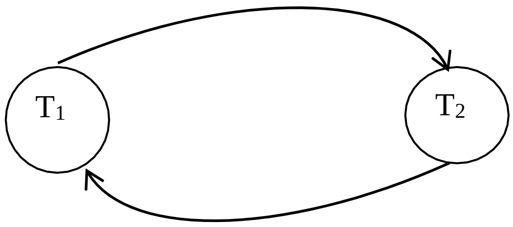


Consider following schedule 2

T_1	T_2
Read(A)	
A:=A-50	read(A)
	temp:=A*0.1
Write(Q)	A:=A-temp
	Write(A)
	Write(B)
Write(A)	
Read(B)	
B:=B+50	
Write(B)	
	B:=B+temp
	Write(B)

schedule 2

The precedence graph for schedule 2 is shown below



Precedence graph for schedule 2

Test for conflict serializability:

To test conflict serializability, construct a precedence graph for given schedule. If graph contains cycle, the schedule is not conflict serializable. If the graph contains no cycle, then the schedule is conflict serializable.

Schedule 1 is conflict serializable, as the precedence graph for Schedule does not contain any cycle. While the schedule 2 is not conflict serializable, as precedence graph for it contains cycle.

Topological sorting:

If the graph is acyclic, then using topological sorting given below, find serial schedule:

- Initialize the serial schedule as empty.
- Find a transaction T_i , such that there are no arcs entering $\mathbf{T_i}$, $\mathbf{T_j}$ is the next transaction in the serial schedule.
- Remove $\mathbf{T_i}$ and all edges emitting from $\mathbf{T_i}$. If the remaining set is non-empty, return to step 2, else the serial schedule is complete.

Example:

Consider the schedule given in Fig, is the corresponding schedule conflict serializable?

T_9	T_{10}	T_{11}
Read(A)		
A:= f_1(A)		
Write(A)		
	Read(A)	
	A:= f_2(A)	
	Write(A)	

	Read(B)	
	B:= f_3(B)	
	Write(B)	
		Read(B)
		B:= f_4(B)
		Write(B)

concurrent schedule

Sol: precedence graph for given schedule is shown below-

$$\begin{tikzcd} T_9 \arrow[r] & T_{10} \arrow[r] & T_{11} \end{tikzcd}$$

As the graph is acyclic, the schedule is conflict serializable.

Example

Consider the following schedule. Is the given schedule conflict serializable?

T_1	T_2	T_3
Read(A)		
	read(B)	
A:= f_1(A)		
		read(C)
	B:= f_2(B)	
	Write(B)	
		C:= f_3(C)
		Write(C)
Write(A)		
		Read(B)
	Read(A)	
	A:= f_4(A)	
Read(C)		
	Write(A)	
C:= f_5(C)		
Write(C)		
		B:= f_6(B)
		Write(B)

A concurrent schedule

Sol: the precedence graph for the given schedule is :

$$\begin{tikzcd} T_1 \arrow[rr] & & T_2 \arrow[dl] \\ & T_3 \arrow[ul] & \end{tikzcd}$$

The graph has cycle, therefore given schedule is not conflict serializable.

Test for view serializability:

The precedence graph used for testing conflict serializability cannot be used for testing view serializability. We need to extend the precedence graph to include labelled edges. This graph is called as **labelled precedence graph**.

Construction of labeled precedence graph:

Let **S** be a schedule consisting of transactions **{T_1 , T_2 ,…… T_n }**. Let **T_b** and **T_f** be two dummy transactions such that **T_b** issues **write(Q**) for each **Q** accessed in **S**, and **T_f** issues **read(Q)** for each **Q** accessed in **S**. we construct a new schedule **S'** from **S** by inserting **T_b** at the beginning of **S**, and appending **T_f** to the end of **S**. we construct the labelled precedence graph for schedule **S'** as follows:

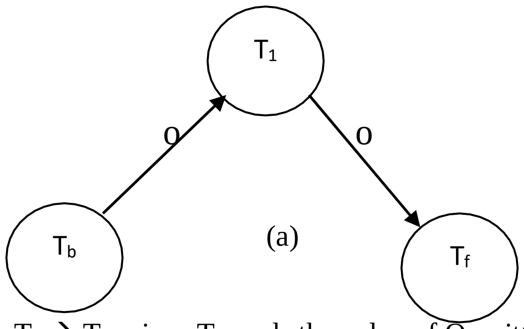
- Add an edge **$T_i \rightarrow T_j$** , if transaction **T_j** reads the value of data item **Q** written by transaction **T_j** .
- Remove all edges incident on useless transactions. A transaction **T_i** is useless if there exists no path , in the precedence graph from **T_i** to transaction **T_f**.
- For each data item **Q** such that **T_j** reads the value of **Q** written by **T_i** , and **T_k** executes **write(Q)** and $T_k \neq T_b$, do the following:
    a. If $T_i = T_b$ and $T_j \neq T_f$, then insert the edge $T_j \rightarrow T_k$ in the labelled precedence graph.
    b. $T_i \neq T_b$ and $T_j = T_f$, then insert the edge $T_k \rightarrow T_i$ in the labelled precedence graph.
    c. If $T_i \neq T_b$ and $T_j \neq T_f$, then insert the pair of edges $T_k \rightarrow T_i$ and $T_j \rightarrow T_k$ in the labelled precedence graph where p is a unique integer larger than **o** that has not been used earlier for labelling edges.

Example: Prepare the labelled precedence for following schedule

T_1	T_2
Read(Q)	
	Write(Q)
Write(Q)	

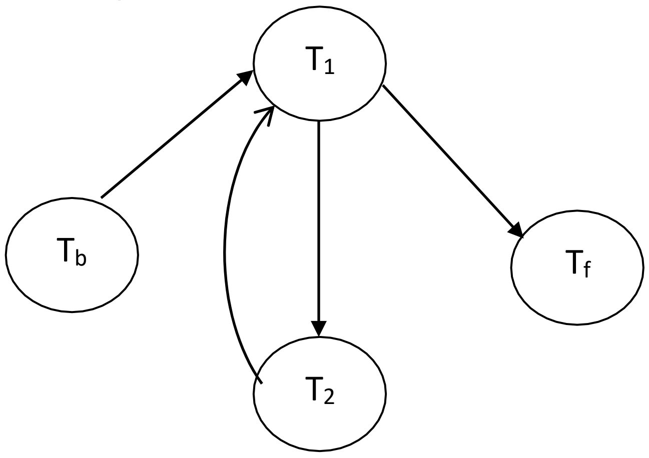
Concurrent schedule 1

Solution: for above schedule, the graph constructed in step1 and step2 is shown below:



It contains the edge $T_b \rightarrow T_1$, since T_1 reads the value of Q written by T_b . it contains the $T_1 \rightarrow T_f$, since T_1 performs the final write(Q), and T_f reads that value.

The final graph corresponding to schedule as shown below:



Recoverability:

If a transaction **T_i** fails, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary to ensure that any transaction **T_j** that is dependent on **T_i** should also be aborted. To achieve this, we need to place restrictions on the type of schedules permitted in the system.

Types of schedules that are acceptable from the view point of recovery from transaction failure are:

- Recoverable schedules
- Cascadeless schedules.

Recoverable schedules:

A Recoverable schedule is one where, for each pair of transactions **T_i** and **T_j** such that **T_j** reads a data item previously written by **T_i,** the commit operation of **T_i** appears before the commit operations of $T_{j.}$

Consider schedule1 in fig,in which **T_2** is a transaction that performs only one instruction; read(A). Suppose that system allows **T_2** to commit immediately after executing the read(A) instruction. Thus, **T_2** commits before **T_1. Suppose that T_1 fails before it commits. Since T_2** has read the value of data item. A written by **$T_{1,}$** we must abort **T_2** to ensure transaction atomicity.

T_1	T_2
Read(A)	
Write(A)	
	Read(A)
Read(B)	

schedule 1

Cascadeless schedules:

Even if a schedule is recoverable, to recover correctly from the failure of a transaction **T_i** , we may have to roll back several transactions. Such transactions occur if transactions have read data written by **$T_{i.}$**

Consider **schedule2** of fig. Transaction **T_1** writes a value of **A** that is read by transaction **$T_{2.}$** Transaction **T_2** writes a value of **A** that is read by **$T_{3.}$** Suppose that, at this point, **T_1** fails, **T_1** must be rolled back. Since **T_2** is dependent on **$T_{1,}$ T_2** must be rolled back. Similarly as **T_3** is dependent on **$T_{2.}$ T_3** should also be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction roll backs, is called *cascading rollback.*

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. Therefore, schedules should not contain ***cascading rollbacks.*** Such schedules are called Cascadeless schedules.

A **Cascadeless schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the read operation of $T_{j.}$

T_1	T_2	T_3
Read(A)		
Read(B)		

Write(A)		
	Read(A)	
	Write(A)	
		Read(A)

Schedule 2

Implementation of isolation:

An overview of how some of most important concurrency control mechanisms work
As we know that, in order to maintain consistency in a database, it follows ACID properties. Among these four properties (Atomicity, Consistency, Isolation and Durability) Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.
Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level are defined by the following phenomena –

- **Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been commited.For example, Let's say transaction 1 updates a row and leaves it uncommited, meanwhile Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
- **Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time. For example, suppose transaction T1 reads a data. Due to concurrency, another transaction T2 updates the same data and commit, Now if transaction T1 rereads the same data, it will retrieve a different value.
- **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that matches the search criteria for transaction T1. If transaction T1 reexecutes the statement that reads the rows, it gets a different set of rows this time.

Based on these phenomena, The SQL standard defines four isolation levels :

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet commited changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction hold a read or write lock on the current row, and thus prevent other rows from reading, updating or deleting it.
3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and write locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non repeatable read.

4. **Serializable** – This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently ececuting transactions appears to be serially executing.

Implementation of isolation levels:

we provide an overview of how some of most important concurrency control mechanisms work,

Locking Instead of locking the entire database, a transaction could, instead, lock only those data items that it accesses. Under such a policy, the transaction must hold locks long enough to ensure serializability, but for a period short enough not to harm performance excessively.

two-phase locking requires a transaction to have two phases, one where it acquires locks but does not release any, and a second phase where the transaction releases locks but does not acquire any.

Further improvements to locking result if we have two kinds of locks: shared and exclusive. Shared locks are used for data that the transaction reads and exclusive locks are used for those it writes

Timestamps Another category of techniques for the implementation of isolation assigns each transaction a timestamp, typically when it begins.

For each data item, the system keeps two timestamps.

The read timestamp :of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item.

The write timestamp of a data item holds the timestamp of the transaction that wrote the current value of the data item