INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad -500 043

INFORMATION TECHNOLOGY

COURSE LECTURE NOTES

Course Name	**DATABASE MANAGEMENT SYSTEMS**
Course Code	ACSB08
Programme	B.Tech
Semester	IV
Course Coordinator	Mr. U. Sivaji, Assistant Professor
Course Faculty	Mr. N PoornaChandra Rao, Assistant Professor Mr. N Bhaswanth, Assistant Professor Ms. B Ramya sree, Assistant Professor Ms. K Mayuri, Assistant Professor Ms. B Vijaya Durga, Assistant Professor
Lecture Numbers	1-63
Topic Covered	All

COURSE OBJECTIVES (COs):

The course should enable the students to:	
I	Understand the role of database management system in an organization and learn the database Concepts.
II	Design Entity Relationship model for a database.
III	Demonstrate the use of constraints and relational algebra operations.
IV	Describe the basics of SQL and construct queries using SQL.
V	Understand the importance of normalization in databases.

COURSE LEARNING OUTCOMES (CLOs):

Students, who complete the course, will have demonstrated the ability to do the following:

ACSB08.01	Describe the Purpose of Database Systems, Data Models, and View of Data.
ACSB08.02	Summarize the concept of Database Languages, Database Users.
ACSB08.03	Identify the Various Components of overall DBS architecture.
ACSB08.04	Use the concept of ER Model.

ACSB08.05	Describe Basics of Relational Model.
ACSB08.06	Determine Relational algebra, The Self variable.
ACSB08.07	Understand selection and projection, set operations.
ACSB08.08	Determine renaming, joins, division.
ACSB08.09	Use examples of algebra queries.
ACSB08.10	Illustrate Tuple relational calculus, Domain relational calculus, and also expressive power of algebra and calculus.
ACSB08.11	Understand SQL – Data Definition commands, Queries with various options.
ACSB08.12	Analyze the concept of Mata manipulation commands, Views, Joins, views.
ACSB08.13	Illustrate Calling a function, Returning multiple values from a function.
ACSB08.14	Contrast the Usage of Relational database design, Functional dependencies, Armstrong Axioms
ACSB08.15	Define Normalization, 2nd and 3rd Normalization, Basic definitions of MVDs and JDs, 4th and 5th normal forms.
ACSB08.16	Discuss the concept of Transaction, Transaction State.
ACSB08.17	Understand Atomicity and Durability, Concurrent Executions.
ACSB08.18	Summarize the concept of Serializability, Recoverability.
ACSB08.19	Discuss the Concurrency Control and various Protocols.
ACSB08.20	Understand the concept of Multiversion Schemes, Deadlock Handling. Recovery: Failure Classification, Storage Structure, Recovery and Atomicity, Log-Based Recovery, Shadow Paging, Recovery with Concurrent Transactions Buffer Management.
ACSB08.21	Knowledge about the Physical Storage Media, Magnetic Disks, Storage Access
ACSB08.22	Apply Working with File Organization, Organization of Records in Files.
ACSB08.23	Understand Ordered Indices, B+-Tree Index Files, B-Tree Index Files, Static Hashing, Dynamic Hashing.
ACSB08.24	Comparison of Ordered Indexing and Hashing.
ACSB08.25	Illustrate Query Processing: Overview, Measures of Query Cost.

SYLLABUS

Module-I	**CONCEPTUAL MODELING INTRODUCTION**	**Classes: 09**
Introduction to Data bases: Purpose of Database Systems, View of Data, Data Models, Database Languages, Database Users, Various Components of overall DBS architecture, Various Concepts of ER Model, Basics of Relational Model		
Module-II	**RELATIONAL APPROACH**	**Classes: 09**
Relational algebra and calculus: Relational algebra, selection and projection, set operations, renaming, joins, division, examples of algebra queries, relational calculus: Tuple relational calculus, Domain relational calculus, expressive power of algebra and calculus		
Module-III	**SQL QUERY - BASICS , RDBMS - NORMALIZATION**	**Classes: 09**
SQL – Data Definition commands, Queries with various options, Mata manipulation commands, Views, Joins, views, integrity and security; Relational database design: Pitfalls of RDBD, Lossless join decomposition, Functional dependencies , Armstrong Axioms, Normalization for relational databases 1st , 2 nd and 3rd normal forms, Basic definitions of MVDs and JDs, 4th and 5th normal forms.		
Module-IV	**TRANSACTION MANAGEMENT**	**Classes: 09**
Transaction processing: Transaction Concept, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability. Concurrency Control: Lock-Based Protocols, Timestamp-Based Protocols, Validation-Based Protocols, Multiple Granularity, Multiversion Schemes, Deadlock Handling. Recovery: Failure Classification, Storage Structure, Recovery and Atomicity, Log-Based Recovery, Shadow Paging, Recovery With Concurrent Transactions Buffer Management.		
Module-V	**DATA STORAGE AND QUERY PROCESSING**	**Classes: 09**
Data storage: Overview of Physical Storage Media, Magnetic Disks, Storage Access, File Organization, Organization of Records in Files. Indexing and Hashing: Basic Concepts: Ordered Indices, B+-Tree Index Files, B-Tree Index Files, Static Hashing, Dynamic Hashing, Comparison of Ordered Indexing and Hashing. Query Processing: Overview, Measures of Query Cost.		
Text Books:		
1. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "Database System Concepts", McGraw-Hill, 6th Edition, 2017.		
Reference Books:		
1. Raghu Ramakrishnan, "Database Management System", Tata McGraw-Hill Publishing Company, 3rd Edition, 2007. 2. Hector Garcia Molina, Jeffrey D. Ullman, Jennifer Widom, "Database System Implementation", Pearson Education, United States, 1st Edition, 2000. 3. Peter Rob, Corlos Coronel, "Database System, Design, Implementation and Management", Thompson Learning Course Technology, 5th Edition, 2003		

Module-I

Conceptual Modeling

Introduction to Data bases

- Data: data is a collection of raw facts and figures.
- DataBase: database is a collection of interrelated data.
- DBMS: database is a collection of interrelated data and set of programs can access that system.

Database System Applications

1. Enter Price Information:
   - Sales: customers, products, purchases
   - Accounting: payments, receipts, account balance, assets.
   - Human Resources: employee records, salaries, tax deductions
   - Manufacturing: production, inventory, orders, supply chain
   - Online Retails: order tracking, customized recommendations
2. Banking and Finance: all transactions
   - Credit card Transaction: generation of monthly statements.
   - Finance: storing information about holdings and sales,
3. Universities: registration, grades
4. Airlines: reservations, schedules
5. Telecommunications: keeping records of calls made, generating monthly bills

Purpose of Database Systems

In the early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data:

- Data redundancy and inconsistency: Multiple file formats, duplication of information in different files
- Difficulty in accessing data: Need to write a new program to carry out each new task
- Data isolation: multiple files and formats
- Integrity problems: Hard to add new constraints or change existing ones

- Atomicity of updates: Failures may leave database in an inconsistent state with partial updates carried out. Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access anomalies: Example: Two people reading a balance and updating it at the same time
- Security problems: Database systems offer solutions to all the above problems

View of Data

A database is a collection of interrelated data and set of programs that allow users to accessand 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 stored and maintained.

Data Abstraction:

Major purpose of DBMS is to provide users with abstract view of data i.e. the system hides certain details of how the data are stored and maintained. Since database system users are not computer trained, developers hide the complexity from users through 3 levels of abstraction, to simplify user's interaction with the system.

➢ Levels of Abstraction

- Physical level of data abstraction: How the data are actually stored. This s the lowest level of abstraction which describes how data are actually stored.
- Logical level of data abstraction: This level hides what data are actually stored in the database and what relations hip exists among them. Describes data stored in database, and the relationships among the data.
- View Level of data abstraction: View provides security mechanism to prevent user from accessing certain parts of database. application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

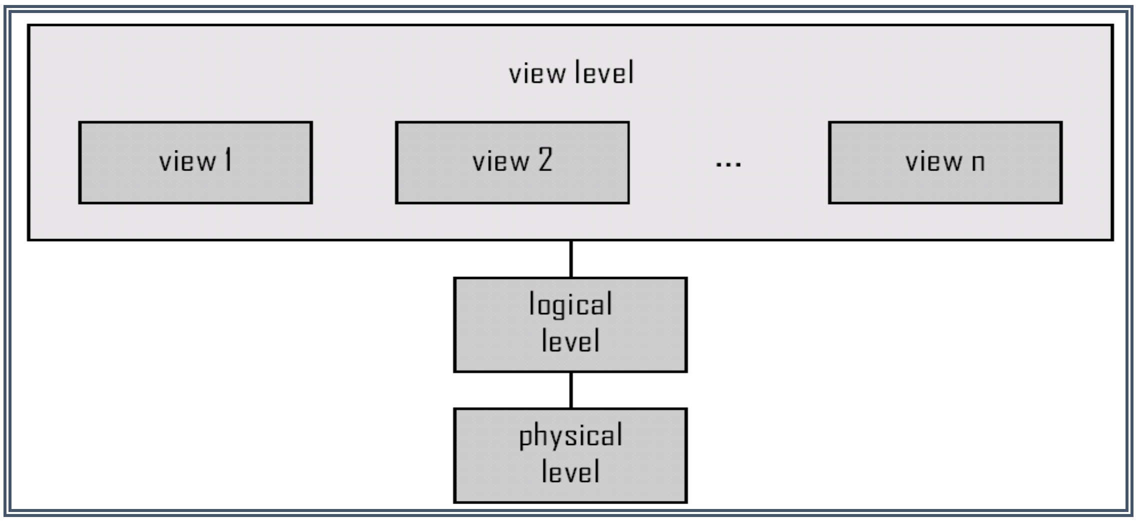


Figure 1.TheThree level of abstraction.

Instances and schemas:

Instance: The collection of information stored in the database at a particular movement is called an instance of the database.

- Similar to types and variables in programming languages

Schema: the overall design of the database is called the database schema.

- Example: The database consists of information about a set of customers and accounts and the relationship between them .Analogous to type information of a variable in a program

– Physical schema: database design at the physical level.

Logical schema: database design at the logical level

Data Models

Data Model: Underlying the structure of a database is the data model,

A collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints.

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

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

An entity is a “thing” or “object” in the real world that is distinguishable from other object

- Object-Based Data Models: Object-oriented Programming (especially in Java, C++,or C#).

- Semi structured Data Model: The semi structured data model permits the specification of data where individual data items of the same type may have different sets of attributes.

- Other older models:
  - Network Model
  - Hierarchical Model

Database Languages

A Database provides a DDL to specify the database schema and a DML to express database queries and updates.

Data-Manipulation Language

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

The types of access are:

• Retrieval of information stored in the database

• Insertion of new information into the database

• Deletion of information from the database

• Medication of information stored in the database

There are basically two types:

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

• Declarative DMLs (also referred to as non procedural DMLs) require user to specify what data are needed without specifying how to get those data.

Data- Definition Language (DDL):

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

SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc…

Example: create table accoun(account_number char(10), branch_name char(10),

balance integer)

In addition, the DDL statement updates the data dictionary, which contains metadata; the schema of a table is an example of metadata.

Data Base Access From Application Programs:

Application programs are programs that are used to interact with the database.

To access the database, DML Statements need to be executed form the host language. '

There are two ways o do this.

- By Providing an Application Program interface (set of procedures) that can be used to send DML and DDL statement to the database and retrieve the results.(ODBC and JDBC).

- By extending the host language syntax to embed DML calls within the host language program. A special character prefaces DML calls and preprocessor called the DML pre complier ,converts the DML statements to normal procedure calls in the host language

SQL: widely used non-procedural language

Example 1: Find the name of the customer with customer-id 192-83-7465

```
select      customer.customer_name
from        customer
where       customer.customer_id = '192-83-7465'
```

Example 2: Find the balances of all accounts held by the customer with customer-Id 192-83-7465.

```
select      account.balance
from     depositor, account
where       depositor.customer_id = '192-83-7465' and
  depositor.account_number = account.account_number
```

Example 3: Find the name of the customer with customer-id 192-83-7465

```
  select    customer.customer_name
  from      customer
  where     customer.customer_id = '192-83-7465'
```

Database Users

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

Data base Users and User Interfaces

- ➢ There are four different types of database system users, differentiated by the way they expect to interact with the system.

- ➢ Different types of user interfaces have been designed for the different types of users.

  - Na¨ıve users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a clerk in the university who needs to add a new instructor to Users are differentiated by the way they expect to interact with the system department A invokes a program called New - hire. This program asks the clerk for the name of the new instructor, her new ID, the name of the department (that is, A), and the salary

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

- Sophisticated users: interact with the system with out writing programs. In- stead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.

- Specialized users: are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer aided design systems, knowledge- base and expert systems, systems that store data with complex data types (for example, graphics data and audio data),and environment-modeling systems.

Database Administrator

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

The functions of a DBA include:

- Schema definition. The DBA creates the original database schema by executing a set of data definition statements in the DDL.

- Storage structure and access-method definition.

- Schema and physical-organization modification.

– Routine maintenance.

– Periodically backing up the database.

– Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.

– Monitoring jobs running on the Data base.

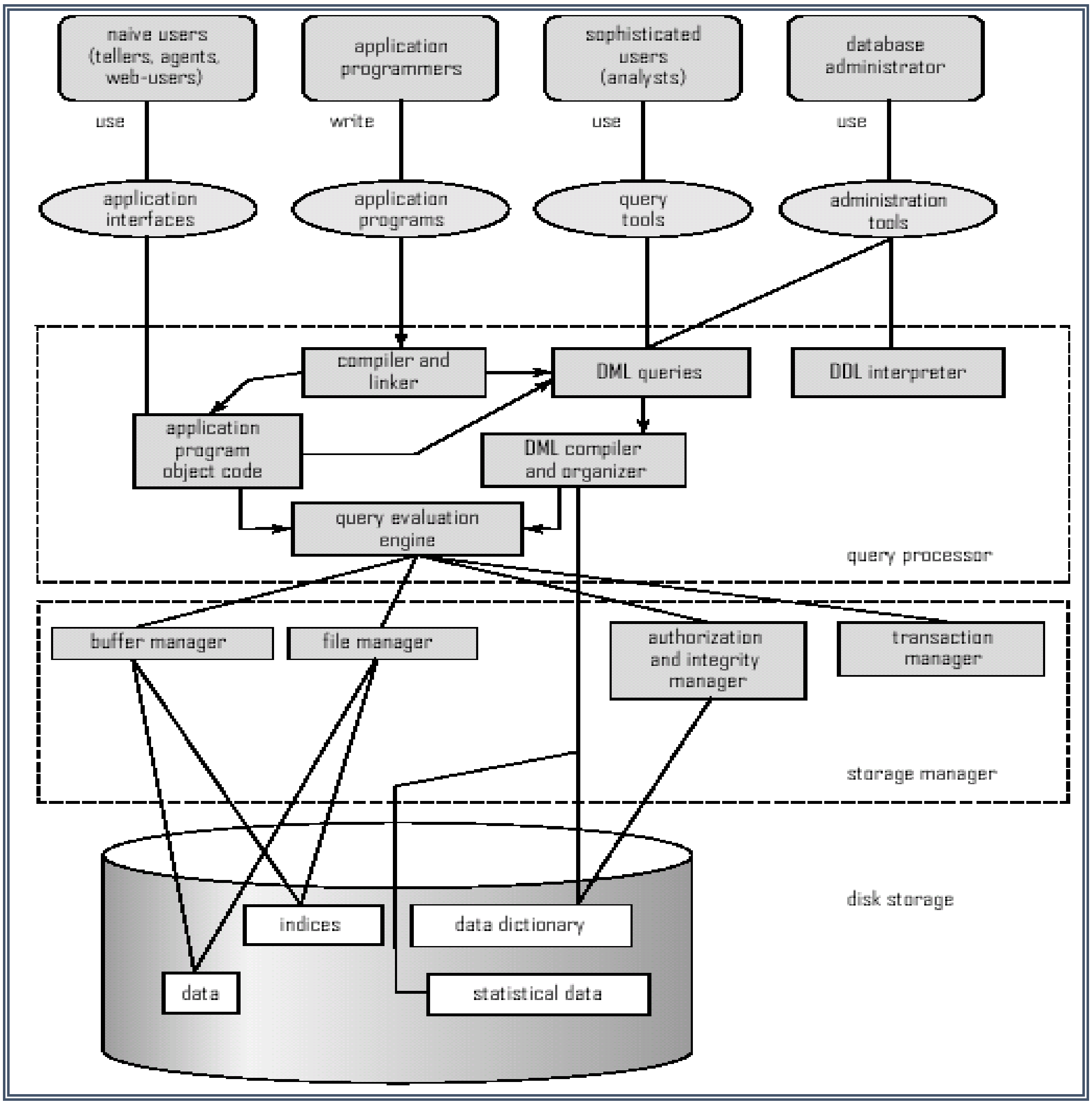
Various Components of overall DBS architecture

The architecture of database systems is greatly influenced by the underlying computer system on which the database is runs:

Database system can be.

- Centralized
- Client-server
- Parallel (multiple processors and disks)
- Distributed

Overall System Structure



Database Application Architectures:

– Database applications are usually partitioned into two or three parts, as in Figure1.8.1..In a two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements.
– Application program interface standards like ODBC and JDBC are used for interaction between the client and the server. In contrast,
– In a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls.
– Instead, the client end communicates with an application server, usually through a forms interface.
– The application server in turn communicates with a database system to access data.

- The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.
- Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.

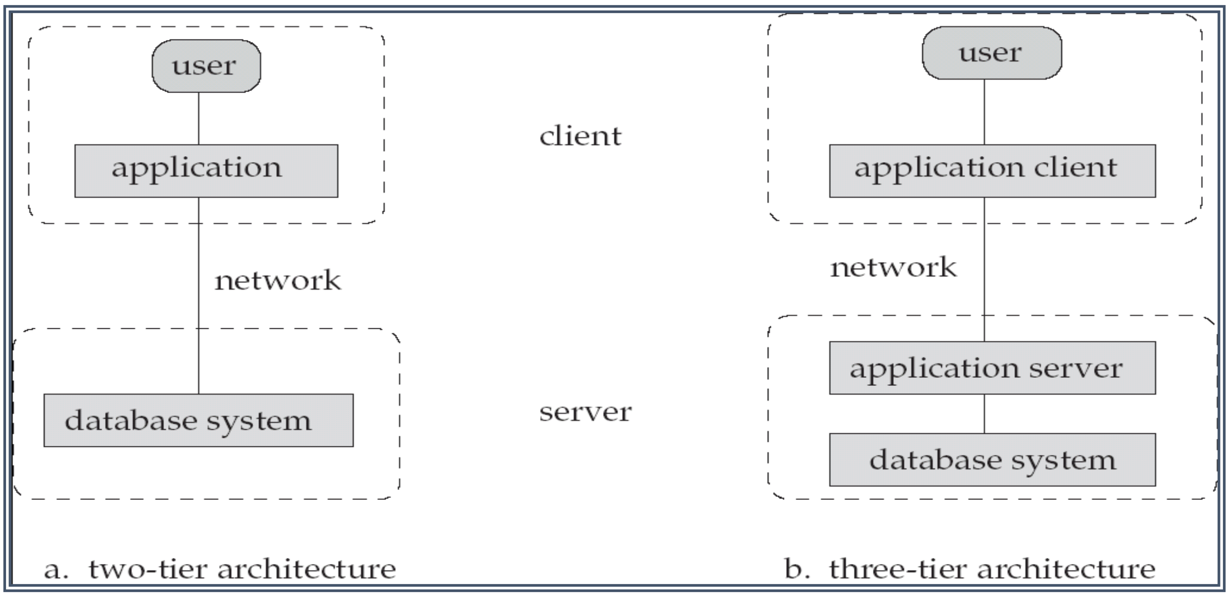


Figure .Two-tier and Three –tier architecture.

Various Concepts of ER Model

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

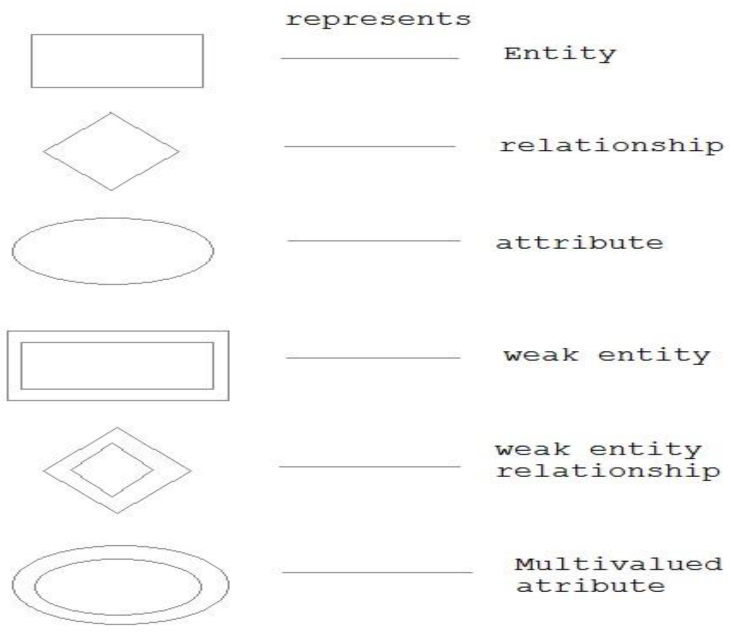
(1) Requirements Analysis: The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out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

(2) 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.

(3) 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 DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.

The result is a conceptual schema, sometimes called the logical schema, in the relational data model.

E-R DIAGRAMS



Beyond ER Design

The ER diagram is just an approximate description of the data, constructed through a very subjective evaluation of the information collected during requirements analysis.

Once we have a good logical schema, we must consider performance criteria and design the physical schema. Finally, we must address security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them. The remaining three steps of database design are briefly described below:

(4) Schema Refinement: The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems,

(5) 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

(6) 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, the product manager).

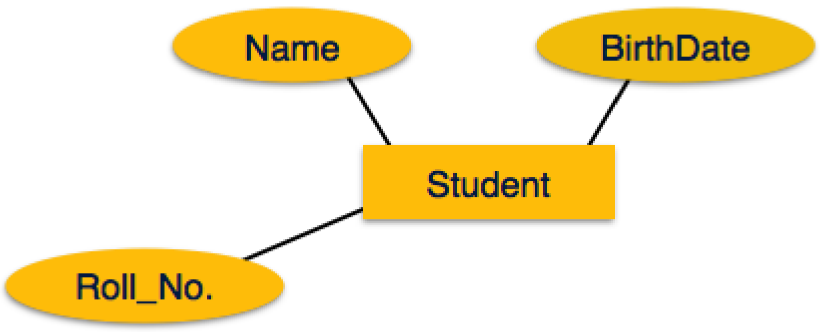
For each role and user group, we must identify the parts of the database that they must be able to access and the parts of the database that they should not be allowed to access, and take steps to ensure that they can access only the necessary parts.

Types of attributes

Attributes are properties of entities. Attributes are represented by means of eclipses. Every eclipse represents one attribute and is directly connected to its entity (rectangle).

- Simple attribute:

  Simple attributes are atomic values, which cannot be divided further. For example, student's phone-number is an atomic value of 10 digits.

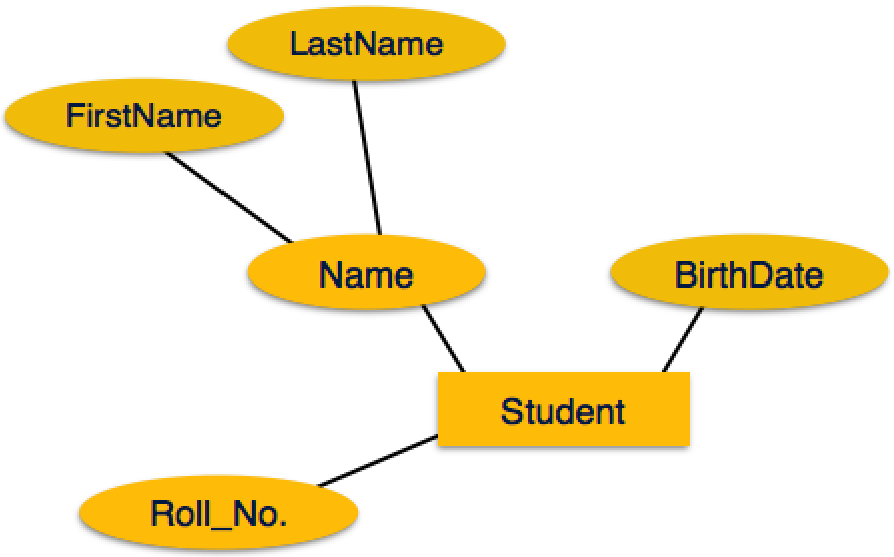


[*Image: Simple Attributes*]

Composite attribute:

Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.

If the attributes are composite, they are further divided in a tree like structure. Every node is then connected to its attribute. That is composite attributes are represented by eclipses that are connected with an eclipse.



[*Image: Composite Attributes*]

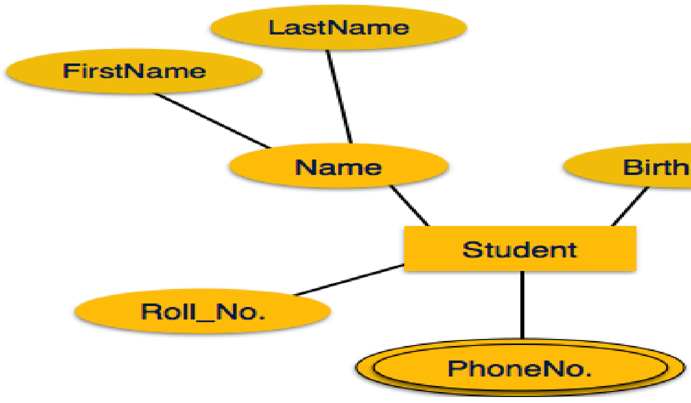
Single-valued attribute:

Single valued attributes contain on single value. For example: Social_Security_Number.

Multi-value attribute:

Multi-value attribute may contain more than one values. For example, a person can have more than one phone numbers, email_addresses etc.

Multivalued attributes are depicted by double

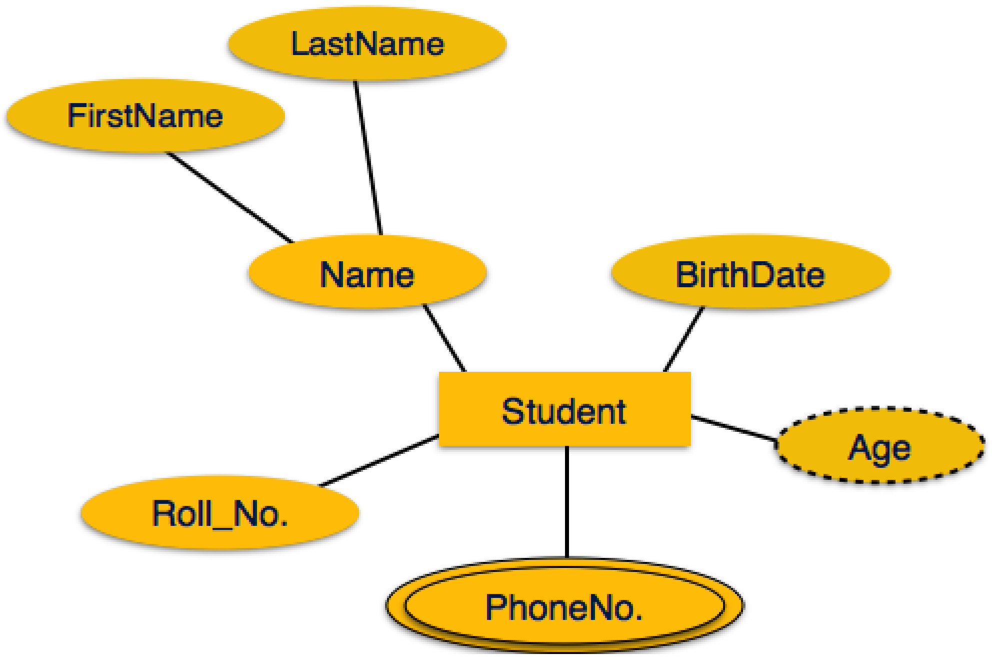


eclipse.

Derived attribute:

Derived attributes are attributes, which do not exist physical in the database, but there values are derived from other attributes presented in the database. For example, average_salary in a department should be saved in database instead it can be derived. For another example, age can be derived from data_of_birth.

Derived attributes are depicted by dashed eclipse.



[*Image: Derived Attributes*]

Relationships and Relationship Sets

The association among entities is called relationship. For example, employee entity has relation work

s_at with department. Another example is for student who enrolls in some course. Here, Works_at and Enrolls are called relationship.

Relationship Set:

Relationship of similar type is called relationship set. Like entities, a relationship too can have attributes. These attributes are called descriptive attributes.

Degree of relationship

The number of participating entities in an relationship defines the degree of the relationship.

Binary = degree 2

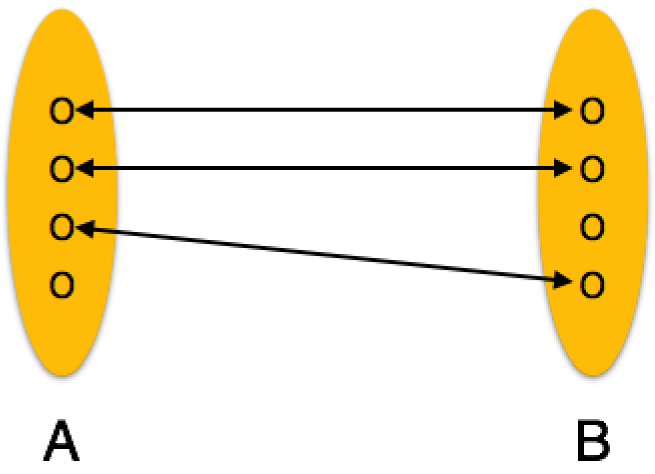
Ternary = degree 3

n-ary = degree

Mapping Cardinalities:

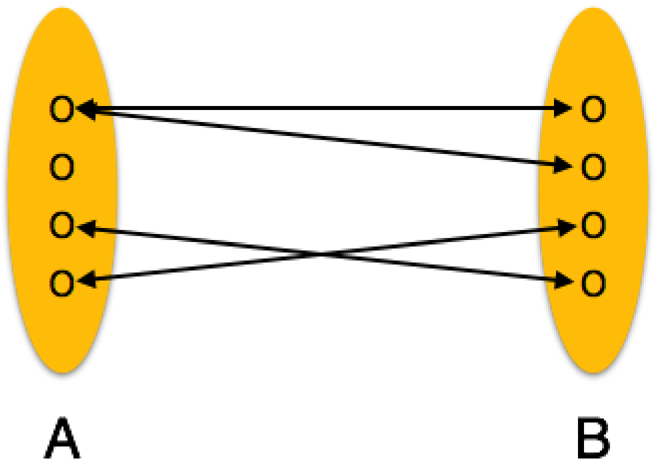
Cardinality defines the number of entities in one entity set which can be associated to the number of entities of other set via relationship set.

One-to-one: one entity from entity set A can be associated with at most one entity of entity set B and vice versa.



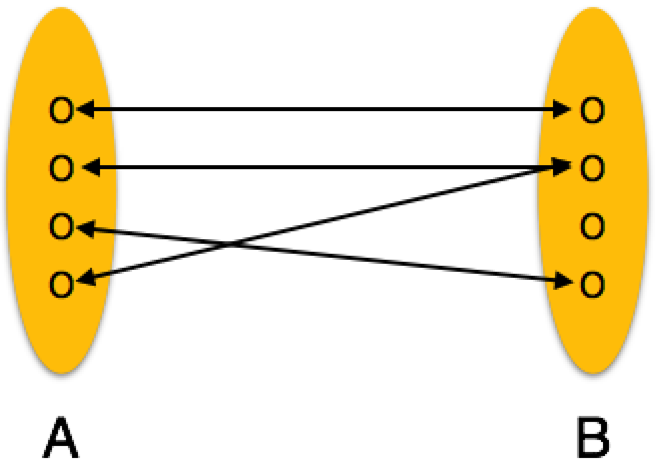
[*Image: One-to-one relation*]

One-to-many: One entity from entity set A can be associated with more than one entities of entity set B but from entity set B one entity can be associated with at most one entity.



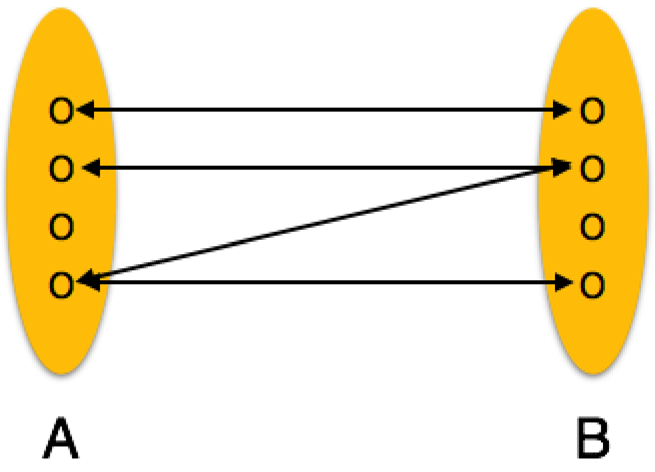
[Image: One-to-many relation]

Many-to-one: More than one entities from entity set A can be associated with at most one entity of entity set B but one entity from entity set B can be associated with more than one entity from entity set A.



[Image: Many-to-one relation]

Many-to-many: one entity from A can be associated with more than one entity from B and vice versa.



[Image: Many-to-many relation]

An entity is an object in the real world that is distinguishable from other objects.

Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy department.

A collection of similar entities is called an entity set.

A attribute is an property of an entity.

Additional Features Of Er Model:

- Key Constraints
- Participation Constraints
- Weak Entities
- Class Hierarchies
- Aggregation
- Key Constraints

There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called key for that relation. If there are more than one such minimal subsets, these are called candidate keys.

Weak Entities

A weak entity can be identified uniquely only by considering the primary key of another (owner) entity.

Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).

Weak entity set must have total participation in this identifying relationship set.

Weak Entity Sets

An entity set that does not have a primary key is referred to as a weak entity set.

The existence of a weak entity set depends on the existence of a identifying entity set

it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set Identifying relationship depicted using a double diamond The discriminator (or partial key) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set. The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator depict a weak entity set by double rectangles.

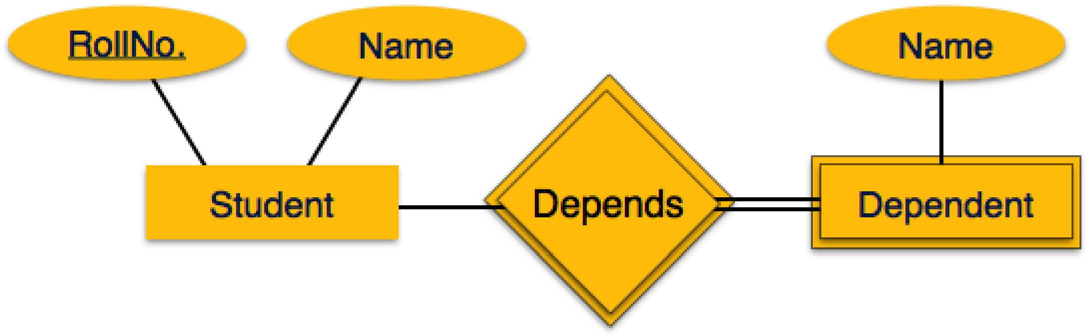
Under line the discriminator of a weak entity set with a dashed line.

More Weak Entity Set Examples

In a university, a course is a strong entity and a course_offering can be modeled as a weak entity The discriminator of course_offering would be semester (including year) and section_number (if there is more than one section). If we model course_offering as a strong entity we would model course_number as an attribute. Then the relationship with course would be implicit in the course_number attribute

A weak entity sets is one which does not have any primary key associated with it.

Mapping process (Algorithm):



[Image: Mapping Weak Entity Sets]

Basics of Relational Model

Relational Model Concepts

1. Attribute: Each column in a Table. Attributes are the properties which define a relation. e.g., Student_Rollno, NAME,etc.
2. Tables – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
3. Tuple – It is nothing but a single row of a table, which contains a single record.
4. Relation Schema: A relation schema represents the name of the relation with its attributes.
5. Degree: The total number of attributes which in the relation is called the degree of the relation.
6. Cardinality: Total number of rows present in the Table.
7. Column: The column represents the set of values for a specific attribute.
8. Relation instance – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
9. Relation key - Every row has one, two or multiple attributes, which is called relation key.
10. Attribute domain – Every attribute has some pre-defined value and scope which is known as attribute domain

Relational Integrity constraints

Relational Integrity constraints is referred to conditions which must be present for a valid relation. These integrity constraints are derived from the rules in the mini-world that the database represents.

There are many types of integrity constraints. Constraints on the Relational database management system is mostly divided into three main categories are:

1. Domain constraints
2. Key constraints
3. Referential integrity constraints

Domain Constraints

Domain constraints can be violated if an attribute value is not appearing in the corresponding domain or it is not of the appropriate data type.

Domain constraints specify that within each tuple, and the value of each attribute must be unique. This is specified as data types which include standard data types integers, real numbers, characters, Booleans, variable length strings, etc.

Example:

Create DOMAIN CustomerName

CHECK (value not NULL)

The example shown demonstrates creating a domain constraint such that CustomerName is not NULL

Key constraints

An attribute that can uniquely identify a tuple in a relation is called the key of the table. The value of the attribute for different tuples in the relation has to be unique.

Example:

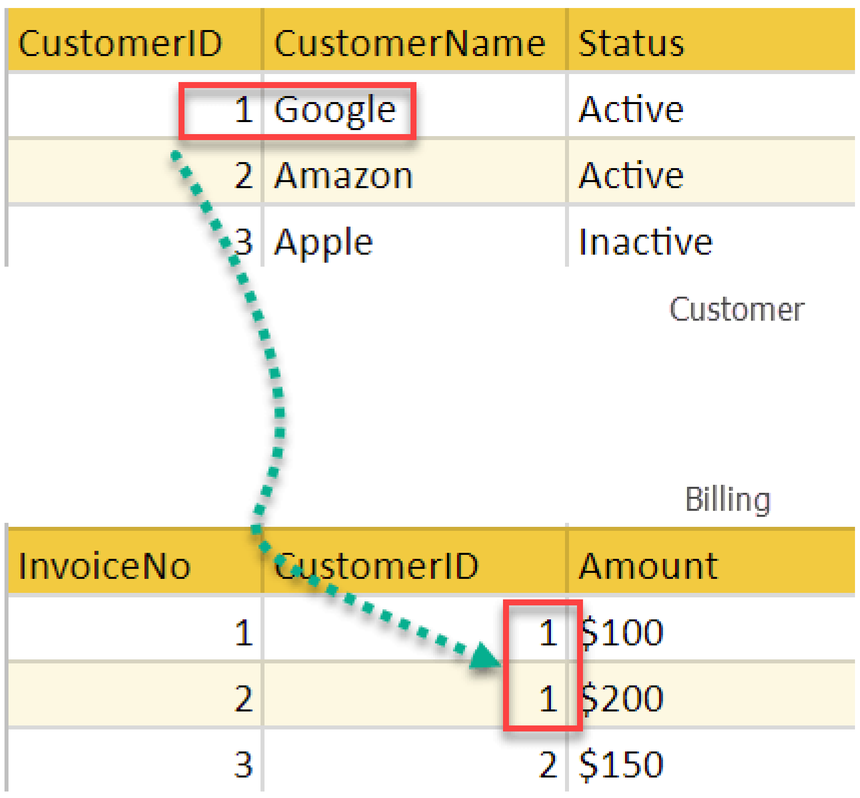
In the given table, CustomerID is a key attribute of Customer Table. It is most likely to have a single key for one customer, CustomerID =1 is only for the CustomerName =" Google".

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Referential integrity constraints

Referential integrity constraints is base on the concept of Foreign Keys. A foreign key is an important attribute of a relation which should be referred to in other relationships. Referential integrity constraint state happens where relation refers to a key attribute of a different or same relation. However, that key element must exist in the table.

Example:



Customer

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Billing

InvoiceNo	CustomerID	Amount
1	1	$100
2	1	$200
3	2	$150

In the above example, we have 2 relations, Customer and Billing.

Tuple for CustomerID =1 is referenced twice in the relation Billing. So we know CustomerName=Google has billing amount $300

Operations in Relational Model

Four basic update operations performed on relational database model are

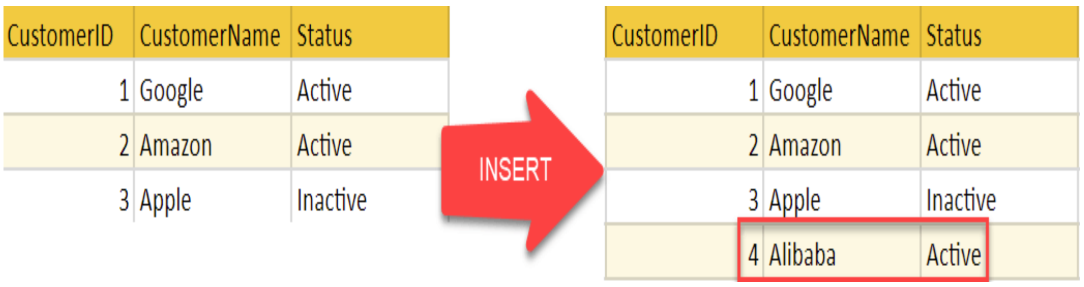
Insert, update, delete and select.

- Insert is used to insert data into the relation
- Delete is used to delete tuples from the table.
- Modify allows you to change the values of some attributes in existing tuples.
- Select allows you to choose a specific range of data.

Whenever one of these operations are applied, integrity constraints specified on the relational database schema must never be violated.

Insert Operation

The insert operation gives values of the attribute for a new tuple which should be inserted into a relation.

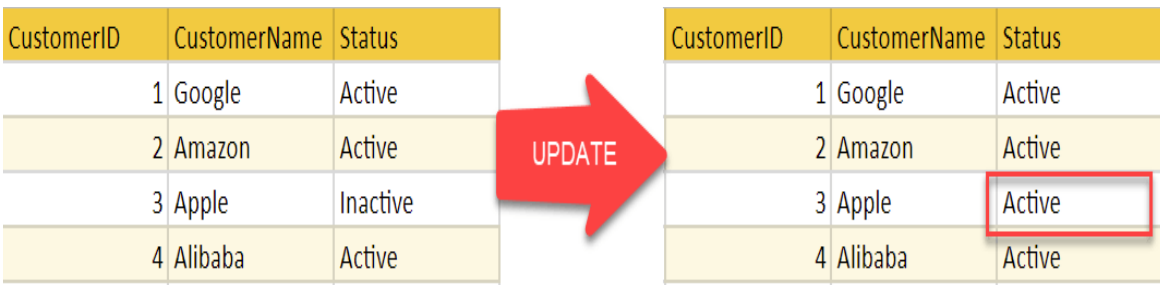


CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive
4	Alibaba	Active

Update Operation

You can see that in the below-given relation table CustomerName= 'Apple' is updated from Inactive to Active.

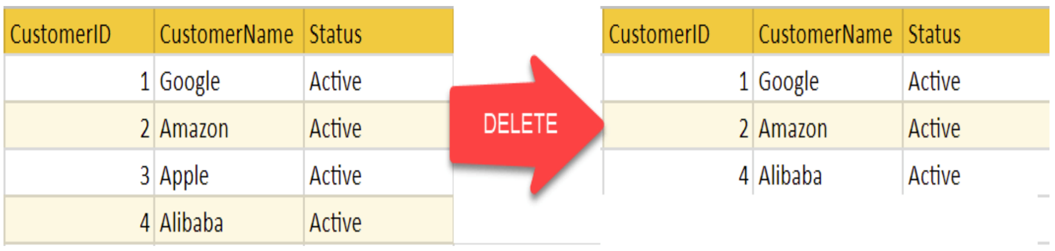


CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive
4	Alibaba	Active

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Active
4	Alibaba	Active

Delete Operation

To specify deletion, a condition on the attributes of the relation selects the tuple to be deleted.



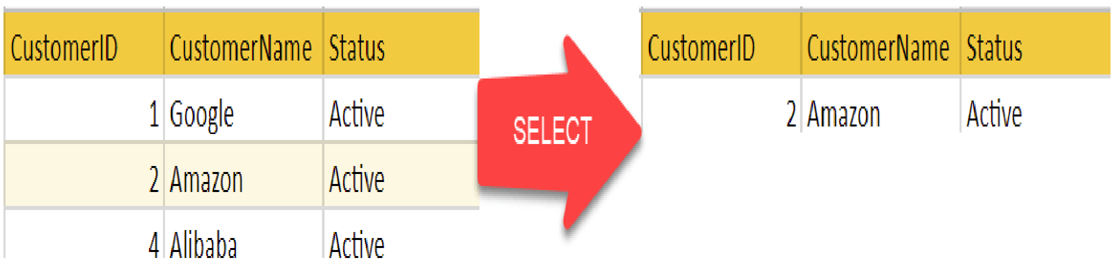
CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Active
4	Alibaba	Active

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
4	Alibaba	Active

In the above-given example, CustomerName= "Apple" is deleted from the table.

The Delete operation could violate referential integrity if the tuple which is deleted is referenced by foreign keys from other tuples in the same database.

Select Operation



CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
4	Alibaba	Active

CustomerID	CustomerName	Status
2	Amazon	Active

In the above-given example, CustomerName="Amazon" is selected

Best Practices for creating a Relational Model

- Data need to be represented as a collection of relations
- Each relation should be depicted clearly in the table
- Rows should contain data about instances of an entity
- Columns must contain data about attributes of the entity
- Cells of the table should hold a single value
- Each column should be given a unique name
- No two rows can be identical
- The values of an attribute should be from the same domain

Advantages of using Relational model

- Simplicity: A relational data model is simpler than the hierarchical and network model.
- Structural Independence: The relational database is only concerned with data and not with a structure. This can improve the performance of the model.
- Easy to use: The relational model is easy as tables consisting of rows and columns is quite natural and simple to understand
- Query capability: It makes possible for a high-level query language like SQL to avoid complex database navigation.
- Data independence: The structure of a database can be changed without having to change any application.
- Scalable: Regarding a number of records, or rows, and the number of fields, a database should be enlarged to enhance its usability.

Disadvantages of using Relational model

- Few relational databases have limits on field lengths which can't be exceeded.
- Relational databases can sometimes become complex as the amount of data grows, and the relations between pieces of data become more complicated.
- Complex relational database systems may lead to isolated databases where the information cannot be shared from one system to another.

Module – II

Relational Approach

Relational algebra and calculus

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

We will discuss all these operations in the following sections.

Select Operation (σ)

It selects tuples that satisfy the given predicate from a relation.

Notation − $\sigma_p(r)$

Where σ stands for selection predicate and r stands for relation. *p* is prepositional logic formula which may use connectors like and, or, and not. These terms may use relational operators like − $=, \neq, \geq, <$, $>$, $\leq$.

For example −

$\sigma_{subject = "database"}$(Books)

Output − Selects tuples from books where subject is 'database'.

$\sigma_{\text{subject = "database" and price = "450"}}$(Books)

Output − Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{\text{subject = "database" and price = "450" or year > "2010"}}$(Books)

Output − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

Project Operation (∏)

It projects column(s) that satisfy a given predicate.

Notation − $\prod_{A1}$, A_2, A_n (r)

Where A_1, A_2 , A_n are attribute names of relation r.

Duplicate rows are automatically eliminated, as relation is a set.

For example −

$\prod_{\text{subject, author}}$ (Books)

Selects and projects columns named as subject and author from the relation Books.

Set Operations:

Union Operation (∪)

It performs binary union between two given relations and is defined as −

r ∪ s = { t | t ∈ r or t ∈ s}

Notation − r U s

Where r and s are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold −

- r, and s must have the same number of attributes.
- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.

$\prod_{\text{author}}$ (Books) ∪ $\prod_{\text{author}}$ (Articles)

Output − Projects the names of the authors who have either written a book or an article or both.

Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

Notation − r − s

Finds all the tuples that are present in r but not in s.

$\prod_{\text{author}}$ (Books) − $\prod_{\text{author}}$ (Articles)

Output − Provides the name of authors who have written books but not articles.

Cartesian Product (X)

Combines information of two different relations into one.

Notation − r X s

Where r and s are relations and their output will be defined as −

r X s = { q t | q ∈ r and t ∈ s}

$\sigma_{\text{author = 'tutorialspoint'}}$(Books X Articles)

Output − Yields a relation, which shows all the books and articles written by tutorialspoint.

Rename Operation (ρ)

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter rho *ρ*.

Notation − ρ_{x} (E)

Where the result of expression E is saved with name of x.

Additional operations are −

- Set intersection
- Assignment
- Natural join

Join Operations

Join operation is essentially a cartesian product followed by a selection criterion.

Join operation denoted by ⋈.

JOIN operation also allows joining variously related tuples from different relations.

Types of JOIN:

Various forms of join operation are:

Inner Joins:

- Theta join
- EQUI join
- Natural join

Outer join:

- Left Outer Join
- Right Outer Join
- Full Outer Join

Inner Join:

In an inner join, only those tuples that satisfy the matching criteria are included, while the rest are excluded. Let's study various types of Inner Joins:

Theta Join:

The general case of JOIN operation is called a Theta join. It is denoted by symbol **θ**

Example

A $\bowtie_{\theta}$ B

Theta join can use any conditions in the selection criteria.

For example:

A ⋈ A.column 2 > B.column 2 (B)

A ⋈ A.column 2 > B.column 2 (B)	
column 1	column 2
1	2

EQUI join:

When a theta join uses only equivalence condition, it becomes a equi join.

For example:

A ⋈ A.column 2 = B.column 2 (B)

A ⋈ A.column 2 = B.column 2 (B)	
column 1	column 2
1	1

EQUI join is the most difficult operations to implement efficiently in an RDBMS and one reason why RDBMS have essential performance problems.

NATURAL JOIN (⋈)

Natural join can only be performed if there is a common attribute (column) between the relations. The name and type of the attribute must be same.

Example

Consider the following two tables

C	
Num	Square
2	4
3	9

D	
Num	Cube
2	8
3	18

C ⋈ D

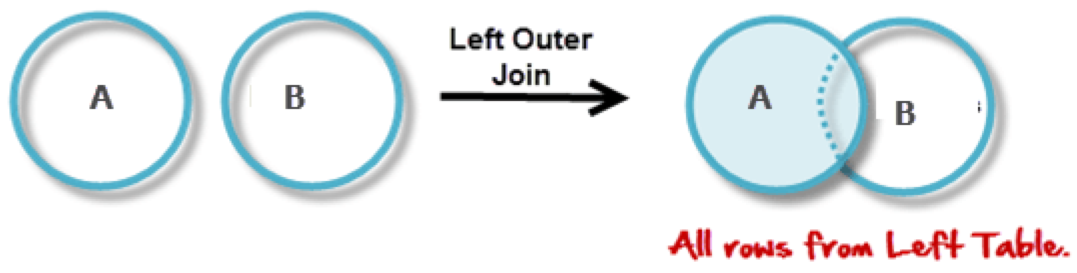
C ⋈ D		
Num	Square	Cube
2	4	4
3	9	9

OUTER JOIN

In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples that do not match the criteria.

Left Outer Join**(A ⟕ B)**

In the left outer join, operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.



Consider the following 2 Tables

A	
Num	Square
2	4
3	9
4	16

B	
Num	Cube
2	8
3	18
5	75

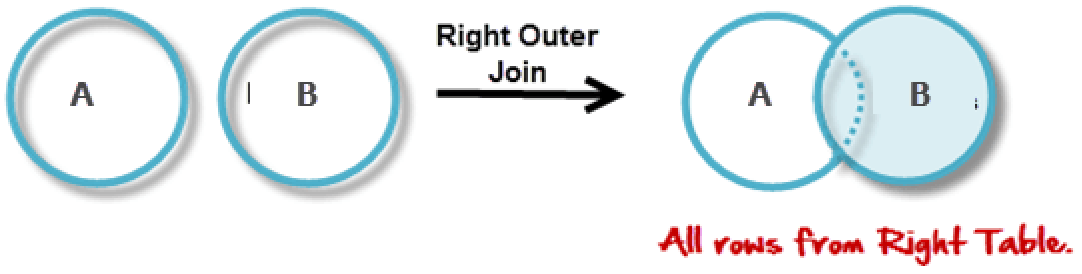
A ⟕ B

A ⋈ B		
Num	Square	Cube
2	4	4
3	9	9

4	16	-

Right Outer Join**: (A ⟖ B)**

In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.



A ⟖ B

A ⋈ B		
Num	Cube	Square
2	8	4
3	18	9
5	75	-

Full Outer Join**: (A ⟗ B)**

In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.

A ⟗ B

A ⋈ B		
Num	Cube	Square
2	4	8
3	9	18

4	16	-
5	-	75

Summary

Operation	**Purpose**
Select(σ)	The SELECT operation is used for selecting a subset of the tuples according to a given selection condition
Projection(π)	The projection eliminates all attributes of the input relation but those mentioned in the projection list.
Union Operation(∪)	UNION is symbolized by symbol. It includes all tuples that are in tables A or in B.
Set Difference(-)	- Symbol denotes it. The result of A - B, is a relation which includes all tuples that are in A but not in B.
Intersection(∩)	Intersection defines a relation consisting of a set of all tuple that are in both A and B.
Cartesian Product(X)	Cartesian operation is helpful to merge columns from two relations.
Inner Join	Inner join, includes only those tuples that satisfy the matching criteria.
Theta Join(θ)	The general case of JOIN operation is called a Theta join. It is denoted by symbol θ.
EQUI Join	When a theta join uses only equivalence condition, it becomes a equi join.
Natural Join(⋈)	Natural join can only be performed if there is a common attribute (column) between the relations.

Outer Join	In an outer join, along with tuples that satisfy the matching criteria.
Left Outer Join(⟕)	In the left outer join, operation allows keeping all tuple in the left relation.
Right Outer join(⟖)	In the right outer join, operation allows keeping all tuple in the right relation.
Full Outer Join(⟗)	In a full outer join, all tuples from both relations are included in the result irrespective of the matching condition.

Relational Calculus

In contrast to Relational Algebra, Relational Calculus is a non-procedural query language, that is, it tells what to do but never explains how to do it.

Relational calculus exists in two forms −

Tuple Relational Calculus (TRC)

In the tuple relational calculus, you will have to find tuples for which a predicate is true. The calculus is dependent on the use of tuple variables. A tuple variable is a variable that 'ranges over' a named relation: i.e., a variable whose only permitted values are tuples of the relation.

Example:
For example, to specify the range of a tuple variable S as the Staff relation, we write:

Staff(S)

To express the query 'Find the set of all tuples S such that F(S) is true,' we can write:

{S | F(S)}

Here, F is called a formula (well-formed formula, or wff in mathematical logic). For example, to express the query 'Find the staffNo, fName, lName, position, sex, DOB, salary, and branchNo of all staff earning more than £10,000', we can write:

{S | Staff(S) ∧ S.salary > 10000}

Example:

{t | TEACHER (t) and t.SALARY>20000}

- It implies that it selects the tuples from the TEACHER in such a way that the resulting teacher tuples will have the salary greater than 20000. This is an example of selecting a range of values.

{t | TEACHER (t) AND t.DEPT_ID = 6}

- T select all the tuples of teachers name who work under Department 8. Any tuple variable with 'For All' (?) or 'there exists' (?) condition is termed as a bound variable. In the last example, for any range of values of SALARY greater than 20000, the meaning of the condition does not alter. Bound variables are those ranges of tuple variables whose meaning will not alter if another tuple variable replaces the tuple variable.

In the second example, you have used DEPT_ID= 8 which means only for DEPT_ID = 8 display the teacher details. Such variable is called free variable. Any tuple variable without any 'For All' or 'there exists' condition is called Free Variable.

Domain Relational Calculus

In the tuple relational calculus, you have use variables that have series of tuples in a relation. In the domain relational calculus, you will also use variables, but in this case, the variables take their values from domains of attributes rather than tuples of relations. A domain relational calculus expression has the following general format:

{d1, d2, . . . , dn | F(d1, d2, . . . , dm)} $m \geq n$

where d1, d2, . . . , dn, . . . , dm stand for domain variables and F(d1, d2, . . . , dm) stands for a formula composed of atoms.

Example:
select TCHR_ID and TCHR_NAME of teachers who work for department 8, (where suppose - dept. 8 is Computer Application Department)

{<tchr_id, tchr_name=""> | <tchr_id, tchr_name=""> ? TEACHER Λ DEPT_ID = 10}

Get the name of the department name where Karlos works:

{DEPT_NAME |< DEPT_NAME > ? DEPT Λ ? DEPT_ID (? TEACHER Λ TCHR_NAME = Karlos)}

It is to be noted that these queries are safe. The use domain relational calculus is restricted to safe expressions; moreover, it is equivalent to the tuple relational calculus which in turn is similar to the relational algebra.

expressive power of algebra and calculus

We presented two formal query languages for the relational model. Are they equivalent in power? Can every query that can be expressed in relational algebra also be expressed in relational calculus? The answer is yes, it can.

Regarding expressiveness, we can show that every query that can be expressed using a *safe relational calculus query* can also be expressed as a relational algebra query. The expressive power of relational algebra is often used as a metric of how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be relationally complete. A practical query language is expected to be relationally complete; in addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

Module-III

SQL Query - Basics, RDBMS - Normalization

SQL

Structured Query Language(SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database. SQL uses certain commands like Create, Drop, Insert etc. to carry out the required tasks.

These SQL commands are mainly categorized into four categories as:

1. DDL – Data Definition Language
2. DQl – Data Query Language
3. DML – Data Manipulation Language
4. DCL – Data Control Language

Though many resources claim there to be another category of SQL clauses TCL – Transaction Control Language. So we will see in detail about TCL as well.

1. DDL(Data Definition Language) : DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Examples of DDL commands:

- CREATE – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- DROP – is used to delete objects from the database.
- ALTER-is used to alter the structure of the database.
- TRUNCATE–is used to remove all records from a table, including all spaces allocated for the records are removed.
- COMMENT –is used to add comments to the data dictionary.
- RENAME –is used to rename an object existing in the database.

2. DQL (Data Query Language) :

DML statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get some schema relation based on the query passed to it.

Example of DQL:

- SELECT – is used to retrieve data from the a database.

3. DML(Data Manipulation Language) : The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

Examples of DML:

- INSERT – is used to insert data into a table.
- UPDATE – is used to update existing data within a table.
- DELETE – is used to delete records from a database table.

4. DCL(Data Control Language) : DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.

Examples of DCL commands:

- GRANT-gives user's access privileges to database.
- REVOKE-withdraw user's access privileges given by using the GRANT command.

5. TCL(transaction Control Language) : TCL commands deals with the transaction within the database.

Examples of TCL commands:

- COMMIT– commits a Transaction.
- ROLLBACK– rollbacks a transaction in case of any error occurs.
- SAVEPOINT–sets a savepoint within a transaction.
- SET TRANSACTION–specify characteristics for the transaction.

Data definition commands

Data definition commands are used to create, modify and remove database objects such as schemas, tables, views, indexes etc.

Common Data Definition commands:

Create

The main use of create command is to create a new table in database. It has a predefined syntax in which we specify the columns and their respective data types.

General syntax:

CREATE TABLE <TABLE NAME>

(<COLUMN NAME> <DATA TYPE>,

<COLUMN NAME> <DATA TYPE>,

<COLUMN NAME> <DATA TYPE>,

<COLUMN NAME> <DATA TYPE>

);

Example: Create a student table with columns student name and roll number.

```
CREATE TABLE STUDENT

(STUDENT_NAME VARCHAR(30),

ROLL_NUMBER INT

);
```

Alter

An existing database object can be modified using the alter command. Alter command can do following changes to any table-

1. Add new columns.
2. Add new integrity constraints.
3. Modify existing columns.
4. Drop integrity constraints.

General Syntax of the ALTER command is mentioned below:

For adding a new column

ALTER TABLE <table_name> ADD <column_name>

For renaming a table

ALTER TABLE <table_name> RENAME To <new_table_name >

For modifying a column

ALTER TABLE <table_name> MODIFY <column_name > <data type >

For deleting a column

ALTER TABLE <table_name> DROP COLUMN <column_name>

Drop

This command can delete an index, table or view. Basically, any component from a relational database management system can be removed using the Drop command. Once the object is dropped, it cannot be reused.

The general syntax of drop command is as follows:

DROP TABLE <table_name>;

DROP DATABASE <database_name>;

DROP TABLE <index_name>;

Truncate

Using the truncate command, all the records in a database are deleted, but the database structure is maintained.

General syntax:

TRUNCATE TABLE <table name>

Comment

This command is used to add comments to the data dictionary.

General syntax:

1. Single line comments: use ‘ --‘ before any text.
2. Multiline comments: /* comments in between   */

Rename

The rename command renames an object

General Syntax:

Rename <old name> to <new name>

Data manipulation commands

Data manipulation commands are used to manipulate data in the database.

Some of the Data Manipulation Commands are-

Select

Select statement retrieves the data from database according to the constraints specifies alongside.

SELECT <COLUMN NAME>

FROM <TABLE NAME>

WHERE <CONDITION>

GROUP BY <COLUMN LIST>

HAVING <CRITERIA FOR FUNCTION RESULTS>

ORDER BY <COLUMN LIST>

General syntax:

Example: select * from employee where e_id>100;

Insert

Insert statement is used to insert data into database tables.

General Syntax:

INSERT INTO <TABLE NAME> (<COLUMNS TO INSERT>) VALUES (<VALUES TO INSERT>)

Example: insert into Employee (name, dept_id) values (‘ABC’, 3);

Update

The update command updates existing data within a table.

General syntax:

UPDATE <TABLE NAME>

SET <COLUMN NAME> = <UPDATED COLUMN VALUE>,

<COLUMN NAME> = <UPDATED COLUMN VALUE>,

<COLUMN NAME> = <UPDATED COLUMN VALUE>,…

WHERE <CONDITION>

Example: update Employee set Name=’AMIT’ where E_id=5;

Delete

Deletes records from the database table according to the given constraints.

General Syntax:

DELETE FROM <TABLE NAME>

WHERE <CONDITION>

Example:

delete from Employee where e_id=5;

To delete all records from the table:

Delete * from <TABLE NAME>;

Merge

Use the MERGE statement to select rows from one table for update or insertion into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause. It is also known as UPSERT i.e. combination of UPDATE and INSERT.

General Syntax (SQL):

MERGE <TARGET TABLE> [AS TARGET]

```
USING <SOURCE TABLE> [AS SOURCE]
ON <SEARCH CONDITION>
[WHEN MATCHED
THEN <MERGE MATCHED  > ]
[WHEN NOT MATCHED [BY TARGET]
THEN < MERGE NOT MATCHED >]
[WHEN NOT MATCHED BY SOURCE
THEN <MERGE MATCHED  >];
```

General Syntax (Oracle)

```
MERGE INTO <TARGET TABLE>
USING <SOURCE TABLE>
ON <SEARCH CONDITION>
[WHEN MATCHED
THEN <MERGE MATCHED > ]
[WHEN NOT MATCHED
THEN < MERGE NOT MATCHED > ];
```

views

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows −

```
CREATE VIEW view_name AS

SELECT column1, column2.....

FROM table_name

WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS

SELECT name, age

FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

- The SELECT clause may not contain the keyword DISTINCT.

- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW
   SET AGE = 35
   WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW

   WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+

| ID | NAME     | AGE | ADDRESS   | SALARY   |

+----+----------+-----+-----------+----------+

|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |

|  2 | Khilan   |  25 | Delhi     |  1500.00 |

|  3 | kaushik  |  23 | Kota      |  2000.00 |

|  4 | Chaitali |  25 | Mumbai    |  6500.00 |

|  5 | Hardik   |  27 | Bhopal    |  8500.00 |

|  7 | Muffy    |  24 | Indore    | 10000.00 |

+----+----------+-----+-----------+----------+
```

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

Joins

The SQL Joins clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables −

Table 1 − CUSTOMERS Table

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
   FROM CUSTOMERS, ORDERS
   WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL −

- INNER JOIN − returns rows when there is a match in both tables.
- LEFT JOIN − returns all rows from the left table, even if there are no matches in the right table.
- RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN − returns rows when there is a match in one of the tables.
- SELF JOIN − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- CARTESIAN JOIN − returns the Cartesian product of the sets of records from the two or more joined tables.

Integrity

Data integrity in the database is the correctness, consistency and completeness of data. Data integrity is enforced using the following three integrity constraints:

1. Entity Integrity - This is related to the concept of primary keys. All tables should have their own primary keys which should uniquely identify a row and not be NULL.
2. Referential Integrity - This is related to the concept of foreign keys. A foreign key is a key of a relation that is referred in another relation.
3. Domain Integrity - This means that there should be a defined domain for all the columns in a database.

DB2 database and functions can be managed by two different modes of security controls:

1. Authentication

2. Authorization

Authentication

Authentication is the process of confirming that a user logs in only in accordance with the rights to perform the activities he is authorized to perform. User authentication can be performed at operating system level or database level itself. By using authentication tools for biometrics such as retina and figure prints are in use to keep the database from hackers or malicious users.

The database security can be managed from outside the db2 database system. Here are some type of security authentication process:

- Based on Operating System authentications.
- Lightweight Directory Access Protocol (LDAP)

For DB2, the security service is a part of operating system as a separate product. For Authentication, it requires two different credentials, those are userid or username, and password.

Authorization

You can access the DB2 Database and its functionality within the DB2 database system, which is managed by the DB2 Database manager. Authorization is a process managed by the DB2 Database manager. The manager obtains information about the current authenticated user, that indicates which database operation the user can perform or access.

Here are different ways of permissions available for authorization:

Primary permission: Grants the authorization ID directly.

Secondary permission: Grants to the groups and roles if the user is a member

Public permission: Grants to all users publicly.

Context-sensitive permission: Grants to the trusted context role.

Authorization can be given to users based on the categories below:

- System-level authorization
- System administrator [SYSADM]
- System Control [SYSCTRL]
- System maintenance [SYSMAINT]
- System monitor [SYSMON]

Authorities provide of control over instance-level functionality. Authority provide to group privileges, to control maintenance and authority operations. For instance, database and database objects.

- Database-level authorization
- Security Administrator [SECADM]
- Database Administrator [DBADM]

- Access Control [ACCESSCTRL]
- Data access [DATAACCESS]
- SQL administrator. [SQLADM]
- Workload management administrator [WLMADM]
- Explain [EXPLAIN]

Authorities provide controls within the database. Other authorities for database include with LDAD and CONNECT.

- Object-Level Authorization: Object-Level authorization involves verifying privileges when an operation is performed on an object.
- Content-based Authorization: User can have read and write access to individual rows and columns on a particular table using Label-based access Control [LBAC].

Security

DB2 tables and configuration files are used to record the permissions associated with authorization names. When a user tries to access the data, the recorded permissions verify the following permissions:

- Authorization name of the user
- Which group belongs to the user
- Which roles are granted directly to the user or indirectly to a group
- Permissions acquired through a trusted context.

While working with the SQL statements, the DB2 authorization model considers the combination of the following permissions:

- Permissions granted to the primary authorization ID associated with the SQL statements.
- Secondary authorization IDs associated with the SQL statements.
- Granted to PUBLIC
- Granted to the trusted context role.

Instance level authorities

Let us discuss some instance related authorities.

System administration authority (SYSADM)

It is highest level administrative authority at the instance-level. Users with SYSADM authority can execute some databases and database manager commands within the instance. Users with SYSADM authority can perform the following operations:

- Upgrade a Database
- Restore a Database
- Update Database manager configuration file.

System control authority (SYSCTRL)

It is the highest level in System control authority. It provides to perform maintenance and utility operations against the database manager instance and its databases. These operations can affect system resources, but they do not allow direct access to data in the database.

Users with SYSCTRL authority can perform the following actions:

- Updating the database, Node, or Distributed Connect Service (DCS) directory
- Forcing users off the system-level
- Creating or Dropping a database-level
- Creating, altering, or dropping a table space
- Using any table space
- Restoring Database

System maintenance authority (SYSMAINT)

It is a second level of system control authority. It provides to perform maintenance and utility operations against the database manager instance and its databases. These operations affect the system resources without allowing direct access to data in the database. This authority is designed for users to maintain databases within a database manager instance that contains sensitive data.

Only Users with SYSMAINT or higher level system authorities can perform the following tasks:

- Taking backup
- Restoring the backup
- Roll forward recovery
- Starting or stopping instance
- Restoring tablespaces
- Executing db2trc command
- Taking system monitor snapshots in case of an Instance level user or a database level user.

A user with SYSMAINT can perform the following tasks:

- Query the state of a tablespace
- Updating log history files

- Reorganizing of tables
- Using RUNSTATS (Collection catalog statistics)

System monitor authority (SYSMON)

With this authority, the user can monitor or take snapshots of database manager instance or its database. SYSMON authority enables the user to run the following tasks:

- GET DATABASE MANAGER MONITOR SWITCHES
- GET MONITOR SWITCHES
- GET SNAPSHOT
- LIST
- RESET MONITOR
- UPDATE MONITOR SWITCHES

Database authorities

Each database authority holds the authorization ID to perform some action on the database. These database authorities are different from privileges. Here is the list of some database authorities:

- ACCESSCTRL: allows to grant and revoke all object privileges and database authorities.
- BINDADD: Allows to create a new package in the database.
- CONNECT: Allows to connect to the database.
- CREATETAB: Allows to create new tables in the database.
- CREATE_EXTERNAL_ROUTINE: Allows to create a procedure to be used by applications and the users of the databases.
- DATAACCESS: Allows to access data stored in the database tables.
- DBADM: Act as a database administrator. It gives all other database authorities except ACCESSCTRL, DATAACCESS, and SECADM.
- EXPLAIN: Allows to explain query plans without requiring them to hold the privileges to access the data in the tables.
- IMPLICIT_SCHEMA: Allows a user to create a schema implicitly by creating an object using a CREATE statement.
- LOAD: Allows to load data into table.
- QUIESCE_CONNECT: Allows to access the database while it is quiesce (temporarily disabled).
- SECADM: Allows to act as a security administrator for the database.

- SQLADM: Allows to monitor and tune SQL statements.
- WLMADM: Allows to act as a workload administrator
- Privileges
- SETSESSIONUSER

Authorization ID privileges involve actions on authorization IDs. There is only one privilege, called the SETSESSIONUSER privilege. It can be granted to user or a group and it allows to session user to switch identities to any of the authorization IDs on which the privileges are granted. This privilege is granted by user SECADM authority.

Schema privileges

This privileges involve actions on schema in the database. The owner of the schema has all the permissions to manipulate the schema objects like tables, views, indexes, packages, data types, functions, triggers, procedures and aliases. A user, a group, a role, or PUBLIC can be granted any user of the following privileges:

- CREATEIN: allows to create objects within the schema
- ALTERIN: allows to modify objects within the schema.
- DROPIN

This allows to delete the objects within the schema.

Tablespace privileges

These privileges involve actions on the tablespaces in the database. User can be granted the USE privilege for the tablespaces. The privileges then allow them to create tables within tablespaces. The privilege owner can grant the USE privilege with the command WITH GRANT OPTION on the tablespace when tablespace is created. And SECADM or ACCESSCTRL authorities have the permissions to USE privileges on the tablespace.

Table and view privileges

The user must have CONNECT authority on the database to be able to use table and view privileges. The privileges for tables and views are as given below:

CONTROL

It provides all the privileges for a table or a view including drop and grant, revoke individual table privileges to the user.

ALTER

It allows user to modify a table.

DELETE

It allows the user to delete rows from the table or view.

INDEX

It allows the user to insert a row into table or view. It can also run import utility.

REFERENCES

It allows the users to create and drop a foreign key.

SELECT

It allows the user to retrieve rows from a table or view.

UPDATE

It allows the user to change entries in a table, view.

Package privileges

User must have CONNECT authority to the database. Package is a database object that contains the information of database manager to access data in the most efficient way for a particular application.

CONTROL

It provides the user with privileges of rebinding, dropping or executing packages. A user with this privileges is granted to BIND and EXECUTE privileges.

BIND

It allows the user to bind or rebind that package.

EXECUTE

Allows to execute a package.

Index privileges

This privilege automatically receives CONTROL privilege on the index.

Sequence privileges

Sequence automatically receives the USAGE and ALTER privileges on the sequence.

Routine privileges

It involves the action of routines such as functions, procedures, and methods within a database.

Pitfalls in Relational-Database Design

Obviously, we can have good and bad designs. Among the undesirable design items are:

- Repetition of information
- Inability to represent certain information

The relation *lending* with the schema is an example of a bad design:

Lending-Schema=(branch-name, branch-city, assets, cutomer-name, loan-number, amount)

branch-name	branch-city	assets	customer-name	loan-number	amount
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-23	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Looking at the Downtown and Perryridge, when a new loan is added, the branch-city and assets must be repeated. That makes updating the table more difficult, because the update must guarantee that all tuples are updated. Additional problems come from having two people take out one loan (L-23). More complexity is involved when Jones took out a loan at a second branch (maybe one near home and the other near work.) Notice that there is no way to represent information on a branch unless there is a loan.

Decomposition

The obvious solution is that we should decompose this relation. As an alternative design, we can use the Decomposition rule: If *A* implies *BC* then *A* implies *B* and *A* implies *C*.

This gives us the schemas:

- *branch-customer-schema = (branch-name, branch-city, assets, customer-name)*
- *customer-loan-schema = (customer-name, loan-number, amount)*

branch-name	branch-city	assets	customer-name
Downtown	Brooklyn	9000000	Jones

Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

customer-name	loan-number	amount
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-23	2000
Glenn	L-25	2500
Brooklyn	L-10	2200

Then when we need to get back to the original table, we can do a natural join on the two relations *branch-customer* and *customer-loan.*

Evaluating this design, how does it compare to the first version?

- Looking at the Downtown and Perryridge, when a new loan is added, the branch-city and assets must be repeated. Problem still exists.
- Problems come from having two people take out one loan (L-23). Problem still exists.
- More complexity is involved when Jones took out a loan at a second branch. Problem still exists.
- Notice that there is no way to represent information on a branch unless there is a loan. Problem still exists.

Worse, there is a new problem! When we do the natural join, we get back four additional tuples that did not exists in the original table:

- (Downtown, Brooklyn, 9000000, Jones, L-93, 500)
- (Perryridge, Horseneck, 1700000, Hayes, L-16, 1300)
- (Mianus, Horseneck, 400000, Jones, L-17, 1000)
- (North Town, Rye, 3700000, Hayes, L-15, 1500)

We are no long able to represent in the database information about which customers are borrows from which branch. This is called a *lossy decomposition* or *lossy-join decomposition*. A decomposition that is not a lossy-decomposition is a *lossless-join decomposition.* Lossless-joins are a requirement for good design and this causes constraints on the set of possible relations. We say that a relation is *legal* if it satisfies all rules, or constraints, imposed.

The proper way to decomposition this example so that we can have a lossless-join is to use three relations.

- *branch-schema = (<u>branch-name</u>, assets, branch-city)*
- *loan-schema = (branch-name, <u>loan-number</u>, amount)*
- *borrower-schema = (<u>customer-name</u>, loan-number)*

Lossless Join Decomposition:

- The lossless join property is a feature of decomposition supported by normalization. It is the ability to ensure that any instance of the original relation can be identified from corresponding instances in the smaller relations.

R : relation, F : set of functional dependencies on R,

X,Y : decomposition of R

- A decomposition {R1, R2,…, Rn} of a relation R is called a lossless decomposition for R if the natural join of R1, R2,…, Rn produces exactly the relation R.

- A decomposition is lossless if we can recover:

R(A, B, C) -> Decompose -> R1(A, B) R2(A, C) -> Recover -> R'(A, B, C)

Thus,R' = R

- Decomposition is lossles if :

X ∩ Y -> X, that is: all attributes common to both X and Y functionally determine ALL the attributes in X.

X ∩ Y -> Y, that is: all attributes common to both X and Y functionally determine ALL the attributes in Y

If X ∩ Y forms a superkey of either X or Y, the decomposition of R is a lossless decomposition.

Dependency Preserving Decomposition:

- A decomposition D = {R1, R2, ..., Rn} of R is dependency-preserving with respect to F if the union of the projections of F on each Ri in D is equivalent to F;

if (F1∪ F2 ∪ …∪Fn)+ = F +

- Example-

R= (A, B, C)

F = {A ->B, B->C}

Key = {A}

Ris not in BCNF

Decomposition R1 = (A, B), R2 = (B, C)

R1 and R2 are in BCNF, Lossless-join decomposition, Dependency preserving

- Each Functional Dependency specified in F either appears directly in one of the relations in the decomposition.

- It is not necessary that all dependencies from the relation R appear in some relation Ri.

- It is sufficient that the union of the dependencies on all the relations Ri be equivalent to the dependencies on R.

- is lost in the decomposition

Functional dependencies

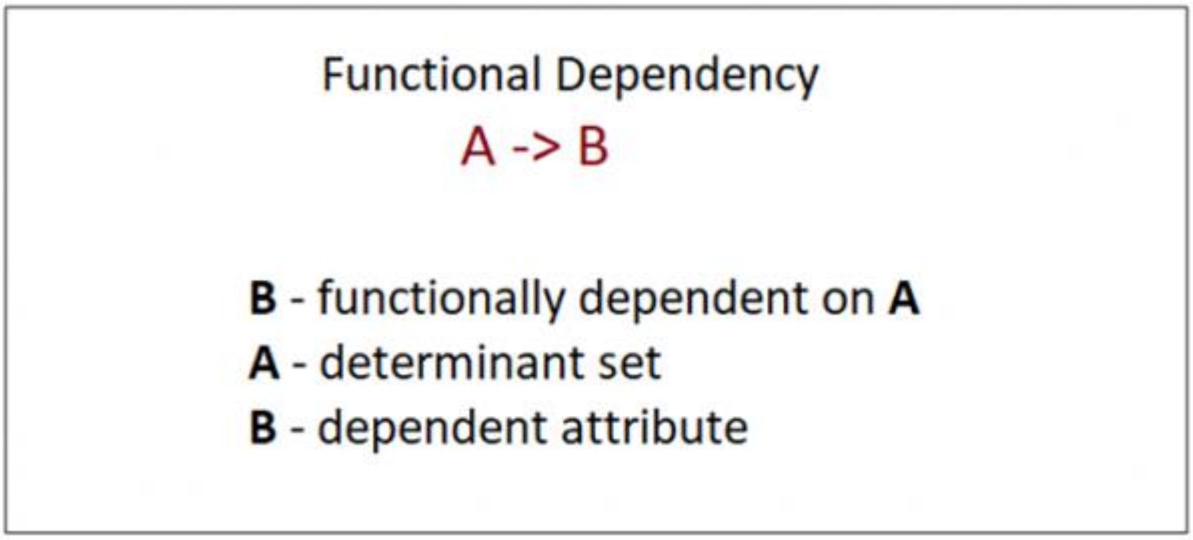
Functional dependency in DBMS, as the name suggests is a relationship between attributes of a table dependent on each other. Introduced by E. F. Codd, it helps in preventing data redundancy and gets to know about bad designs.

To understand the concept thoroughly, let us consider P is a relation with attributes A and B. Functional Dependency is represented by -> (arrow sign)

Then the following will represent the functional dependency between attributes with an arrow sign:

A -> B

Above suggests the following:


Functional Dependency

A -> B

B - functionally dependent on **A**

A - determinant set

B - dependent attribute

Example

The following is an example that would make it easier to understand functional dependency:

We have a <Department> table with two attributes: DeptId and DeptName.

DeptId = Department ID
DeptName = Department Name

The DeptId is our primary key. Here, DeptId uniquely identifies the DeptName attribute. This is because if you want to know the department name, then at first you need to have the DeptId.

DeptId	DeptName
001	Finance
002	Marketing
003	HR

Therefore, the above functional dependency between DeptId and DeptName can be determined as DeptId is functionally dependent on DeptName:

DeptId -> DeptName

Types of Functional Dependency

Functional Dependency has three forms:

1. Trivial Functional Dependency
2. Non-Trivial Functional Dependency
3. Completely Non-Trivial Functional Dependency

Let us begin with Trivial Functional Dependency:

Trivial Functional Dependency

It occurs when B is a subset of A in:

```
A ->B
```

Example

We are considering the same <Department> table with two attributes to understand the concept of trivial dependency.

The following is a trivial functional dependency since DeptId is a subset of DeptId and DeptName

```
{ DeptId,  DeptName } -> Dept Id
```

Non –Trivial Functional Dependency

It occurs when B is not a subset of A in:

```
A ->B
```

Example

```
DeptId ->  DeptName
```

The above is a non-trivial functional dependency since DeptName is a not a subset of DeptId.

Completely Non - Trivial Functional Dependency

It occurs when A intersection B is null in:

```
A ->B
```

Armstrong Axioms

Armstrong's Axioms Property of Functional Dependency

Armstrong's Axioms property was developed by William Armstrong in 1974 to reason about functional dependencies.

The property suggests rules that hold true if the following are satisfied:

1. Transitivity
   If A->B and B->C, then A->C i.e. a transitive relation.

2. Reflexivity
   A-> B, if B is a subset of A.

3. Augmentation
   The last rule suggests: AC->BC, if A->B

If F is a set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

- Reflexive rule − If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.

- Augmentation rule − If a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.

- Transitivity rule − Same as transitive rule in algebra, if a → b holds and b → c holds, then a → c also holds. a → b is called as a functionally that determines b.

Normalization

If a database design is not perfect, it may contain anomalies, which are like a bad dream for any database administrator. Managing a database with anomalies is next to impossible.

- Update anomalies − If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.

- Deletion anomalies − We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.

- Insert anomalies − We tried to insert data in a record that does not exist at all.

Normalization is a method to remove all these anomalies and bring the database to a consistent state.

First Normal Form

First Normal Form is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains. The values in an atomic domain are indivisible units.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

We re-arrange the relation (table) as below, to convert it to First Normal Form.

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

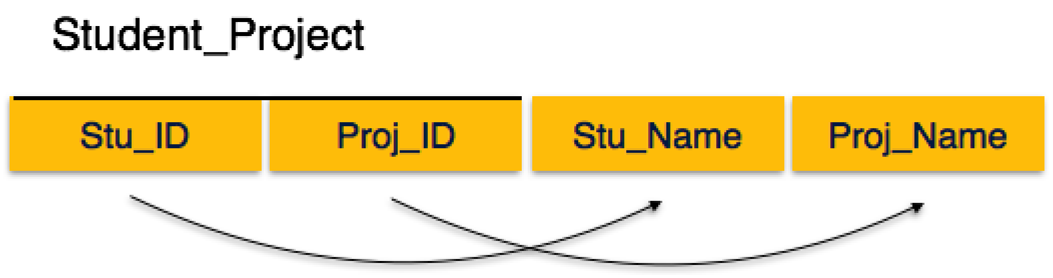
Each attribute must contain only a single value from its pre-defined domain.

Second Normal Form

Before we learn about the second normal form, we need to understand the following −

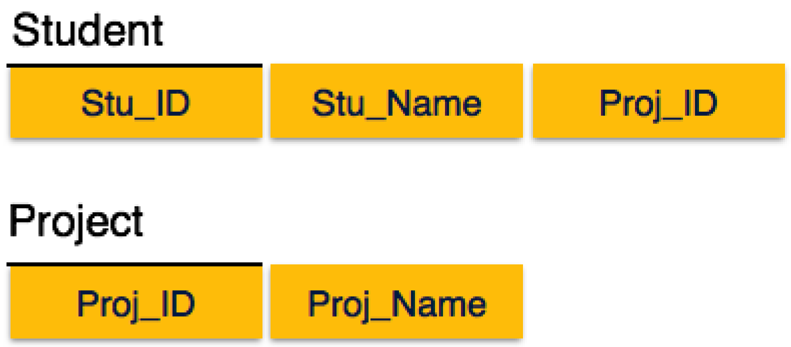
- Prime attribute − An attribute, which is a part of the candidate-key, is known as a prime attribute.
- Non-prime attribute − An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.

If we follow second normal form, then every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if X → A holds, then there should not be any proper subset Y of X, for which Y → A also holds true.



We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by

Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called partial dependency, which is not allowed in Second Normal Form.

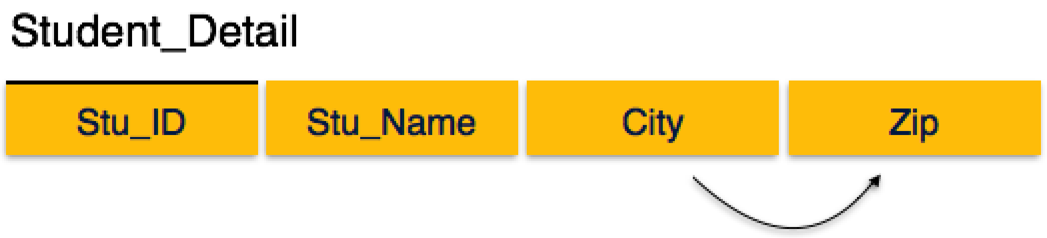


We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

Third Normal Form

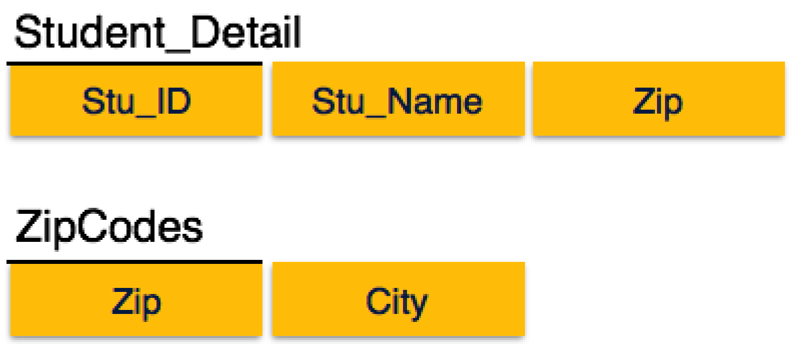
For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy –

- No non-prime attribute is transitively dependent on prime key attribute.
- For any non-trivial functional dependency, X → A, then either –
    - X is a superkey or,
    - A is prime attribute.



We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu_ID → Zip → City, so there exists transitive dependency.

To bring this relation into third normal form, we break the relation into two relations as follows –



Boyce-Codd Normal Form

Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms. BCNF states that −

- For any non-trivial functional dependency, X → A, X must be a super-key.

In the above image, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So,

Stu_ID → Stu_Name, Zip

and

Zip → City

Which confirms that both the relations are in BCNF.

Multi-valued dependency

When existence of one or more rows in a table implies one or more other rows in the same table, then the Multi-valued dependencies occur.

If a table has attributes P, Q and R, then Q and R are multi-valued facts of P.

It is represented by double arrow:

```
->->
```

For our example:

```
P->->Q
P->->R
```

In the above case, Multivalued Dependency exists only if Q and R are independent attributes.

A table with multivalued dependency violates the 4NF.

Example

Let us see an example:

<Student>

StudentName	CourseDiscipline	Activities
Amit	Mathematics	Singing
Amit	Mathematics	Dancing
Yuvraj	Computers	Cricket
Akash	Literature	Dancing
Akash	Literature	Cricket
Akash	Literature	Singing

In the above table, we can see Students Amit and Akash have interest in more than one activity.

This is multivalued dependency because CourseDiscipline of a student are independent of Activities, but are dependent on the student.

Therefore, multivalued dependency:

StudentName ->-> CourseDiscipline StudentName ->-> Activities

The above relation violates Fourth Normal Form in Normalization.

To correct it, divide the table into two separate tables and break Multivalued Dependency:

<StudentCourse>

StudentName	CourseDiscipline
Amit	Mathematics
Amit	Mathematics
Yuvraj	Computers
Akash	Literature
Akash	Literature
Akash	Literature

<StudentActivities>

StudentName	Activities
Amit	Singing
Amit	Dancing
Yuvraj	Cricket
Akash	Dancing
Akash	Cricket
Akash	Singing

This breaks the multivalued dependency and now we have two functional dependencies:

StudentName -> CourseDiscipline StudentName - > Activities

Join Dependency

If a table can be recreated by joining multiple tables and each of this table have a subset of the attributes of the table, then the table is in Join Dependency. It is a generalization of Multivalued Dependency

Join Dependency can be related to 5NF, wherein a relation is in 5NF, only if it is already in 4NF and it cannot be decomposed further.

Example

<Employee>

EmpName	EmpSkills	EmpJob (Assigned Work)
Tom	Networking	EJ001
Harry	Web Development	EJ002
Katie	Programming	EJ002

The above table can be decomposed into the following three tables; therefore it is not in 5NF:

<EmployeeSkills>

EmpName	EmpSkills
Tom	Networking

Harry	Web Development
Katie	Programming

<EmployeeJob>

EmpName	EmpJob
Tom	EJ001
Harry	EJ002
Katie	EJ002

<JobSkills>

EmpSkills	EmpJob
Networking	EJ001
Web Development	EJ002
Programming	EJ002

Our Join Dependency:

{(EmpName, EmpSkills), (EmpName, EmpJob), (EmpSkills, EmpJob)}

The above relations have join dependency, so they are not in 5NF. That would mean that a join relation of the above three relations is equal to our original relation <Employee>.

4NF

The 4NF comes after 1NF, 2NF, 3NF, and Boyce-Codd Normal Form. It was introduced by Ronald Fagin in 1977.

To be in 4NF, a relation should be in Bouce-Codd Normal Form and may not contain more than one multi-valued attribute.

Example

Let us see an example:

<Movie>

Movie_Name	Shooting_Location	Listing

MovieOne	UK	Comedy
MovieOne	UK	Thriller
MovieTwo	Australia	Action
MovieTwo	Australia	Crime
MovieThree	India	Drama

The above is not in 4NF, since

1. More than one movie can have the same listing
2. Many shooting locations can have the same movie

Let us convert the above table in 4NF:

<Movie_Shooting>

Movie_Name	Shooting_Location
MovieOne	UK
MovieOne	UK
MovieTwo	Australia
MovieTwo	Australia
MovieThree	India

<Movie_Listing>

Movie_Name	Listing
MovieOne	Comedy
MovieOne	Thriller
MovieTwo	Action
MovieTwo	Crime
MovieThree	Drama

Now the violation is removed and the tables are in 4NF.

5NF

The 5NF (Fifth Normal Form) is also known as project-join normal form. A relation is in Fifth Normal Form (5NF), if it is in 4NF, and won’t have lossless decomposition into smaller tables.

You can also consider that a relation is in 5NF, if the candidate key implies every join dependency in it.

Example

The below relation violates the Fifth Normal Form (5NF) of Normalization:

<Employee>

EmpName	EmpSkills	EmpJob (Assigned Work)
David	Java	E145
John	JavaScript	E146
Jamie	jQuery	E146
Emma	Java	E147

The above relation can be decomposed into the following three tables; therefore, it is not in 5NF:

<EmployeeSkills>

EmpName	EmpSkills
David	Java
John	JavaScript
Jamie	jQuery
Emma	Java

The following is the <EmployeeJob> relation that displays the jobs assigned to each employee:

<EmployeeJob>

EmpName	EmpJob
David	E145
John	E146
Jamie	E146

Emma	E147

Here is the skills that are related to the assigned jobs:

<JobSkills>

EmpSkills	EmpJob
Java	E145
JavaScript	E146
jQuery	E146
Java	E147

Our Join Dependency:

{(EmpName, EmpSkills), (EmpName, EmpJob), (EmpSkills, EmpJob)}

The above relations have join dependency, so they are not in 5NF. That would mean that a join relation of the above three relations is equal to our original relation <Employee>.

Module –IV

Transaction Management

TRANSACTION CONCEPT

A Transaction is a *unit* of program execution that accesses and possibly updates various data items.

Example transaction to transfer $50 from account A to account B:

1. read(*A*)
2. *A* := *A* – 50
3. write(*A*)
4. read(*B*)
5. *B* := *B* + 50
6. write(*B)*

Two main issues to deal with:

Failures of various kinds, such as hardware failures and system crashes

Concurrent execution of multiple transactions

Example of Fund Transfer Transaction to transfer $50 from account A to account B:

1. read(*A*)
2. *A* := *A* – 50
3. write(*A*)
4. read(*B*)
5. *B* := *B* + 50
6. write(*B)*

Atomicity requirement

if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

Failure could be due to software or hardware the system should ensure that updates of a partially executed transaction are not reflected in the database.

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer Transaction to transfer $50 from account A to account B:

1. read(*A*)
2. *A* := *A* – 50
3. write(*A*)
4. read(*B*)
5. *B* := *B* + 50
6. write(*B)*

Consistency requirement in above example: the sum of A and B is unchanged by the execution of the transaction In general, consistency requirements include Explicitly specified integrity constraints such as primary keys and foreign keys Implicit integrity constraints Example sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand A transaction must see a consistent database. During transaction execution the database may be temporarily inconsistent. When the transaction completes successfully the database must be consistent Erroneous transaction logic can lead to inconsistency.

Example of Fund Transfer Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1 T2

1. read(*A*)
2. *A* := *A* – 50
3. write(*A*)
   read(A), read(B), print(A+B)
4. read(*B*)
5. *B* := *B* + 50
6. write(*B*)

Isolation can be ensured trivially by running transactions serially that is, one after the other.

However, executing multiple transactions concurrently has significant benefits.

ACID Properties

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

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

Isolation. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

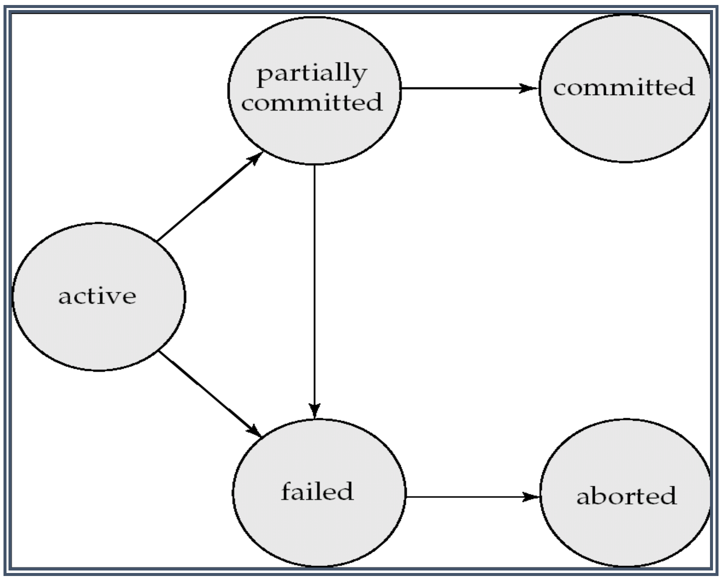
Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- 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 restored to its

State prior to the start of the transaction. Two options after it has been aborted: restart the transaction can be done only if no internal logical error kill the transaction

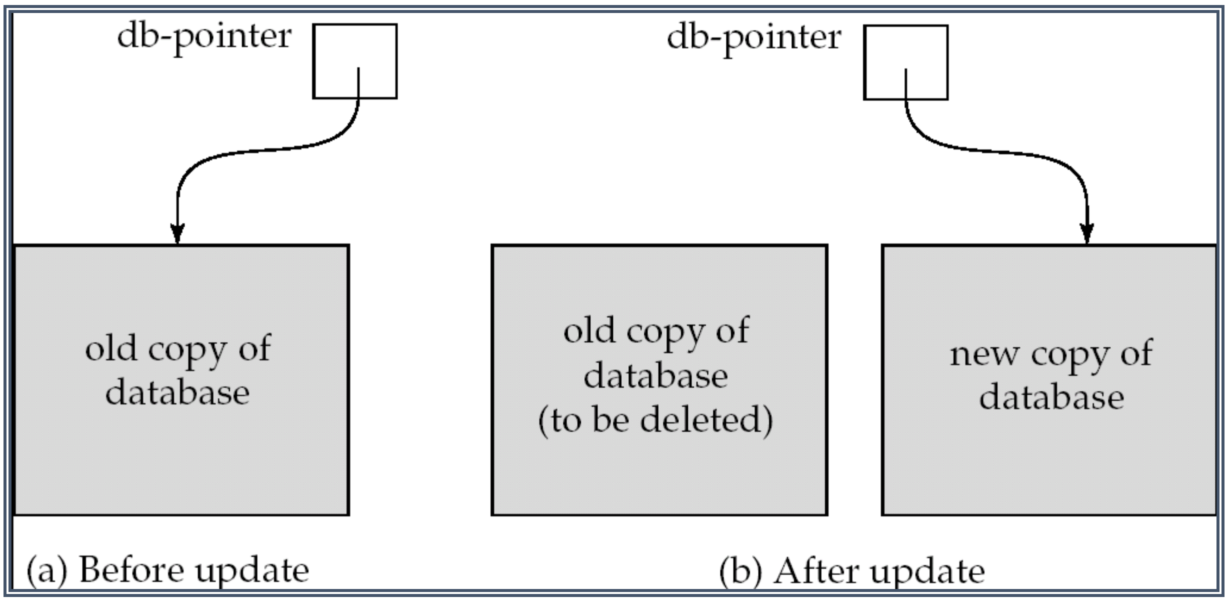
- Committed – after successful completion.



Implementation Of Atomicity And Durability

The recovery-management component of a database system implements the support for atomicity and durability. Example of the *shadow-database* scheme:all updates are made on a *shadow copy* of the database db_pointer is made to point to the updated shadow copy after the transaction reaches partial commit and all updated pages have been flushed to disk.

db_pointer always points to the current consistent copy of the database.In case transaction fails, old consistent copy pointed to by db_pointer can be used, and the shadow copy can be deleted.



The shadow-database scheme: Assumes that only one transaction is active at a time. Assumes disks do not fail Useful for text editors, but extremely inefficient for large databases (why?)

Variant called shadow paging reduces copying of data, but is still not practical for large databases does not handle concurrent transactions

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are: increased processor and disk utilization, leading to better transaction throughput.

Example one transaction can be using the CPU while another is reading from or writing to the disk reduced average response time for transactions: short transactions need not wait behind long ones Concurrency control schemes – mechanisms to achieve isolation that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

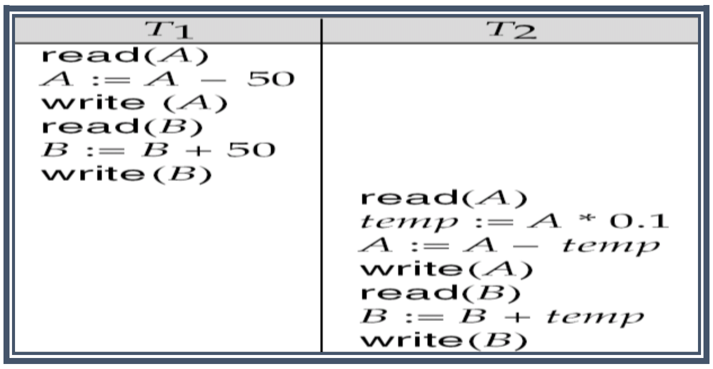
Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed a schedule for a set of transactions must consist of all instructions of those transactions must preserve the order in which the instructions appear in each individual transaction.

A transaction that successfully completes its execution will have commit instructions as the last statement by default transaction assumed to execute commit instruction as its last step

A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

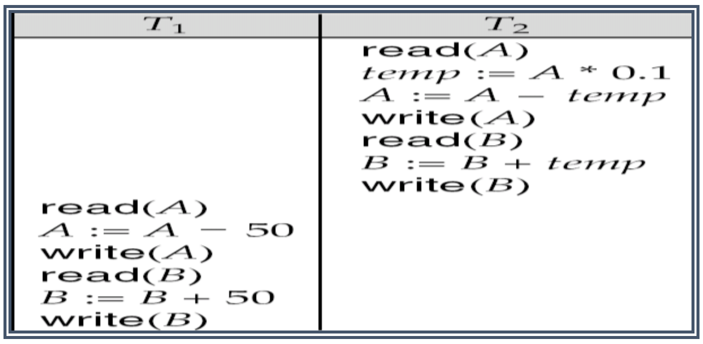
Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :



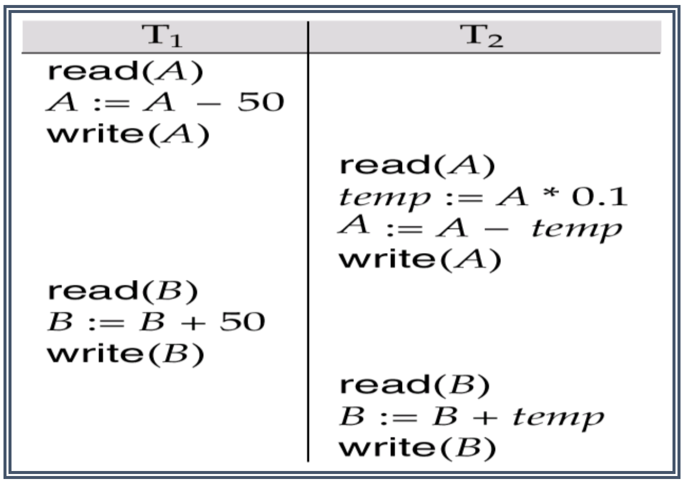
T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write (B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2



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

Schedule 3



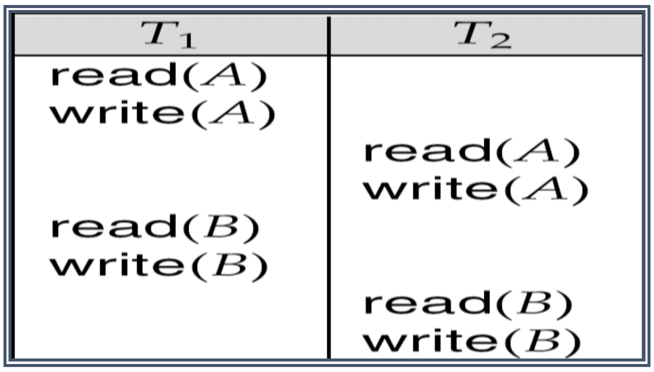
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Let T_1 and T_2 be the transactions defined previously.The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

Serializability

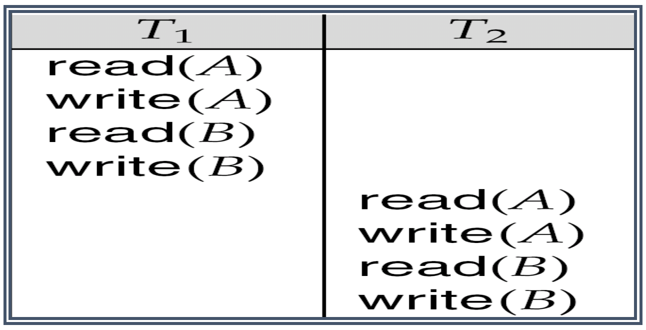
Basic Assumption – Each transaction preserves database consistency.Thus serial execution of a set of transactions preserves database consistency.A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. conflict serializability



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

2.view serializability



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

Simplified view of transactions We ignore operations other than read and write instructions; We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only read and write instructions. Conflicting Instructions Instructions l_i and l_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

1. l_i = read(Q), l_j = read(Q). l_i and l_j don't conflict.
2. l_i = read(Q), l_j = write(Q). They conflict.
3. l_i = write(Q), l_j = read(Q). They conflict
4. l_i = write(Q), l_j = write(Q). They conflict

Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are conflict equivalent.

We say that a schedule *S* is conflict serializable if it is conflict equivalent to a serial schedule

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

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.Therefore Schedule 3 is conflict serializable.

Example of a schedule that is not conflict serializable:We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are view equivalent if the following three conditions are met, for each data item *Q*,If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.

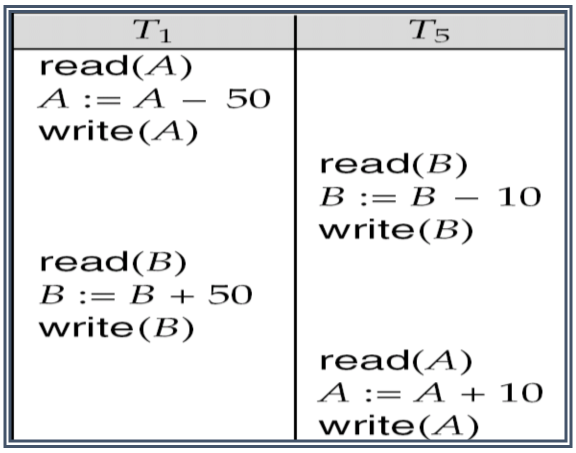
If in schedule S transaction T_i executes read(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same write(Q) operation of transaction T_j .The transaction (if any) that performs the final write(*Q*) operation in schedule *S* must also perform the final write(*Q*) operation in schedule *S'*. As can be seen, view equivalence is also based purely on reads and writes alone.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

A schedule *S* is view serializable if it is view equivalent to a serial schedule.Every conflict serializable schedule is also view serializable.Below is a schedule which is view-serializable but *not* conflict serializable.

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has blind writes.

Other Notions of Serializability

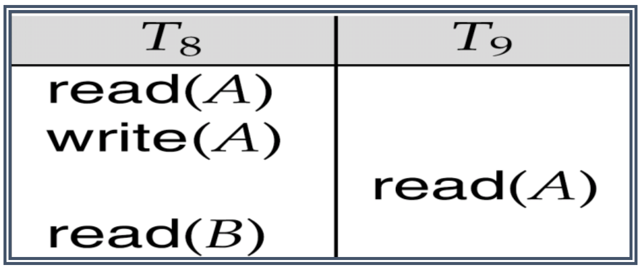


T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it. Determining such equivalence requires analysis of operations other than read and write.

Recoverability

Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

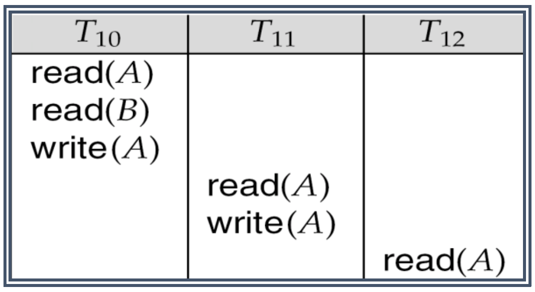


T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

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



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

If T_{10} fails, T_{11} and T_{12} must also be rolled back. Can lead to the undoing of a significant amount of work Cascade less schedules — cascading rollbacks cannot occur; 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.Every cascade less schedule is also recoverable It is desirable to restrict the schedules to those that are cascade less

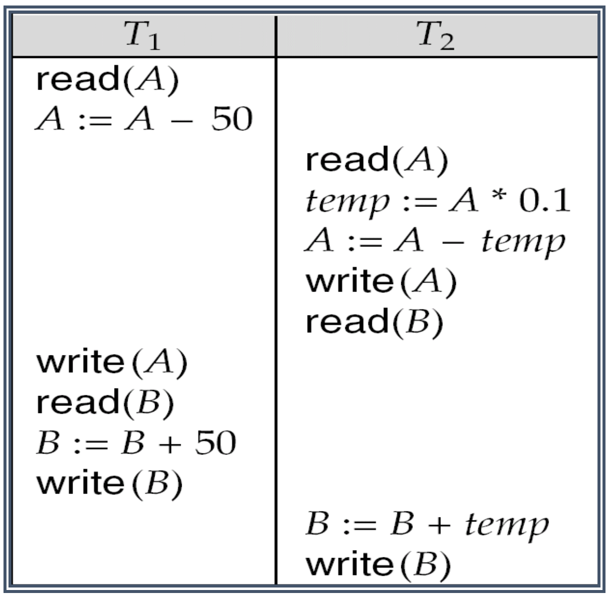
Concurrency Control

A database must provide a mechanism that will ensure that all possible schedules are

either conflict or view serializable, and are recoverable and preferably cascadeless A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency Are serial schedules recoverable/cascadeless? Testing a schedule for serializability *after* it has executed is a little too late! Goal – to develop concurrency control protocols that will assure serializability.

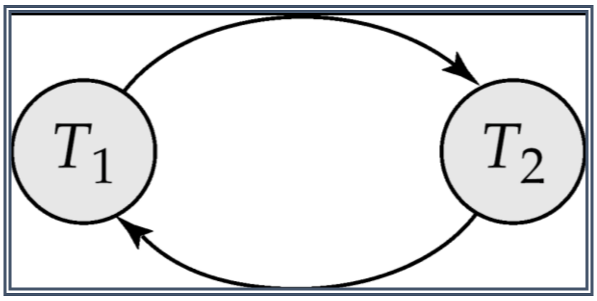
Implementation Of Isolation

Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.A policy in which only one transaction can execute at a time generates serial schedules,but provides a poor degree of concurrency.Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.



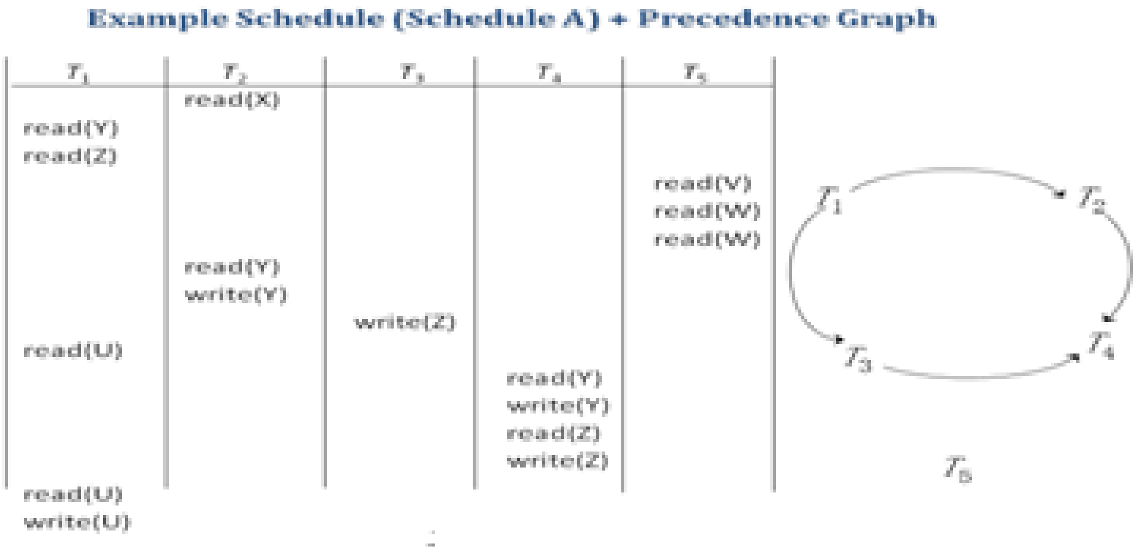
T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Testing For Serializability



- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- Precedence graph — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data

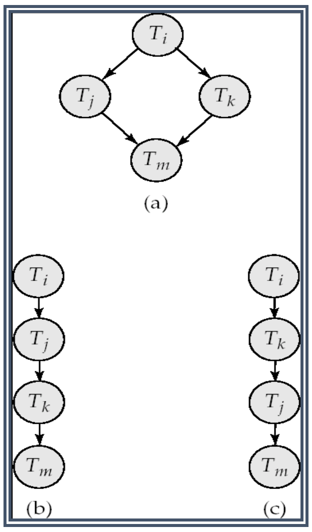
  item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.



Example Schedule (Schedule A) + Precedence Graph

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

Test for Conflict Serializability A schedule is conflict serializable if and only if its precedence graph is acyclic.Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)



If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph.

For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$ Are there others?

Test for View Serializability

The precedence graph test for conflict serializability cannot be used directly to test for view serializability.Extension to test for view serializability has cost exponential in the size of the precedence graph.The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems. Thus existence of an efficient algorithm is *extremely* unlikely.

However practical algorithms that just check some sufficient conditions for view serializability can still be used.

Concurrency Control

Concurrency Control vs. Serializability Tests

Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .Concurrency control protocols generally do not examine the precedence graph as it is being created Instead a protocol imposes a discipline that avoids nonseralizable schedules.Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable E.g. a read-only transaction that wants to get an approximate total balance of all Accounts. Example. database statistics computed for query optimization can be approximate (why?) Such transactions need not be serializable with respect to other transactions Tradeoff accuracy for performance Levels of Consistency in SQL-92 Serializable — default Repeatable read — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable

it may find some records inserted by a transaction but not find others.

Read committed — only committed records can be read, but successive reads of recor may

return different (but committed) values.

Read uncommitted — even uncommitted records may be read.Transaction Definition in

SQL Data manipulation language must include a construct for specifying the set of actions

that comprise a transaction.In SQL, a transaction begins implicitly.A transaction in SQL ends

by:Commit work commits current transaction and begins a new one.

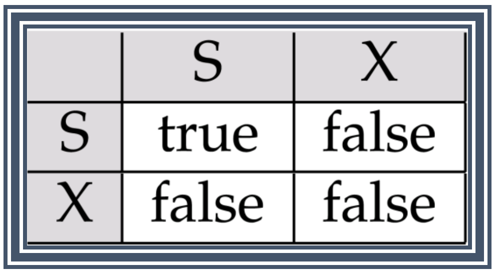
Rollback work causes current transaction to abort In almost all database systems, by default,

every SQL statement also commits implicitly if it executes successfully Implicit commit can

be turned off by a database directive E.g. in JDBC,connection. setAutoCommit(false);

Lock Based Protocols

A lock is a mechanism to control concurrent access to a data item



	S	X
S	true	false
X	false	false

Fig:Lock-compatibility matrix

Data items can be locked in two modes :

1. *exclusive (X) mode*. Data item can be both read as well as

   written. X-lock is requested using lock-X instruction.

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

   requested using lock-S instruction.

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

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

- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

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

Example :if a transaction performing locking:

T_2: lock-S*(A)*;

read *(A)*;

unlock*(A)*;

lock-S*(B)*;

read *(B)*;

unlock*(B)*;

display*(A+B)*

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

- A locking protocol is a set of rules followed by all transactions while requesting and

releasing locks. Locking protocols restrict the set of possible schedules.Pitfalls of Lock-Based Protocols Consider the partial schedule Neither T_3 nor T_4 can make progress — executing lock-S*(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing lock-X*(A)* causes T_3 to wait for T_4 to release its lock on *A*.Such a situation is called a deadlock. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

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

Two-Phase Locking Protocol

This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase

– transaction may obtain locks

– transaction may not release locks

Phase 2: Shrinking Phase

– transaction may release locks

– transaction may not obtain locks

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock). All locks are released after commit or abort

Implementation of Locking

A lock manager can be implemented as a separate process to which transactions send lock and unlock requests The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock).The requesting transaction waits until its request is answered The lock manager maintains a data-structure called a lock table to record granted locks and pending requests The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.

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

- Cascading roll-back is possible under two-phase locking. To avoid this, follow a

modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.

- Rigorous two-phase locking is even stricter: here *all* locks are held till commit/abort.

In this protocol transactions can be serialized in the order in which they commit.

Timestamp Based Protocols

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

The protocol manages concurrent execution such that the time-stamps determine the serializability order.In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:

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

write(*Q*) successfully.

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

read(*Q*) successfully.

The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

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

If TS(T_i) ≤ W-timestamp(*Q*), then T_i needs to read a value of *Q* that was

already overwritten.Hence, the read operation is rejected, and T_i is rolled back.

If TS(T_i)≥ W-timestamp(*Q*), then the read operation is executed, and R-

timestamp(*Q*) is set to max(R-timestamp(*Q*), TS(T_i)).

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

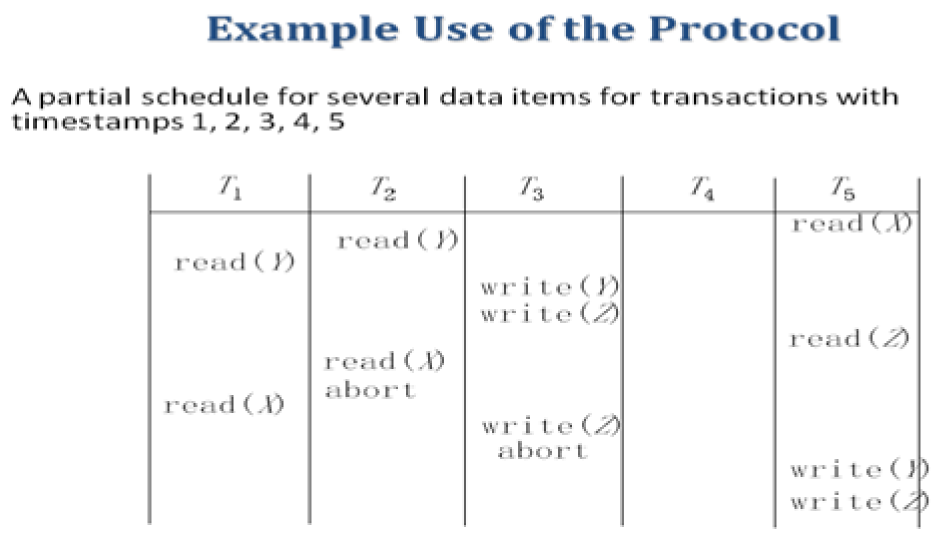
If TS(T_i) < R-timestamp(*Q*), then the value of *Q* that T_i is producing was needed

previously, and the system assumed that that value would never be produced.

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

If TS(T_i) < W-timestamp(*Q*), then T_i is attempting to write an obsolete value of *Q*.

Hence, this write operation is rejected, and T_i is rolled back.Otherwise, the write operation is executed, and W-timestamp(*Q*) is set to TS(T_i).



Example Use of the Protocol

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

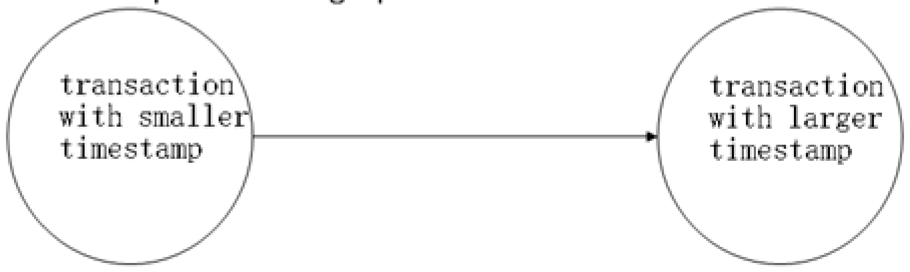
T_1	T_2	T_3	T_4	T_5
				read(X)
	read(Y)			
read(Y)				
		write(Y)		
		write(Z)		
				read(Z)
	read(X)			
	abort			
read(X)				
		write(Z)		
		abort		
				write(Y)
				write(Z)

A partial schedule for several data items for transactions with

timestamps 1, 2, 3, 4, 5

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:


transaction with smaller timestamp
transaction with larger timestamp

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph Timestamp protocol ensures freedom from deadlock as no transaction ever waits. But the schedule may not be cascade-free, and may not even be recoverable.

Thomas' Write Rule Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances. When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}. Rather than rolling back T_i as the timestamp ordering protocol would have done, this {write} operation can be ignored.Otherwise this protocol is the same as the timestamp ordering protocol.

- ✓ Thomas' Write Rule allows greater potential concurrency.
- ✓ Allows some view-serializable schedules that are not conflict-serializable.

Validation Based Protocol

Execution of transaction T_i is done in three phases.

1. Read and execution phase: Transaction T_i writes only to temporary local variables
2. Validation phase: Transaction T_i performs a ``validation test" to determine if local variables can

   be written without violating serializability.
3. Write phase: If T_i is validated, the updates are applied to the database; otherwise, T_i is

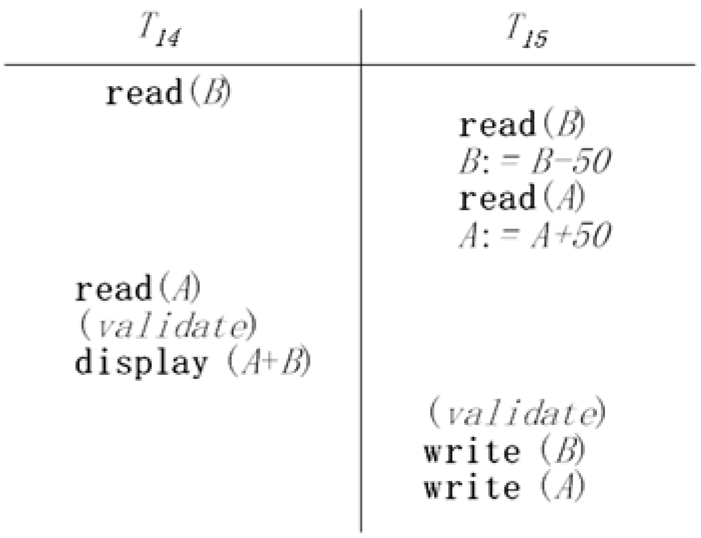
   rolled back.

The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.Assume for simplicity that the validation and write phase occur together,atomically and serially i.e., only one transaction executes validation/write at a time.Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation.

Each transaction T_i has 3 timestamps

- Start(T_i) : the time when T_i started its execution
- Validation(T_i): the time when T_i entered its validation phase
- Finish(T_i) : the time when T_i finished its write phase Serializability order is determined by timestamp given at validation time, to increase concurrency.

- Example of schedule produced using validation



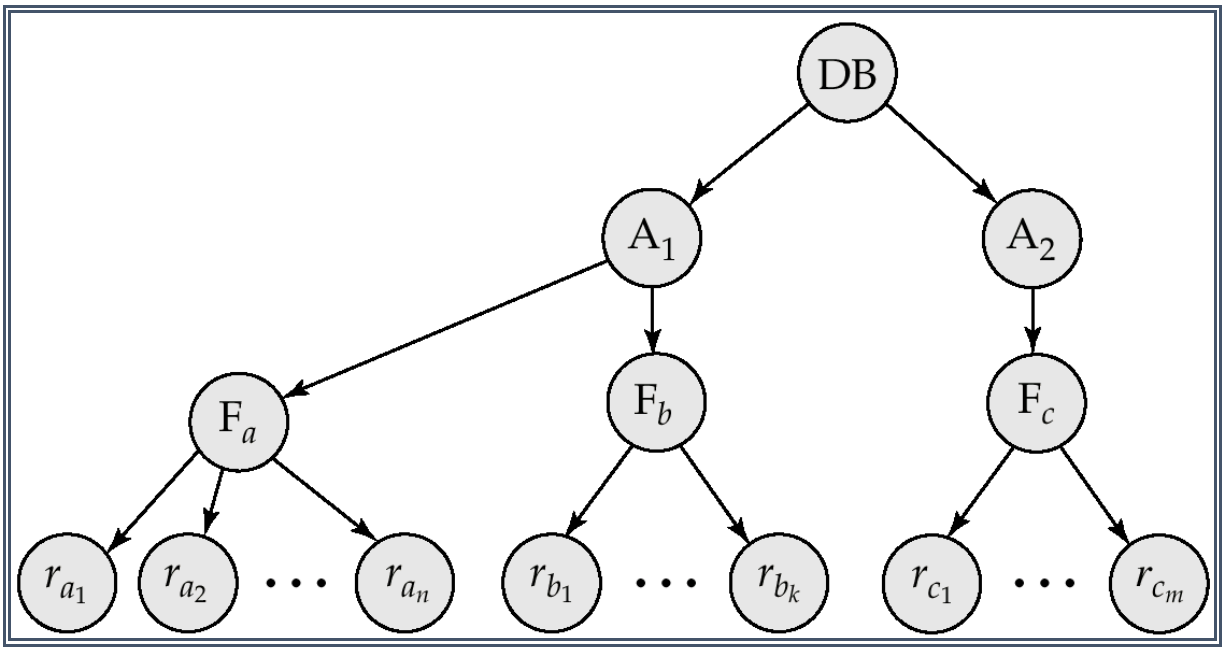
T_{14}	T_{15}
read(B)	
	read(B)
	$B := B-50$
	read(A)
	$A := A+50$
read(A)	
($validate$)	
display ($A+B$)	
	($validate$)
	write (B)
	write (A)

Multiple Granularities

Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones Can be represented graphically as a tree (but don't confuse with tree-locking protocol) When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

Granularity of locking (level in tree where locking is done):ine granularity (lower in tree): high concurrency, high locking overhead coarse granularity (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



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

– *database*

– *area*

– *file*

– *record*

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

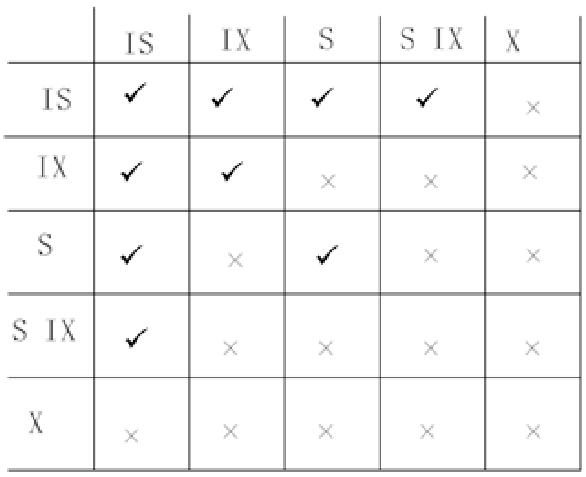
intention-shared (IS): indicates explicit locking at a lower level of the tree but

only with shared locks.

intention-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks

shared and intention-exclusive (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

- The compatibility matrix for all lock modes is:



	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

Multiversion Schemes

Multiversion concurrency control techniques keep the old values of a data item when the item is updated. Several versions (values) of an item are maintained. When a transaction requires access to an item, an appropriate version is chosen to maintain the serialisability of the concurrently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted, by reading an older version of the item to maintain serialisability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway â€“ for example, for recovery purpose. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a temporal database, which keeps track of all changes and the items at which they occurred. In such cases, there is no additional penalty for multiversion techniques, since older versions are already maintained.

Multiversion techniques based on timestamp ordering

In this technique, several versions X1, X2, â€¦ Xk of each data item X are kept by the system. For each version, the value of version Xi and the following two timestamps are kept:

1. read_TS(Xi): The read timestamp of Xi; this is the largest of all the timestamps of transactions that have successfully read version Xi.
2. write_TS(Xi): The write timestamp of Xi; this is the timestamp of the transaction that wrote the value of version Xi.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version of item X, Xk+1, is created, with both the write_TS(Xk+1) and the read_TS(Xk+1) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version Xi, the value of read_TS(Xi) is set to the largest of read_TS(Xi) and TS(T).

To ensure serialisability, we use the following two rules to control the reading and writing of data items:

1. If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(Xi) of all versions of X which is also less than or equal to TS(T), and TS(T) < read_TS(Xi), then abort and roll back transaction T; otherwise, create a new version Xj of X with read_TS(Xj) = write_TS(Xj) = TS(T).

2. If transaction T issues a read_item(X) operation, and version i of X has the highest write_TS(Xi) of all versions of X which is also less than or equal to TS(T), then return the value of Xi to transaction T, and set the value of read_TS(Xj) to the largest of TS(T) and the current read_TS(Xj).

Multiversion two-phase locking

In this scheme, there are three locking modes for an item: read, write and certify. Hence, the state of an item X can be one of 'read locked', 'write locked', 'certify locked' and 'unlocked'. The idea behind the multiversion two-phase locking is to allow other transactions Tâ€™ to read an item X while a single transaction T holds a write lock X. (Compare with standard locking scheme.) This is accomplished by allowing two versions for each item X; one version must always have been written by some committed transaction. The second version Xâ€™ is created when a transaction T acquires a write lock on the item. Other transactions can continue to read the committed version X while T holds the write lock. Now transaction T can change the value of Xâ€™ as needed, without affecting the value of the committed version X. However, once T is ready to commit, it must obtain a certify lock on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write lock items are released by any reading transactions. At this point, the committed version X of the data item is set to the value of version Xâ€™, version Xâ€™ is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown below:

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 13.35

In this multiversion two-phase locking scheme, reads can proceed concurrently with a write operation â€“ an arrangement not permitted under the standard two-phase locking schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on all items it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version X that was written by committed transaction. However, deadlock may occur.

Granularity of data items

All concurrency control techniques assumed that the database was formed of a number of items. A database item could be chosen to be one of the following:

- A database record.
- A field value of a database record.
- A disk block.
- A whole file.
- The whole database.

Several trade-offs must be considered in choosing the data item size. We shall discuss data item size in the context of locking, although similar arguments can be made for other concurrency control techniques.

First, the larger the data item size is, the lower the degree of concurrency permitted. For example, if the data item is a disk block, a transaction T that needs to lock a record A must lock the whole disk block X that contains A. This is because a lock is associated with the whole data item X. Now, if another transaction S wants to lock a different record B that happens to reside in the same block X in a conflicting disk mode, it is forced to wait until the first transaction releases the lock on block X. If the data item size was a single record, transaction S could proceed as it would be locking a different data item (record B).

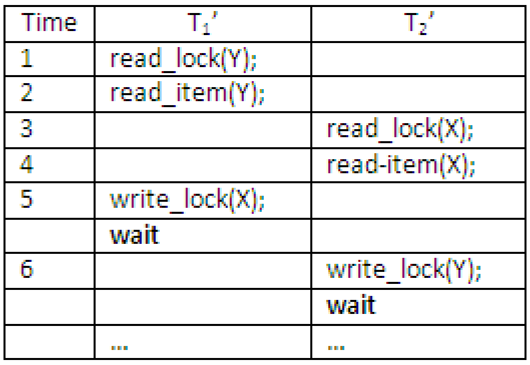
On the other hand, the smaller the data item size is, the more items will exist in the database. Because every item is associated with a lock, the system will have a larger number of locks to be handled by the lock manger. More lock and unlock operations will be performed, causing a higher overhead. In addition, more storage space will be required for the lock table. For timestamps, storage is required for the read_TS and write_TS for each data item, and the overhead of handling a large number of items is similar to that in the case of locking.

The size of data items is often called the data item granularity. Fine granularity refers to small item size, whereas coarse granularity refers to large item size. Given the above trade-offs, the obvious question to ask is: What is the best item size? The answer is that it depends on the types of transactions involved. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record. On the other hand, if a transaction typically accesses many records of the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items.

Most concurrency control techniques have a uniform data item size. However, some techniques have been proposed that permit variable item sizes. In these techniques, the data item size may be changed to the granularity that best suits the transactions that are currently executing on the system.

deadlock

Another problem that may be introduced by 2PL protocol is deadlock. The formal definition of deadlock will be discussed below. Here, an example is used to give you an intuitive idea about the deadlock situation. The two transactions that follow the 2PL protocol can be interleaved as shown here:



Time	T_1'	T_2'
1	read_lock(Y);	
2	read_item(Y);	
3		read_lock(X);
4		read-item(X);
5	write_lock(X);	
	wait	
6		write_lock(Y);
		wait
	...	...

Figure 13.24

At time step 5, it is not possible for T1â€™ to acquire an exclusive lock on X as there is already a shared lock on X held by T2â€™. Therefore, T1â€™ has to wait. Transaction T2â€™ at time step 6 tries to get an exclusive lock on Y, but it is unable to as T1â€™ has a shared lock on Y already. T2â€™ is put in waiting too. Therefore, both transactions wait fruitlessly for the other to release a lock. This situation is known as a deadly embrace or deadlock. The above schedule would terminate in a deadlock.

Conservative 2PL

A variation of the basic 2PL is conservative 2PL also known as static 2PL, which is a way of avoiding deadlock. The conservative 2PL requires a transaction to lock all the data items it needs in advance. If at least one of the required data items cannot be obtained then none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Although conservative 2PL is a deadlock-free protocol, this solution further limits concurrency.

Strict 2PL

In practice, the most popular variation of 2PL is strict 2PL, which guarantees a strict schedule. (Strict schedules are those in which transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted). In strict 2PL, a transaction T does not release any of its locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Notice the difference between conservative and strict 2PL; the former must lock all items before it starts, whereas the latter does not unlock any of its items until after it terminates (by committing or aborting). Strict 2PL is not deadlock-free unless it is combined with conservative 2PL.

In summary, all type 2PL protocols guarantee serialisability (correctness) of a schedule but limit concurrency. The use of locks can also cause two additional problems: deadlock and livelock. Conservative 2PL is deadlock-free.

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions {T_0, T_1, T_2, ...,T_n}. T_0 needs a resource X to complete its task. Resource X is held by T_1, and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock, is older than T_j − then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is T_i is younger than T_j − then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

Deadlock Avoidance

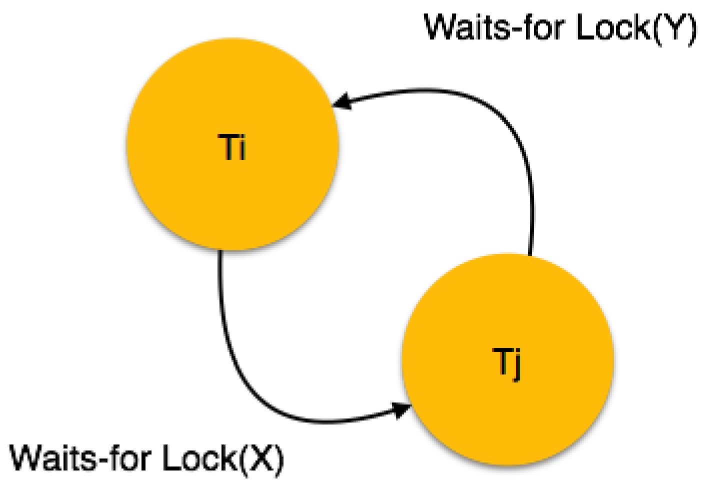
Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction T_i requests for a lock on an item, say

X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j. If T_j releases item X, the edge between them is dropped and T_i locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.



Here, we can use any of the two following approaches –

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.
- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**.

Recovery System:

Crash Recovery

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

Failure Classification

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

Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further. This is called transaction failure where only a few transactions or processes are hurt.

Reasons for a transaction failure could be –

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

System Crash

There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

Storage Structure

We have already described the storage system. In brief, the storage structure can be divided into two categories −

- Volatile storage − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- Non-volatile storage − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.

- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-based Recovery

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

Log-based recovery works as follows −

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.

<T_n, Start>

- When the transaction modifies an item X, it write logs as follows −

<T_n, X, V_1, V_2>

It reads T_n has changed the value of X, from V_1 to V_2.

- When the transaction finishes, it logs −

<T_n, commit>

The database can be modified using two approaches −

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

Recovery with Concurrent Transactions

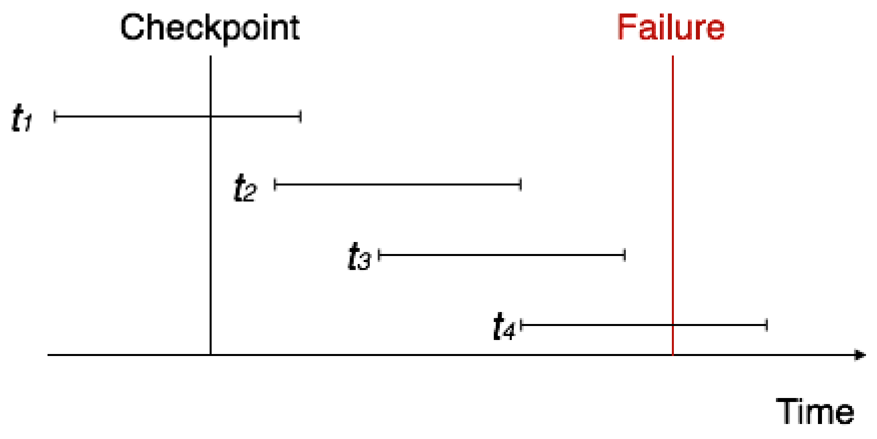
When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

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

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner –



- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

Module-V

Data Storage and Query Processing

Overview of Physical Storage Media

Storage media are classified by speed of access, cost per unit of data to buy the media, and by the medium's reliability. Unfortunately, as speed and cost go up, the reliability does down.

1. Cache is the fastest and the most costly for of storage. The type of cache referred to here is the type that is typically built into the CPU chip and is 256KB, 512KB, or 1MB. Thus, cache is used by the operating system and has no application to database, per se.

2. Main memory is the volatile memory in the computer system that is used to hold programs and data. While prices have been dropping at a staggering rate, the increases in the demand for memory have been increasing faster. Today's 32-bit computers have a limitation of 4GB of memory. This may not be sufficient to hold the entire database and all the associated programs, but the more memory available will increase the response time of the DBMS. There are attempts underway to create a system with the most memory that is cost effective, and to reduce the functionality of the operating system so that only the DBMS is supported, so that system response can be increased. However, the contents of main memory are lost if a power failure or system crash occurs.

3. Flash memory is also referred to as *electrically erasable programmable read-only memory (EEPROM)*. Since it is small (5 to 10MB) and expensive, it has little or no application to the DBMS.

4. Magnetic-disk storage is the primary medium for long-term on-line storage today. Prices have been dropping significantly with a corresponding increase in capacity. New disks today are in excess of 20GB. Unfortunately, the demands have been increasing and the volume of data has been increasing faster. The organizations using a DBMS are always trying to keep up with the demand for storage. This media is the most cost-effective for on-line storage for large databases.

5. Optical storage is very popular, especially CD-ROM systems. This is limited to data that is read-only. It can be reproduced at a very low-cost and it is expected to grow in popularity, especially for replacing written manuals.

6. Tape storage is used for backup and archival data. It is cheaper and slower than all of the other forms, but it does have the feature that there is no limit on the amount of data that can be stored, since more tapes can be purchased. As the tapes get increased capacity, however, restoration of data takes longer and longer, especially when only a small amount of data is to be restored. This is because the retrieval is sequential, the slowest possible method.

Magnetic Disks

A typical large commercial database may require hundreds of disks!

Physical Characteristics of Disks

Disks are actually relatively simple. There is normally a collection of platters on a spindle. Each platter is coated with a magnetic material on both sides and the data is stored on the surfaces. There is a read-write head for each surface that is on an arm assembly that moves back and forth. A motor spins the platters at a high constant speed, (60, 90, or 120 revolutions per seconds.)

The surface is divided into a set of tracks (circles). These tracks are divided into a set of sectors, which is the smallest unit of data that can be written or read at one time. Sectors can range in size from 31 bytes to 4096 bytes, with 512 bytes being the most common. A collection of a specific track from both surfaces and from all of the platters is called a cylinder.

Platters can range in size from 1.8 inches to 14 inches. Today, 5 1/4 inches and 3 1/2 inches are the most common, because they have the highest seek times and lowest cost.

A disk controller interfaces the computer system and the actual hardware of the disk drive. The controller accepts high-level command to read or write sectors. The controller then converts the commands in the necessary specific low-level commands. The controller will also attempt to protect the integrity of the data by computing and using checksums for each sector. When attempting to read the data back, the controller recalculates the checksum and makes several attempts to correctly read the data and get matching checksums. If the controller is unsuccessful, it will notify the operating system of the failure.

The controller can also handle the problem of eliminating bad sectors. Should a sector go bad, the controller logically remaps the sector to one of the extra unused sectors that disk vendors provide, so that the reliability of the disk system is higher. It is cheaper to produce disks with a greater amount of sectors than advertised and then map out bad sectors than it is to produce disks with no bad sectors or with extremely limited possibility of sectors going bad.

There are many different types of disk controllers, but the most common ones today are SCSI, IDE, and EIDE.

One other characteristic of disks that provides an interesting performance is the distance from the read-write head to the surface of the platter. The smaller this gap is means that data can be written in a smaller area on the disk, so that the tracks can be closer together and the disk has a greater capacity. Often the distance is measured in microns. However, this means that the possibility of the head touching the surface is increased. When the head touches the surface while the surface is spinning at a high speed, the result is called a "head crash", which scratches the surface and defaces the head. The bottom line to this is that someone must replace the disk.

Performance Measures of Disks

1. *Seek time* is the time to reposition the head and increases with the distance that the head must move. Seek times can range from 2 to 30 milliseconds. *Average seek time* is the average of all seek times and is normally one-third of the worst-case seek time.

2. *Rotational latency time* is the time from when the head is over the correct track until the data rotates around and is under the head and can be read. When the rotation is 120 rotations per second, the rotation time is 8.35 milliseconds. Normally, the *average rotational latency time* is one-half of the rotation time.

3. *Access time* is the time from when a read or write request is issued to when the data transfer begins. It is the sum of the seek time and latency time.

4. *Data-transfer rate* is the rate at which data can be retrieved from the disk and sent to the controller. This will be measured as megabytes per second.

5. *Mean time to failure* is the number of hours (on average) until a disk fails. Typical times today range from 30,000 to 800,000 hours (or 3.4 to 91 years).

Optimization of Disk-Block Access

Requests for disk I/O are generated by both the file system and by the virtual memory manager found in most systems. Each request specifies the address on the disk to be referenced; that address specifies is in the form of a block number. Each block is a contiguous sequence of sectors from a single track of one platter and ranges from 512 bytes to several kilobytes of data. The lower level file manager must convert block addresses into the hardware-level cylinder, surface, and sector number.

Since access to data on disk is several orders of magnitude slower is access to data in main memory; much attention has been paid to improving the speed of access to blocks on the disk. This is also where more main memory can speed up the response time, by making sure that the data needed is in memory when it is needed.

This is the same problem that is addressed in designing operating systems, to insure the best response time from the file system manager and the virtual memory manager.

- Scheduling. Disk-arm scheduling algorithms attempt to order accesses in an attempt to increase the number of accesses that can be processed in a given amount of time. The might include First-Come/First-Serve, Shortest Seek First, and elevator.

- File organization. To reduce block-access time, data could be arranged on the disk in the same order that it is expected to be retrieved. (This would be storing the data on the disk in order based on the primary key.) At best, this starts to produce less and less of a benefit, as there are more inserts and deletes. Also we have little control of where on the disk things get stored. The more the data gets fragmented on the disk, the more time it takes to locate it.

- Nonvolatile write buffer. Using non-volatile memory (flash memory) can be used to protect the data in memory from crashes, but it does increase the cost. It is possible that the use of an UPS would be more effective and cheaper.

- Log disk. You can use a disk for writing a sequential log.

- Buffering. The more information you have in buffers in main memory, the more likely you are to not have to get the information from the disk. However it is more likely that more of the memory will be wasted with information not necessary.

RAID

RAIDs are Redundant Arrays of Inexpensive Disks. There are six levels of organizing these disks:

- 0 -- Non-redundant Striping

- 1 -- Mirrored Disks

- 2 -- Memory Style Error Correcting Codes

- 3 -- Bit Interleaved Parity

- 4 -- Block Interleaved Parity
- 5 -- Block Interleaved Distributed Parity
- 6 -- P + Q Redundancy

Tertiary Storage

This is commonly optical disks and magnetic tapes.

Storage Access

A database is mapped into a number of different files, which are maintained by the underlying operating system. Files are organized into block and a block may contain one or more data item.

A major goal of the DBMS is to minimize the number of block transfers between the disk and memory. Since it is not possible to keep all blocks in main memory, we need to manage the allocation of the space available for the storage of blocks. This is also similar to the problems encountered by the operating system, and can be in conflict with the operating system, since the OS is concerned with processes and the DBMS is concerned with only one family of processes.

Buffer Manager

Programs in a DBMS make requests (that is, calls) on the buffer manager when they need a block from a disk. If the block is already in the buffer, the requester is passed the address of the block in main memory. If the block in not in the buffer, the buffer manager first allocates space in the buffer for the block, through out some other block, if required, to make space for the new block. If the block that is to be thrown out has been modified, it must first be written back to the disk. The internal actions of the buffer manager are transparent to the programs that issue disk-block requests.

- *Replacement strategy*. When there is no room left in the buffer, a block must be removed from the buffer before a new one can be read in. Typically, operating systems use a least recently use (LRU) scheme. There is also a Most Recent Used (MRU) that can be more optimal for DBMSs.
- *Pinned blocks*. A block that is not allowed to be written back to disk is said to be pinned. This could be used to store data that has not been committed yet.
- *Forced output of blocks*. There are situations in which it is necessary to write back to the block to the disk, even though the buffer space is not currently needed. This might be done during system lulls, so that when activity picks up, a write of a modified block can be avoided in peak periods.

File Organization

Fixed-Length Records

Suppose we have a table that has the following organization:

```
type deposit = record
   branch-name : char(22);
```

```
    account-number : char(10);
    balance : real;
end
```

- If each character occupies 1 byte and a real occupies 8 bytes, then this record occupies 40 bytes. If the first record occupies the first 40 bytes and the second record occupies the second 40 bytes, etc. we have some problems.
- It is difficult to delete a record, because there is no way to indicate that the record is deleted. (At least one system automatically adds one byte to each record as a flag to show if the record is deleted.) Unless the block size happens to be a multiple of 40 (which is extremely unlikely), some records will cross block boundaries. It would require two block access to read or write such a record.

One solution might be to compress the file after each deletion. This will incur a major amount of overhead processing, especially on larger files. Additionally, there is the same problem on inserts!

Another solution would be to have two sets of pointers. One that would link the current record to the next logical record (linked list) plus a free list (a list of free slots.) This increases the size the file.

Variable-Length Records

We can use variable length records:

- Storage of multiple record types in one file.
- Record types that allow variable lengths for one or more fields
- Record types that allow repeating fields.

A simple method for implementing variable-length records is to attach a special *end-of-record* symbol at the end of each record. But this has problems:

- To easy to reuse space occupied formerly by a deleted record.
- There is no space in general for records to grow. If a variable-length record is updated and needs more space, it must be moved. This can be very costly.

It could be solved:

- By making a variable-length into a fixed length.
- By using pointers to point to fixed length records, chained together by pointers.

As you can see, there is not an easy answer.

Organization of Records in Files

Heap File Organization

Any record can be placed anywhere in the file. There is no ordering of records and there is a single file for each relation.

Sequential File Organization

Records are stored in sequential order based on the primary key.

Hashing File Organization

Any record can be placed anywhere in the file. A hashing function is computed on some attribute of each record. The function specifies in which block the record should be placed.

Clustering File Organization

Several different relations can be stored in the same file. Related records of the different relations can be stored in the same block.

Data Dictionary Storage

A RDBMS needs to maintain data about the relations, such as the schema. This is stored in a data dictionary (sometimes called a system catalog):

- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of views, defined on the database, and definitions of those views
- Integrity constraints
- Names of authorized users
- Accounting information about users
- Number of tuples in each relation
- Method of storage for each relation (clustered/non-clustered)
- Name of the index
- Name of the relation being indexed
- Attributes on which the index in defined
- Type of index formed

Indexing and Hashing: Basic Concepts

Basic Concepts

An index for a file is like a catalog for a book in the library. Cards in the catalog are stored in order with portions of the catalog order by author's name, book title, or subject. Items in the database are catalogued with indices based on keys. When a table is defined, it has a primary key; however, it can have additional keys defined.

Typical databases are too large to search sequentially looking for specific records and more sophisticated indexing techniques are employed. The two basic kinds of indices are:

Ordered indices

- Hash indices

Different techniques are evaluated on the basis of several opposing factors:

- Access types: The access could be for a specific value or a specified range.
- Access time: The amount of time it takes to find a particular data item or set of items.
- Insertion time: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data items, as well as update the index structure. Obviously, the more indices that must be updated, the longer this will take. (Note: When optimizing performance of a DBMS, it is useful to review all indices to make sure that the primary key is the key used most often and then any other indices are being used often enough to justify the overhead of maintaining the index.)
- Deletion time: The time it takes to delete a data item. This includes the time to locate the item and then delete it, as well as updating the index structure. (Note: When optimizing performance of a DBMS, it is useful to review how deletions can cascade and cause deletions in other tables. This is part of the overhead of maintaining the index structure.)
- Space overhead: The addition space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance. (With the rapidly dropping prices and increased capacity, the developer must still be limited by the constraint that there is only so much space available and that it still requires more time to maintain the structure. This overhead can still be significant!)

With indexing, there is now a new meaning for the over-used word "key":

- Primary key
- Candidate key
- Superkey
- Search key

Records in an indexed file are stored in some type of order (including unordered.). If the file containing the records is order sequentially, the index whose search key specifies the sequential order of the file is the primary key. Primary indices are also called *clustering indices.* Indices whose search key specifies an order different from the sequential order are called *secondary indices*, or *nonclustering indices*.

Primary Index

If we assume that all files are ordered sequentially on some search key (primary index), then such files are called *index-sequential files.* (The Index-Sequential Access Method or *ISAM* is one of the oldest index schemes used in database systems.) They are used for applications that require both sequential processing of the entire file and random access to the individual records. An example for a back account would be:

RECORD NR (not in table)	**LOCATION (primary key)**	**ACCOUNT NR**	**BALANCE**	**NEXT RECORD**
1	Brighton	A-217	750	2
2	Downtown	A-101	500	3
3	Downtown	A-110	600	4
4	Mianus	A-215	700	5
5	Perryridge	A-102	900	6
6	Perryridge	A-201	700	7
7	Perryridge	A-218	700	8
8	Redwood	A-222	700	9
9	Round Hill	A-305	350	**NULL**

It could be even more complex if the primary key was the combination of the location and the account number. The advantage of this method is that when processing is being done sequentially, this is the fastest access method. However it is expensive in terms of maintaining. If a record is to be added to the table, the appropriate slot must be located, a new slot must be added to the table, and all records must be moved down one slot. Then the new slot is inserted. Then all of the index columns must be updated. The file stored on the disk can be stored as contiguous or non-contiguous data. In terms of the DBMS performance, being able to maintain the file contiguously offers better DBMS performance, however the is probably in conflict with the operating system's allocation method. (Some DBMS systems will allocate the maximum space that a file can have when the file is created to reduce the overhead caused when records are added to the file.) Some systems require space that the operating system can not manage, so that some of these problems are reduced.

Deletions are preformed in the reverse sequence. The appropriate slot must be located and all records below that slot must be moved up. Again, all of the index columns must be update.

Updates on these kind of files are normally batched together to minimize the system overhead, because the cost of updating one record is almost exactly the same as updating one hundred records. This is not particularly suited to interactive updates.

Dense and Sparse Indices

This can be improved by adding another data structure (especially for random access).

- Dense Index. An *index record* (or *index entry*) appears for every search-key value, containing the value and the location for the first data record with that value. The dense index for the previous table would look like:

Value	**Start**
Brighton	1
Downtown	2
Mianus	4
Perryridge	5
Redwood	8
Round Hill	9

- Sparse Index. An index record is created only for some of the values, which is the only difference between the two versions in terms of the data and its structure. With the sparse index, the system has to locate the largest value in the index that does not exceed the search key. From that point the records must be checked sequentially until a match is found.

Obviously, it is generally faster to locate a record if we have a dense index rather than a spare index, but the maintenance overhead is greater for a dense index in addition to requiring more disk space. This is the tradeoff that the system designer must make. The major advantage to the sparse index is that it will required bring in less blocks, which can be expensive in terms of time, because searching the block once is memory is SO much faster! The obvious answer might not be the best answer.

Multilevel Indices

The above system appears to work well, but it does not scale up well. If the index is small enough to keep in memory this is a really good method, but when the index becomes too large, the problems of searching the index become just like searching the file: It takes too long!

Suppose we have a file with 100,000 records with 10 records stored per block. If we have one index record for each block, the index has 10,000 entries. Since the index entries are smaller, there might be 100 entries per block in 100 blocks that are stored sequentially on the disk. Doing a sequential search would involve up to 100 reads (average would be n/2, or 50 reads).

Binary searches could be used. The cost for *b* blocks would be $\log_2(b)$ blocks to be read. In this example that would be seven reads. If each read is 30 milliseconds access time, we have 210 milliseconds, which by today's standards is long! (That example ignores a number of other time-consuming activities.) A binary search is not possible if the system was built using overflow blocks.

What if we use a sparse index on the index file, which is small enough to keep in memory! That new index is an *outer index* and the original index is an *inner index*. The system performs a binary search on the outer index and now only has to make one read for the correct block of the inner index. There can be any number of inner levels, creating a multilevel index system.

Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing.

Index Update

Algorithms for updating single-level indices:

- Deletion. To delete a record, look up the record to delete. If the deleted record was the only record for that particular search-key, then the search-key value is also deleted from a dense index. If the deleted record was the first record for a particular search-key value, then the index was be updated to point the next record. Likewise if a sparse index was used, it too may have to be similarly updated.
- Insertion. First we perform a lookup using the search-key value of the record to be inserted. The index is dense and the search-key does not appear in the index, the search-key value must be inserted into the index. If the index is sparse, it may not have to be modified.

Insertion and deletion algorithms for multilevel indices are a simple extension of this scheme.

Secondary Indices

A secondary index on a candidate key looks just like a dense primary index, except that the record pointers are not stored in the record itself and must be stored somewhere else. Additionally, the secondary index might not be based on a candidate key. Therefore, it is not sufficient to point to the first occurrence. There must be pointers to every record. The "dense" index points to a structure that has an entry for each occurrence of that value. Obviously, the maintenance of secondary indices is more costly.

B^+-Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. A secondary disadvantage is to overhead of maintaining the file in a sequential order when inserting large amounts of data.

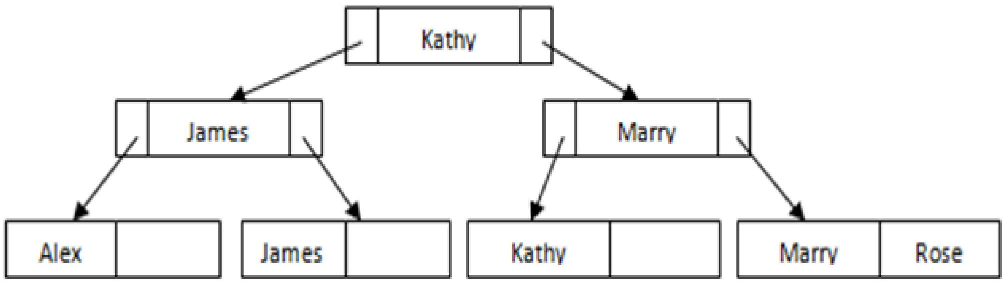
The B^+-tree index structure is widely used to maintain efficiency despite insertions and deletions of data. It is a multilevel index, where each node contains some pointers and search-key values.

B+ tree is used to store the records in the secondary memory. If the records are stored using this concept, then those files are called as B+ tree index files. Since this tree is balanced and sorted, all the nodes will be at same distance and only leaf node has the actual value, makes searching for any record easy and quick in B+ tree index files.

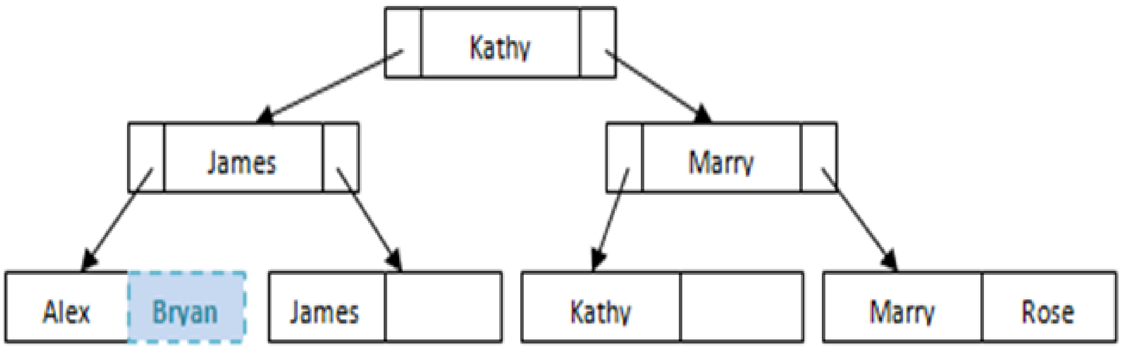
Even insertion/deletion in B+ tree does not take much time. Hence B+ tree forms an efficient method to store the records.

Searching, inserting and deleting a record is done in the same way we have seen above. Since it is a balance tree, it searches for the position of the records in the file, and then it fetches/inserts /deletes the records. In case it finds that tree will be unbalanced because of insert/delete/update, it does the proper re-arrangement of nodes so that definition of B+ tree is not changed.

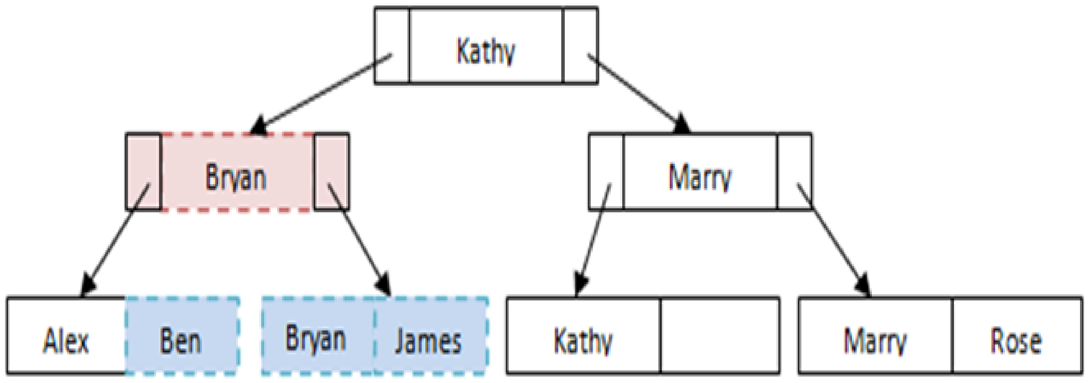
Below is the simple example of how student details are stored in B+ tree index files.



Suppose we have a new student Bryan. Where will he fit in the file? He will fit in the 1st leaf node. Since this leaf node is not full, we can easily add him in the node.



But what happens if we want to insert another student Ben to this file? Some re-arrangement to the nodes is needed to maintain the balance of the file.



Benefits of B+ Tree index files

As the file grows in the database, the performance remains the same. It does not degrade like in ISAM. This is because all the records are maintained at leaf node and all the nodes are at equi-distance from root. In addition, if there is any overflow, it automatically re-organizes the structure.

Even though insertion and deletion are little complicated, it can be done in fraction of seconds.

Leaf node allows only partial/ half filled, since records are larger than pointers.

B Tree index Files

B tree index file is similar to B+ tree index files, but it uses binary search concepts. In this method, each root will branch to only two nodes and each intermediary node will also have the data. And leaf node will have lowest level of data. However, in this method also, records will be sorted. Since all intermediary nodes also have records, it reduces the traversing till leaf node for the data. A simple B tree can be represented as below:

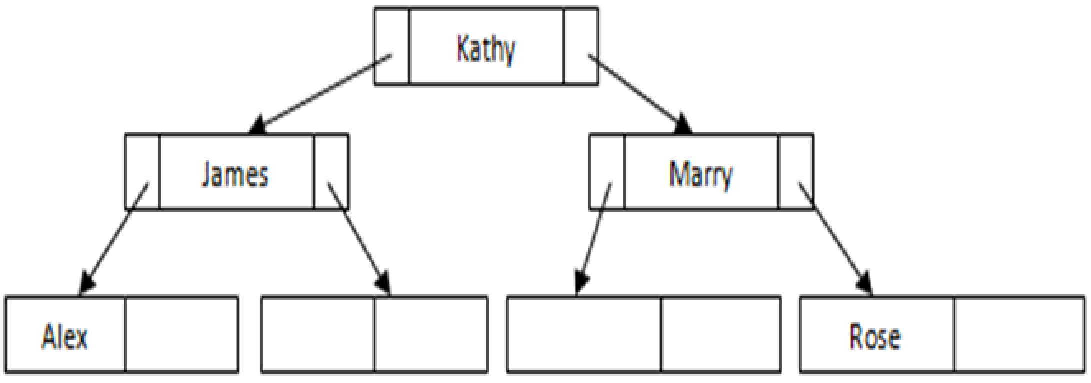


Fig: B tree

See the difference between this tree structure and B+ tree for the same example above. Here there is no repetition or pointers till leaf node. All the records are stored in all the nodes.

If we need to insert any record, it will be done as B+ tree index files, but it will make sure that each node will branch only to two nodes. If there is not enough space in any of the node, it will split the node and store the records.

Example of Simple Insert

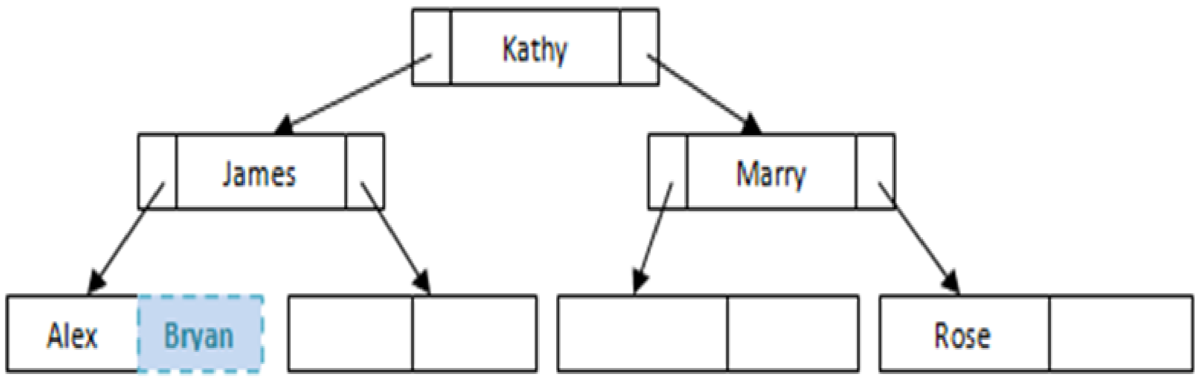


Fig: Example of Simple Insert

Example of splitting the nodes while inserting

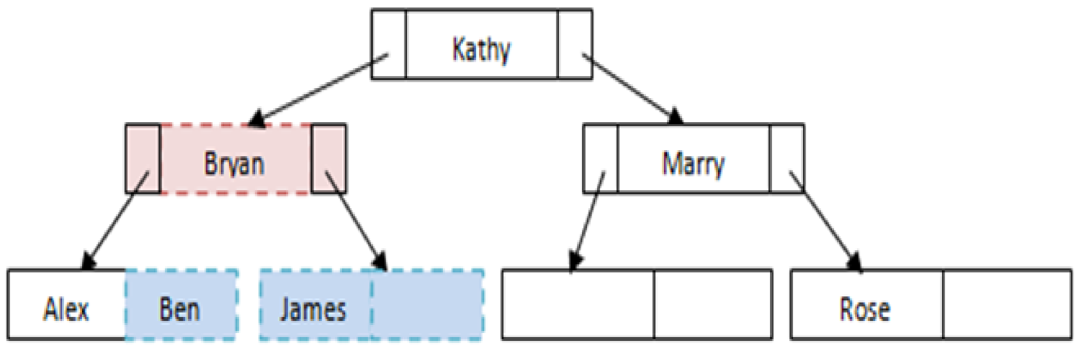
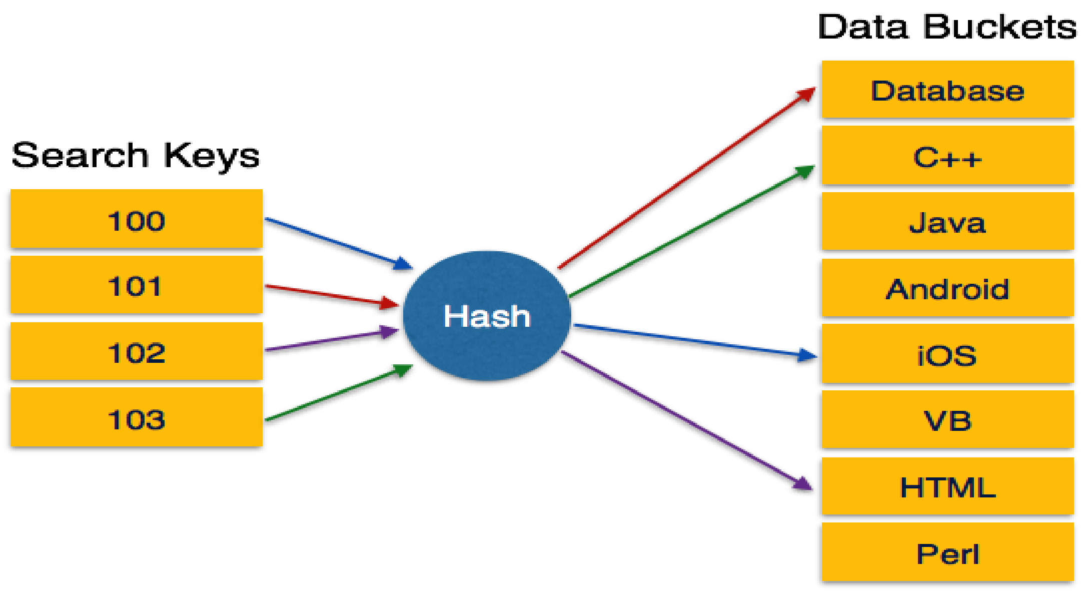


Fig: Example of splitting the nodes while inserting

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.

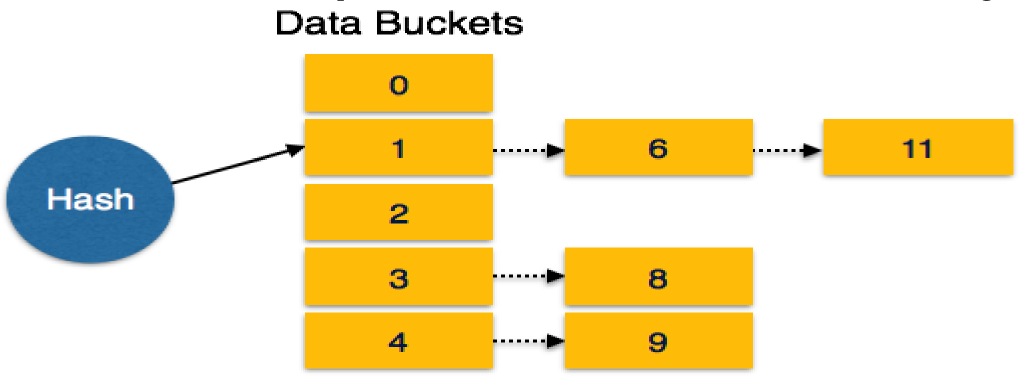


Operation

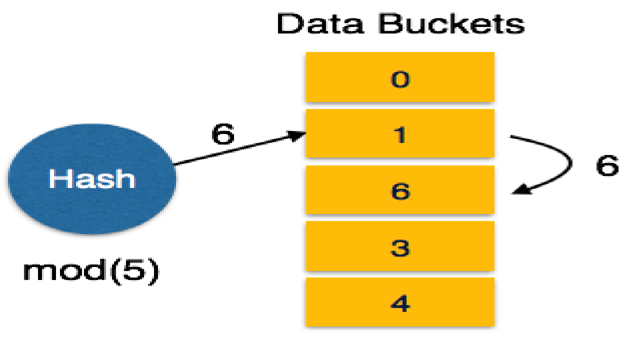
- Insertion − When a record is required to be entered using static hash, the hash function h computes the bucket address for search key K, where the record will be stored.

  Bucket address = h(K)

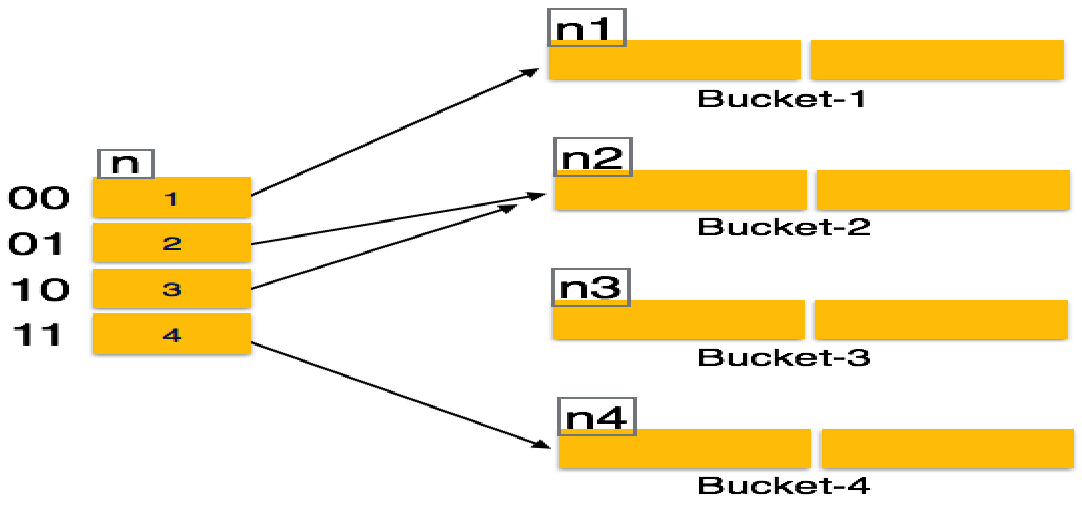
Search − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
Delete − This is simply a search followed by a deletion operation.
Bucket Overflow
The condition of bucket-overflow is known as collision. This is a fatal state for any static hash function. In this case, overflow chaining can be used.
Overflow Chaining − When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing.



Linear Probing − When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called Open Hashing.



Dynamic Hashing
The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing.
Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



Organization
The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed − that is, when all the buckets are full − then the depth value is increased linearly and twice the buckets are allocated.
Operation
Querying − Look at the depth value of the hash index and use those bits to compute the bucket address.
Update − Perform a query as above and update the data.
Deletion − Perform a query to locate the desired data and delete the same.
Insertion − Compute the address of the bucket
If the bucket is already full.
Add more buckets.
Add additional bits to the hash value.
Re-compute the hash function.
Else
Add data to the bucket,
If all the buckets are full, perform the remedies of static hashing.
Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary searches. This results in more I/O operations that are expensive in terms of time. File organizations based on the technique of *hashing* allows us to avoid accessing an index structure, yet indexing the file.

Hash File Organization

In a hash file organization, we obtain the address of the disk block (actually a *bucket* that can contain one or more records) containing the desired record by computing a function on the search-key value of the record.

If **K** is the set of all search-key values, **B** is the set of all bucket addresses, *h* is the hash function, we can computer the address of the bucket to insert a record with the search-key $\mathbf{K_i}$ using $h(\mathbf{K_i})$. Assuming there is space in the bucket, we can simply insert the record. We locate the record with the search-key $\mathbf{K_i}$ using $h(\mathbf{K_i})$. Deletion is done the same way.

However if it turns out the two records have the same hash value, $h(\mathbf{K_5}) = h(\mathbf{K_7})$, then we do a sequence search on the bucket for the record that is desired.

Hash Functions

The worst possible hash function maps all search-key values to the same bucket. The ideal hash functions will spread the records uniformly across all the buckets, so that all buckets have the same number of records.

At design time, we do not necessarily know what search-key values will be stored in the file. The typical hash function performs computation on the internal binary representation of the characters in the search key. It is critical that the hash function be as close to the ideal as possible, because it is the most efficient one for locating the data on a timely basis.

Handling of Bucket Overflow

When trying to insert a record into a full bucket causes an overflow and can be caused by either insufficient buckets or skew.

The number of buckets required is based on the maximum number of records stored divided by the number of records that will fit into a bucket. Rarely do we get that exact fit, the required number of buckets is increased by some fudge factor, typically 20% This wastes twenty percent of the space, but reduces that probability of having the overflow problem.

The skew can be caused when either multiple records have the same key or the chosen hash function results in nonuniform distribution of search keys.

Overflow can be handled by adding addition buckets to hold the records from the single bucket that overflowed (called *closed hashing*) or simply using some other bucket (*open hashing*).

An important drawback on hashing is that the hash function must be chosen when we implement the system and can not be easily changed afterwards,

Hash Indices

In addition to using hashing for file organization, the indices can be hashed.

Comparison of Ordered Indexing and Hashing

Each scheme has advantages in certain situations. And the DBMS implementor could leave the decision to the database designer and provide several methods. Normally, the implementor only provides a very limited number of schemes.

Typically, ordered indexing is used unless it is known in advance that range queries will be infrequent, in which case hashing is used. Hash organizations are particularly useful for temporary files created during query processing, if lookups on a key value are required and no ranges queries will be performed.

Index Definition in SQL

Creating an index:

CREATE INDEX <index-name> ON <relation-name> (<attribute-list>)

Example:

CREATE INDEX branch-index ON branch (branch-name)

Deleting an index:

DROP INDEX <index-name>

Query Processing: Overview

1. Parsing and translation

2. Optimization

3. Evaluation

1. Parsing and translation

- Translate the query into its internal form. This is then translated into relational algebra.
- Parser checks syntax, verifies relations.

2. Optimization

- A relational algebra expression may have many equivalent expressions. Each relational algebra operation can be evaluated using one of several different algorithm.

3. Evaluation

- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query.
  - Many factors contribute to time cost
  - "disk accesses, CPU, or even network communication.
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
    - Number of seeks * average-seek-cost
    - Number of blocks read * average-block-read-cost
    - Number of blocks written * average-block-write-cost
    - "Cost to write a block is greater than cost to read a block – data is read back after being written to ensure that the write was successful
- For simplicity we just use number of block transfers from disk as the cost measure
  - We ignore the difference in cost between sequential and random I/O for simplicity
  - We also ignore CPU costs for simplicity
- Costs depends on the size of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account
- We do not include cost to writing output to disk in our cost formulae