A Handbook on
Computer Science

9

Databases

CONTENTS

ER-model and Relational Model

1

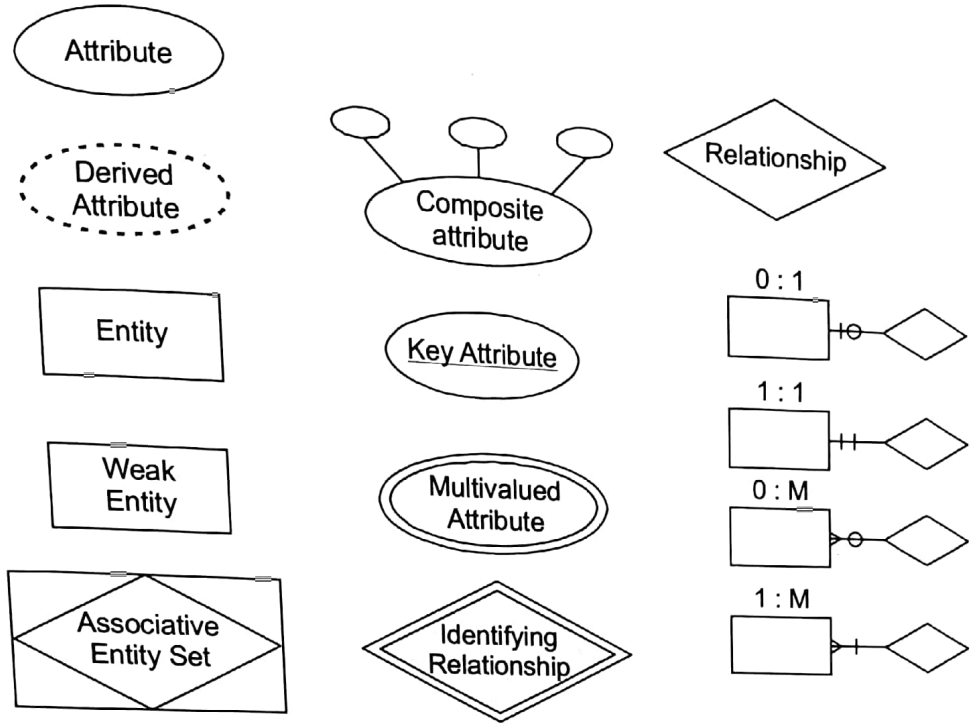
ER Model

- It is a logical tool which is used for database scheme design.
- It does not include implementation details.
- It is described by an ER (Entity-Relationship) diagram.
- **Entity:** An entity is a thing that has an independent existence. It is described by its attributes and determined by its instantiations (are particular values for its attributes).
- **Entity Set:** It is a set of entities of the same type, denote by a rectangular box in ER diagram. Entity can be identified by a list of attributes which are placed in ovals.
- **Relationship:** It is an association among several entities.
- **Relationship Set:** It is a set of relationship of the same type.
- **Superkey:** A superkey is a set of one or more attributes which collectively allow us to identify uniquely an entity in the entity set.
- **Candidate key:** A superkey for which no subset is a super key is called a candidate key.
- **Primary key:** Primary key is a candidate key chosen by the DB designer to identify entities in an entity set.
- **Weak entity set:** An entity set that does not possess sufficient attributes to form a primary key is called a weak entity set. Weak entity set is an entity set whose existence is dependent on one or more other strong entity sets.
- **Strong entity set:** An entity set that does have a primary key is called a strong entity set.

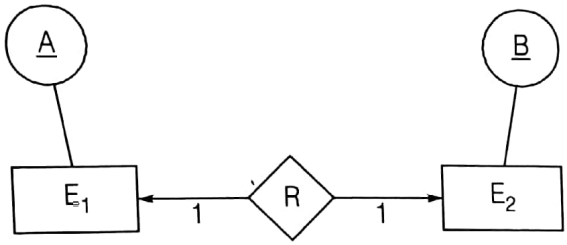
Structural Constraints

- **Degree:** Number of participating entity sets.
- **Cardinality constraints:** One-to-one, one-to-Many and Many-to-Many.
- **Participation constraints:** Partial or total.

ER Diagram Symbols

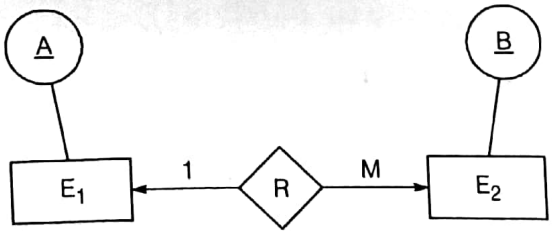


One-to-one



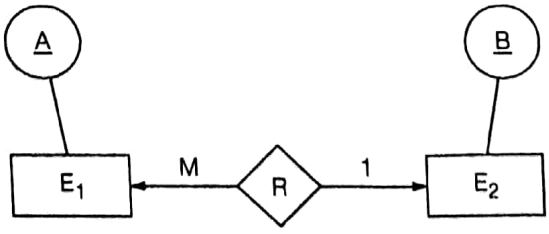
Candidate key of R : A, B
One-to-one

One-to-many



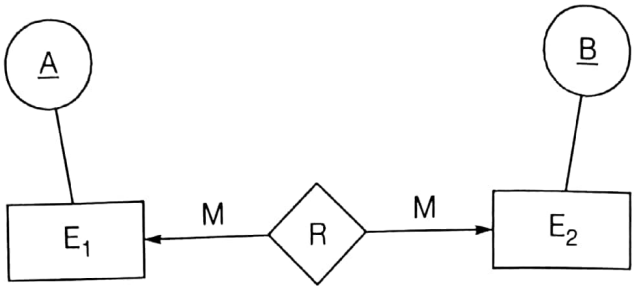
Candidate key of R : B
One-to-many

Many-to-one



Candidate key of R : A
Many-to-one

Many-to-many



Candidate key of R : AB

Many-to-many

- If we have cardinality one to many or many to one then, we can mix relational table with many involved table.
- If cardinality is many to many we cant mix any two tables.
- If we have one to one relation and we have total participation of one entity then we can mix that entity with relation table and if we have total participation of both entity then we can make one table by mixing two entities and their relation.

Commands

DDL Commands

CREATE, ALTER, DROP, GRANT, REVOKE, TRUNCATE AND COMMENT.

DOL Commands

COMMIT, SAVEPOINT, ROLLBACK, and SET TRANSACTION.

DML Commands

SELECT, INSERT, UPDATE, DELETE, CALL, LOCK TABLE AND EXPLAIN PLAN.

Relational Commands

Cartesian Product (×)

- Deg (R × S) = deg(R) + deg(s) and
- $|(R \times S)| = |R| \times |S|$

Natural Join ($\bowtie$)

- It is same as cross product + $\sigma_{\text{equality of common attribute}}$
- deg $(R_1 \bowtie R_2)$ = deg (R_1) + deg (R_2) – number of common attribute.
- Number of n-ary relation that can be formed over n-domains having n elements: n^n

- Number of equivalent representation of a relation having degree m and cardinality n are m! × n!

Orderby

Default ascending order.

Aggregation Operations

SUM, COUNT, AVG, MIN, MAX

Selection Operation (Restriction) (σ)

- $\deg(\sigma_c(R)) = \deg(R)$ (Number of attributes)
- $0 \le |\sigma_c(R)| \le |R|$ (Number of tuples)
- $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R)) = \sigma_{c_1,c_2}(R)$

Projection Operation (π)

- $0 \le \deg(\pi_A(R)) \le \deg(R)$
- $|\pi_{A_i}(R)| = |R|$

Rename (ρ)

$\deg(\rho_s(R)) = \text{degree}(R)$ and $|(\rho_s(R))| = |R|$

Union Operation

- Two relations are union compatible if they are of same degree and same domain.
- $\deg(R \cup S) = \deg(R) = \deg(S)$
- $\max(|R|, |S|) \le |R \cup S| < (|R| + |S|)$

Intersection Operation

- Two relation should be union compatible.
- $\deg(R \cap S) = \deg(R) = \deg(S)$
- $0 \le |(R \cap S)| \le \min(|R|, |S|)$

Set Difference (–)

- Two relation should be union compatible
- $\deg(R-S) = \deg(R) = \deg(S)$
- $0 \le |(R - S)| \le |R|$

Operation	Relational Algebra	Tuple Relational Calculas
Selection	σ_{cond} (R)	{T | R(T) AND Cond (T)}
Projection	$\Pi_{A_1, A_2, \ldots, A_k}$ (R)	{T | T.A_1, T.A_2, ..., T.A_k | R(T)}
Cartesian Product	R × S	{T | $\exists T_1 \in R$, $\exists T_2 \in R$(T.A_1=T_1.A_1 AND,..., T.A_n=T_1.A_n AND T.B_1=T_2.B_1 AND,..., T.B_m=T_2.B_m)}
Union	R ∪ S	{T | R(T) AND S(T)}
Set Difference	R – S	{T | R(T) AND $\forall T_1 \in S$, (T_1<>T)} where T<>T_1 is T.A_1<>T_1A_1OR... T.A_n<>T_1.A_n

■■■

Database Design

2

Database Design

Limitations of File System

1. To access database user should know about the physical details of the data.
2. Operating system fails to control concurrency when database is large.
3. Once got access user can access the whole data of the file system.

Database Design Goals

1. 0% redundancy
2. Loss-less join
3. Dependency preservation.

According to Codd

No two records of the table should be equal.

Difference between Primary Key and Alternative Key

1. At most one primary key is allowed for any relation but more than one alternative keys are allowed.
2. Null values are not allowed in primary key. Null values are allowed in alternative keys.
3. Primary keys used to design primary key, unique keys are used to design alternative keys.

Primary key: Unique + not null

Super key: Set of attributes used to differentiate all the tuples of the relation or super set of candidate key.

Example: Let R be the relational schema with n-attributes, $R(A_1, A_2 \ldots A_n)$.

Number of super keys possible

(a) With only candidate key $A_1 \rightarrow 2^{n-1}$

(b) With only candidate key $A_1, A_2 \rightarrow 2^{n-1} - 2^{n-2} + 2^{n-1}$

(c) With only candidate key $A_1A_2, A_3A_4 \rightarrow 2^{n-2} - 2^{n-4} + 2^{n-2}$

Functional Dependency

Let R be the relational schema and x, y be the non-empty set of attributes. t_1, t_2 are any tuples of relation then $x \rightarrow y$ (y functionally determined by x) if $t_1.x = t_2.x$ then $t_1.y = t_2.y$

Trivial Functional Dependency

If $x \supseteq y$ then $x \rightarrow y$ is trivial dependency.

Example:

- Sid → Sid
- Sid Same ⇒ Sid

Non-trivial Functional Dependency

If $x \cap y = \phi$ and if $x \Rightarrow y$ satisfy functional dependency definition.

Example:

Cid → Cname.

Armstrong's Axioms

1. Reflexive: if $x \supseteq y$ then $x \rightarrow y$
2. Transitivity: If $x \rightarrow y$ and $y \rightarrow z$ then $x \rightarrow z$
3. Argumentation: If $x \rightarrow y$ then $xz \rightarrow yz$
4. Splitting: If $x \rightarrow yz$ then $x \rightarrow y$, $x \rightarrow z$
5. Union: If $x \rightarrow y$ and $x \rightarrow z$ then $x \rightarrow yz$
6. Pseudo transitivity: If $x \rightarrow y$, $yw \rightarrow z$ then $xw \rightarrow z$

Attribute Closure (x^+)

Set of all attributes functionally determined by X.

Example:

R(ABCD) {A → B, B → C, C → D}

$(A)^+ = \{ABCD\}$

Functional Dependency Set closure (F^+)

Set of all trivial and non-trivial FD's derived from given FD set F.

$x \rightarrow y$ is logically applied in given FD set F only if x^+ should determine y.

Let R be the relational schema decomposed into R_1 and R_2. Given decomposition is loss-less only if

1. $R_1 \cup R_2 = R$

2. $R_1 \cap R_2 \neq \phi$
3. $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$.

Example:

R(ABCDE)

{A → BC, CD → E, B → D, E → A}

D = {ABC, ACDE} → Loss-less.

Properties of Decomposition

Loss-less Join Decomposition

Because of decomposition there should not be generation of any additional tuples.

$$\text{i.e. } R \equiv R_1 \bowtie R_2$$

Dependency Preserving Decomposition

Because of decomposition there should not be any loss of any dependency. Let R be the relational schema with functional dependency set F is decomposed into $R_1, R_2, \ldots R_n$ with FD sets $F_1, F_2, \ldots F_n$ respectively. In general $F_1 \cup F_2 \cup F_3 \ldots F_n$ can be $\subseteq F$.

If $F_1 \cup F_2 \ldots \cup F_n \equiv F$ then decomposition is preserving dependency.

NORMALIZATION

The normalization is especially meant to eliminate the following anomalies:

- Insertion anomaly
- Deletion anomaly
- Update anomaly
- Join anomaly

Goals of Normalisation

- Integrity
- Maintainability

Side Effects of Normalization

- Reduced storage space required (usually, but it could increase)
- Simpler queries (sometimes, but some could be more complex)
- Simpler updates (sometimes, but some could be more complex)

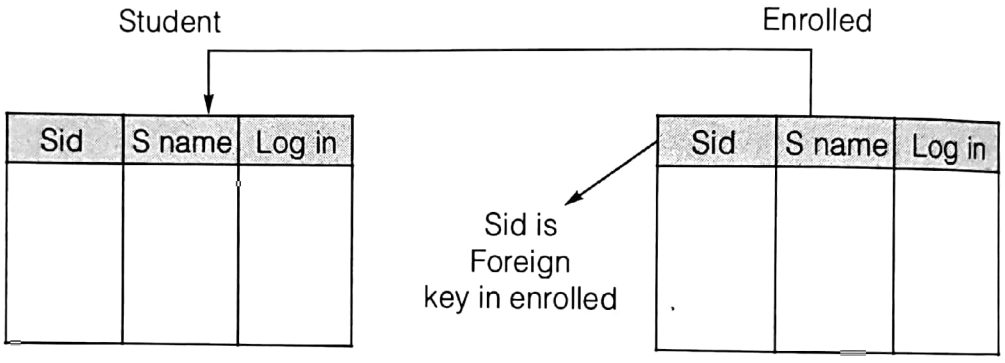
Example: F = {A → BC}, here F = {A → B, A → C} is simple.

- It is left reduced (removal of extraneous symbols)
  Example: F = {AB → C, B → C}, here B is extraneous attribute.
  So F = {A → C, B → C}.
- It is non-redundant (eliminating unnecessary FDs)
  Example: F = {A → B, B → C, **A → C**}, here A → C is redundant. Hence, F = {A → B, B → C}.

Referential Integrity Constraints

Foreign Key

Foreign Key is a set of attributes that references primary key or alternative key of the same relation or other relation.



Referenced Relation

1. **Insertion:** no violation.
2. **Deletion:**
   (a) **On delete no action:** Means if it causes problem on delete then not allowed to delete.
   (b) **On delete cascade:** If we want to delete primary key value from referenced table then it will delete that value from referencing table also.
   (c) **On delete set null:** If we want to delete primary key from referenced table then it will try to set the null values in place of that value in referencing table.
3. **Updation:**
   (a) On update no action
   (b) On update cascade
   (c) On update set null

Referencing Relation

- **Insertion:** May cause violation
- **Deletion:** No violation
- **Updation:** May cause violation

Example:

A	C
2	4
3	4
4	3
5	4
6	2

C is foreign key referencing A
Delete (2, 4) and on delete cascade

A	C
3	4
4	3
5	4

← Result

■■■

3

SQL

Introduction

- SQL stands for structured query language.
- SQL lets you access and manipulate databases.
- SQL is an ANSI (American National Standards Institute) standard.

Functions of SQL

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures and views.

RDBMS

- RDBMS stands for Relational Database Management System.
- RDBMS is the bases for SQL, and for all modern database systems such as MS SQL server, IBM DB2, Oracle, MySQL and Microsoft Access.
- The data in RDBMS is stored in database objects called tables.
- A table is a collection of related data entries and it consists of columns and rows.

Type of SQL

Two types of SQL:

- **DML:** Data Manipulation Language (SELECT)
- **DDL:** Data Definition Language (CREATE TABLE)

SQL Commands

- **SELECT Statement:**
  - (*i*) SELECT statement defines WHAT is to be returned (separated by commas)
  - (*ii*) "*" mean all columns from all tables in the FROM statement.

    Example: SELECT state-code, state-name.
- **FROM Statement:**
  - (*i*) Defines the Table(s) or View(s) used by the SELECT or WHERE statements.
  - (*ii*) Multiple Tables/Views are separated by commas.
- **WHERE Clause:**
  - (*i*) Defines what records are to be included in the query, it is optional.
  - (*ii*) Uses conditional operators:
    - (a) =, >, >=, <=, !=(<>)
    - (b) BETWEEN x AND y
    - (c) IN (list)
    - (d) LIKE '% string' ("%" is a wild-card)
    - (e) IS NULL
    - (f) NOT (BETWEEN/IN/LIKE/NULL)
- **SELECT DISTINCT Statement:**
  - (*i*) The SELECT DISTINCT statement is used to return only distinct (different) values.
  - (*ii*) In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.
  - (*iii*) **Syntax:** SELECT DISTINCT column_name, column_name FROM table_name;
- **AND & OR:**
  - (*i*) "AND" means all conditions are TRUE for the record
  - (*ii*) "OR" means at least one of the conditions is TRUE.
  - (*iii*) Multiple WHERE conditions are linked by AND/OR statements.
- **ORDER BY statement:**
  - (*i*) Defines how the records are to be sorted
  - (*ii*) Must be in the SELECT statement to be ORDER BY
  - (*iii*) Default is to order in ASC (Ascending) order. To sort the records in a descending order, use the DESC keyword.

(*iv*) **Syntax:** SELECT column_name, column_name FROM table_name ORDER BY column_name, column ASC/DESC;

- **Group Functions:**
  (*i*) Performs common mathematical operations on a group of records.
  (*ii*) Must define what constitutes a group by using the GROUP BY clause.
  (*iii*) All non-group elements in the SELECT statement must be in the GROUP BY clause.
- **INSERT INTO statement:**
  (*i*) Used to insert new records in a table.
  (*ii*) **Syntax:** INSERT INTO table_name VALUES (value1, value2,...);

  or

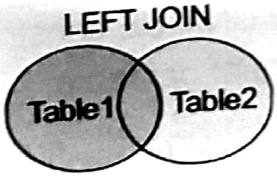
  INSERT INTO table_name (column1, column2,...) VALUES (value1, value2,...)
- **UPDATE statement:**
  (*i*) Used to update records in a table
  (*ii*) **Syntax:** UPDATE table_name SET column1=value1, column2=value2,... WHERE some_column = some_value

Note:

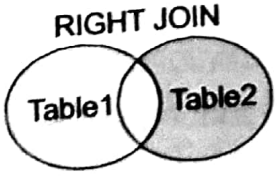
- The WHERE clause specifies which record or records that should be updated. IF WHERE clause is omitted, all records will be updated.

- **DELETE statement:**
  (*i*) Used to delete rows in a table.
  (*ii*) **Syntax:** DELETE FROM table_name WHERE some_column = some_value;
- **LIKE operator:**
  (*i*) The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
  (*ii*) **Syntax:** SELECT column_name(s) FROM table_name WHERE column_name LIKE pattern;
- **IN operator:**
  (*i*) The IN operator allows you to specify multiple values in a WHERE clause.
  (*ii*) **Syntax:** SELECT column_name(s) FROM table_name WHERE column_name IN (value1, value2, ...)

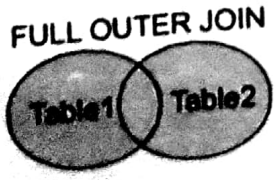
- **BETWEEN operator:**
  - (*i*) It selects values within a range. The values can be numbers, text, or dates.
  - (*ii*) **Syntax:** SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 and value2;
- **Aliases:**
  - (*i*) SQL aliases are used to temporarily rename a table or a column heading.
  - (*ii*) SQL aliases are used to give a database table, or a column in a table, a temporary name.
  - (*iii*) Basically aliases are created to make column names more readable.
  - (*iv*) **Syntax:** SELECT column_name AS alias_name FROM table_name AS alias_name;
- **Joining Tables:**
  - (*i*) Joins between tables are usually based on primary/foreign keys.
  - (*ii*) Make sure joins between all tables in the from clause exist.
  - (*iii*) List joins between tables before other selection elements.
    - (a) **Left outer join keyword:** It returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.



    - (b) **Right outer join keyword:** It returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.



    - (c) **Full outer join keyword:** It returns all rows from the left table (table1) and from the right table (table2). It combines the result of both LEFT and RIGHT joins.



- **SQL constraints:**
  - (*i*) SQL constraints are used to specify rules for the data in a table.
  - (*ii*) If there is any violation between the constraint and the data action, the action is aborted by the constraint.
  - (*iii*) Constraints can be specified when the table is created (inside the CREATE TABLE statement) or after the table is created (inside the ALTER TABLE statement).
  - (*iv*) In SQL, we have the following constraints:
    - **(a) NOT NULL:** Indicates that a column cannot store NULL value. It enforces a field to always contain a value. This means that one cannot insert a new record, or update a record insert a new record, or update a record without adding a value to this field.
    - **(b) UNIQUE:**
      - ♦ Ensures that each row for a column must have unique value.
      - ♦ The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.
      - ♦ A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

Note:

There might be many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

- **(c) PRIMARY KEY:** A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have an unique identity which helps to find a particular record in a table easily and quickly.
- **(d) FOREIGN KEY:** Ensure the referential integrity of the data in one table to match values in another table.
- **(e) CHECK:** Ensures that the value in a column meets a specific condition. It is used to limit the value range that can placed in a column.
- **(f) DEFAULT:** Specifies a default value when specified none for this column.

- **ALTER TABLE statement:**
  - (*i*) It is used to add, delete, or modify columns in an existing table.

(*ii*) **Syntax:**

(a) To add a column in a table:

ALTER TABLE table_name
ADD column_name datatype

(b) To delete a column from a table:

ALTER TABLE table_name
DROP COLUMN column_name

- **Aggregate Functions:**

(*i*) Aggregate functions perform a calculation on a set of values and return a single value.

(*ii*) Aggregate functions return the same value any time that they are called by using a specific set of input values.

(*iii*) Aggregate functions can be used as expressions only in the following:

(a) The select list of a SELECT (either a subquery or an outer query).

(b) A HAVING clause.

♦ **AVG( ) Function:**

The AVG( ) function returns the average value of a numeric column.

Syntax: SELECT AVG (column_name) FROM table_name

♦ **COUNT( ) Function:**

The COUNT( ) function returns the number of rows that matches a specified criteria.

COUNT (column_name) syntax:

It returns the number of values (NULL values will not be counted) of the specified column:

SELECT COUNT (column_name) FROM table_name;

COUNT (*) Syntax:

It returns the number of records in a table:

SELECT COUNT (*) FROM table_name;

COUNT (DISTINCT column_name) Syntax:

It returns the number of distinct values of the specified column:

SELECT COUNT (DISTINCT column_name) FROM table_name;

- **MAX( ) Function:**
  The MAX( ) function returns the largest value of the selected column.
  Syntax: SELECT MAX (column_name) FROM table_name;
- **MIN( ) Function:**
  The MIN( ) function returns the smallest value of the selected column.
  Syntax: SELECT MIN (column_name) FROM table_name;
- **SUM( ) Function:**
  The SUM( ) function returns the total sum of a numeric column.
  Syntax: SELECT SUM (column_name) FROM table_name;
- **LEN( ) Function:**
  The LEN( ) function returns the length of the value in a text field.
  Syntax: SELECT LEN (column_name) FROM table_name;
- **HAVING Clause:** The HAVING clause was added to SQL because the WHERE keyword could be used with aggregate functions.
- **GROUP BY Statement:** The GROUP BY statement is used in conjunction with the aggregate functions to group the result_set by one or more columns.

■■■

File Structures (B and B+ Trees)

4

Sequential File

- Blocking factor : $\left\lfloor \frac{\text{Block size}}{\text{Record size}} \right\rfloor$
- Number of record blocks : $\left\lceil \frac{\text{Total no. of records}}{\text{Blocking factor}} \right\rceil$
- Average number of blocks accessed by linear search:

$$\left\lceil \frac{\text{\# of record blocks}}{2} \right\rceil$$

- Average number of blocks accessed by binary search:

$$\lceil \log_2 \text{ \# of record blocks} \rceil$$

Index File

- Index blocking factor = $\left\lfloor \frac{\text{\# of record blocks} + 1}{2} \right\rfloor$
- First (single) level index blocks: $\left\lceil \frac{\text{\# of record blocks}}{\text{index blocking factor}} \right\rceil$
- Number of block accesses = $\lceil \log_2 (\text{first level index blocks}) \rceil + 1$

Indexing Types

1. **Single level Index:**
   - (a) **primary index (sparse):** Ordered with key field.
   - (b) **clustered index (sparse):** Ordered with non key field.
   - (c) **secondary index (dense):** Ordered with either key or non key field.
2. **Multilevel Index:**
   - (a) Indexed sequential access method
   - (b) B-tree index
   - (c) B+-tree index

Single Level Index

Dense Index

- For every data base record there exists entry in the index file.
- Number of data base records = Number of entries of the index file.

Sparse Index

- For set of database records there exists single entry in the index files.
- Number of index file entries < Number of database records
- Number of index file entries = Number of database blocks.

Clustered Index

- Search key should be used to physically order the DB.
- Search key is non-key.
- Atmost one clustering index is possible.
- Single level index blocks: $\left\lceil \frac{\text{No. of distinct values over non key field}}{\text{Index blocking factor}} \right\rceil + 1$
- Number of block accesses: $\lceil \log_2 (\text{single level index blocks}) \rceil + 1$

Secondary Index

- Search key is used to non ordered the DB file.
- Search key is primary or alternative key.
- Dense indexing is used.
- Number of block accesses $\lceil \log_2 (\text{single level index block}) \rceil + 1$
- Index blocking factor same for all indexes.

Multilevel Index (Static level index)

- Second level index is always sparse.
- Level 1 = "first level index blocks" computed by index.

$$\text{Level } 2 = \left\lceil \frac{\text{\# of blocks in level (1)}}{\text{index blocking factor}} \right\rceil$$

$$\vdots$$

$$\text{Level } n = \left\lceil \frac{\text{\# of blocks in level } (n-1)}{\text{index blocking factor}} \right\rceil = 1$$

- Number of levels = n
- Number of blocks = $\sum_{i=1}^{n}$(Number of blocks in level i)
- Number of blocks access = n + 1

B-tree (Bayes's/Balanced Search Tree)

- **Root node:** 2 to n children (pointers)
- **Internal node:** $\lceil n/2 \rceil$ to n children
- leaf nodes are all at same level.
- **Block Size** = p × (Size of block pointer) + (p – 1) × (Size of key field + size of record pointer)
- Minimum number of nodes = $1+\left(\frac{2\left[\lceil p/2 \rceil^{h}-1\right]}{\lceil p/2 \rceil-1}\right)$ height of root is 0
- Maximum number of nodes = $\frac{p^{h+1}-1}{p-1}$
- Minimum possible height = $\lceil \log_p \ell \rceil$ (ℓ : no. of leaves)
- Maximum possible height = $\lfloor 1+\log_{p/2} \ell/2 \rfloor$

+ Tree

It is same as B-tree.
All the records are available at leaf (last) level.
It allows both random and sequential access, but in B-tree only random access allowed.
All the leaf nodes are connected to next leaf node by block pointers [every leaf node has one block pointer]
Order of non-leaf Node:
[p × size of block pointer] + [(p – 1) × size of key field] ≤ Block size.
Order of Leaf Node:
$[(p_{leaf} - 1) \times$ (size of key field + size of record pointer) + (size of block pointer) ≤ Block size.]

■■■

Transactions and Concurrency Control

5

Transactions and Concurrency Control

Transaction Properties (ACID)

- Atomicity: Means execute all the operation or none of them.
- Consistency: Database should be consistent before and after the execution of the transaction.
- Isolation: Concurrent execution of two or more transaction.
- Durability: After commit, effect of transaction should persist.

Schedules

Sequences that indicate the chronological order in which instructions of concurrent transactions are executed.

Serial Schedule

- After commit of one transaction begins another transaction.
- Number of possible serial schedules with n transactions is n!
- No inconsistency.

Concurrent Schedule

- Simultaneous execution of two or more transaction.
- may result inconsistency.
- Better throughput and less response time.
- To maintain consistency transaction should satisfy ACID property

Conflicts in Concurrent Execution

- **WR problem:**

T_i	T_j
W(A)	⋮
⋮	R(A) → uncommited read

- **RW problem:**

T_i	T_j
R(A)	⋮
⋮	W(A)

- WW problem:

T_i	T_j
W(A)	⋮
⋮	W(A)

Classification of Schedule based on Serializability

- **Serializability:** A schedule is serialisable if it is equivalent to a serial schedule.
  - (*i*) Conflict serialisability
  - (*ii*) View serialisability
- **Conflict serialisability:** Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  - (*i*) I_i = read (Q), I_j = read (Q). I_i and I_j don't conflict.
  - (*ii*) I_i = read (Q), I_j = write (Q). They conflict.
  - (*iii*) I_i = write (Q), I_j = read (Q). They conflict.
  - (*iv*) I_i = write (Q), I_j = write (Q). They conflict.
- **Conflict equivalent:** If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.

  Conflict serialisable: We say that a schedule S is conflict serialisable if it is conflict equivalent to a serial schedule.

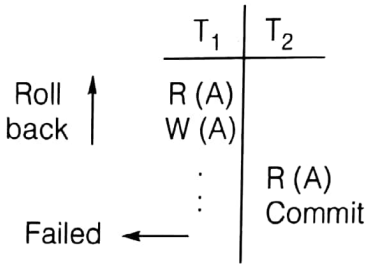
  View serialisability: Let S and S' be two schedules with the same set of transactions. S and S' are view equivalent if the following three conditions are met:
  - (*i*) For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.
  - (*ii*) For each data item Q if transaction T_i executes **read** (Q) in schedule S, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j.
  - (*iii*) For each data item Q, the transaction (if any) that performs the final **write** (Q) operation in schedule S must perform the final **write**(Q) operation in schedule S'.

Remember:

- Every conflict serialisable schedule is also view serialisable.
- Every view serialisable schedule that is not conflict serialisable has blind writes.

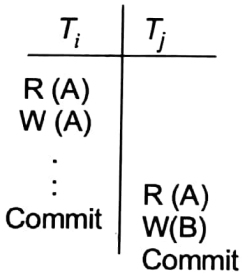
- **Irrecoverable Schedule:** Its not possible to roll back after the commitment of a transaction as the initial data is no where.

  Example:



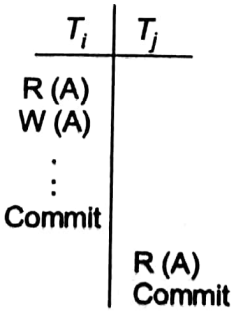
- **Recoverable Schedule:** A schedule is recoverable if a transaction T_j reads a data items previously written by transaction T_i, the commit operation of T_i appears before the commit operation of T_j.

  Example:



- **Cascadeless Recoverable Schedule:** 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 cascadeless schedule is also recoverable.

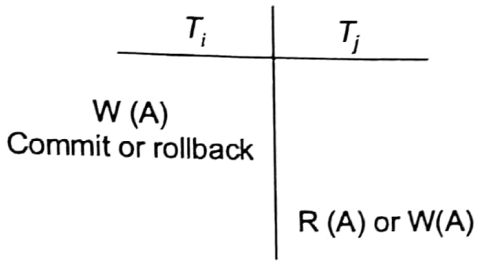
  Example:



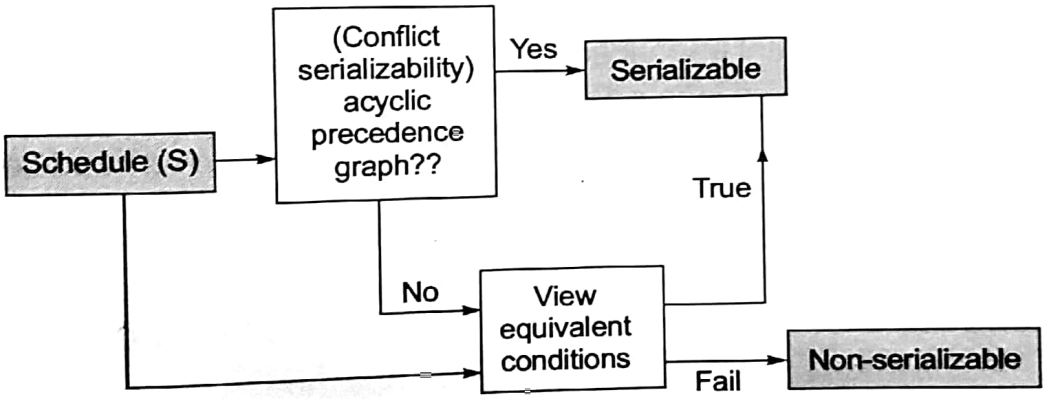
Cascading rollback: A single transaction failure leads to a series of transaction rollbacks.

- **Strict Recoverable Schedule:** If transaction T_i updates the data item A, any other transaction T_j not allowed to R(A) or W(A) until commit or roll back of T_i.

Example:



- S: $r_3(x)\, r_1(x)\, w_3(x)\, r_2(x)\, w_1(y)\, r_2(y)\, w_2(x)\, c_3 c_1 c_2$ is recoverable schedule but not cascadeless schedule.
- S: $r_1(x)\, r_2(z)\, r_1(z)\, r_3(x)\, r_3(y)\, w_1(x)\, w_3(y)\, r_2(y)\, w_2(z)\, w_2(y)\, c_1 c_2 c_3$ is irrecoverable.
- **Serializability checking:**



Concurrency Control with Locks

- Lock guarantees current transaction exclusive use of data item.
- Acquires lock prior to access.
- Lock released when transaction is completed.
- 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.

Lock Granularity

Lock granularity indicates level of lock use: database, table, page, row, or field (attribute).

Database Level

- The entire database is locked.
- Good for batch processes, but unsuitable for online multi-user DBMSs.

Table Level

- The entire table is locked.
- If a transaction requires access to several tables, each table may be locked.
- Table-level locks are not suitable for multi-user DBMSs.

Page Level

- A page has a fixed size and a table can span several pages while a page can contain several rows of one or more tables.
- Page-level lock is currently the most frequently used multi-user DBMS locking method.

Row-Level

- It allows concurrent transactions to access different rows of same table even if the rows are located on the same page.
- It improves the availability of data, but requires high overhead cost for management.

Field-Level

- It allows concurrent transactions to access the same row, as long as they require the use of different fields.
- The most flexible multi-user data access, but cost extremely high level of computer overhead.

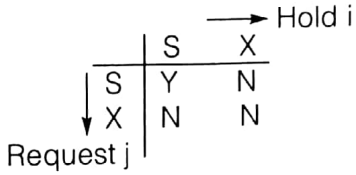
Lock Types

Binary Locks

- Two states: locked (1) or unlocked (0).
- Locked objects are unavailable to other objects.
- Unlocked objects are open to any transaction.
- Transaction unlocks object when complete.
- Every transaction requires a lock and unlock operation for each data item that is accessed.
- It locks before use of data item and release the lock after performing operation.
- Problems with binary locks: Irrecoverability, Deadlock and Low Concurrency level.

Shared/Exclusive Locks

1. **Shared (S Mode):**
   (*i*) Exists when concurrent transactions granted READ access.
   (*ii*) Produces no conflict for read-only transactions.
   (*iii*) Issued when transaction wants to read and exclusive lock not held on item.
2. **Exclusive (X Mode):**
   (*i*) Exists when access reserved for locking transaction.
   (*ii*) Used when potential for conflict exists.
   (*iii*) Issued when transaction wants to update unlocked data.



Note:

- **Lock-compatibility matrix:**
  (a) A transaction may grants a lock on item if the requested lock is compatible with locks already held on item by other transactions.
  (b) Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the no other transaction may hold any lock on the item.
- **Problems with Locking:**
  (a) **Transaction schedule may not be serialisable:** Managed through two-phase locking.
  (b) **Schedule may create deadlocks:** Managed by using deadlock detection and prevention techniques.

Two-Phase Locking

Two-phase locking defines how transactions acquire and relinquish locks.

1. **Growing phase:** Acquires all the required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.
2. **Shrinking phase:** Releases all locks and cannot obtain any new lock.

Governing rules of 2PL:

(*i*) Two transactions cannot have conflicting locks.
(*ii*) No unlock operation can precede a lock operation in the same transaction.
(*iii*) No data are affected until all locks are obtained.

Basic 2PL Protocol

- Equal serial schedule based on lock point.
- **Problems with basic 2PL:** Irrecoverability, Deadlock and Starvation.

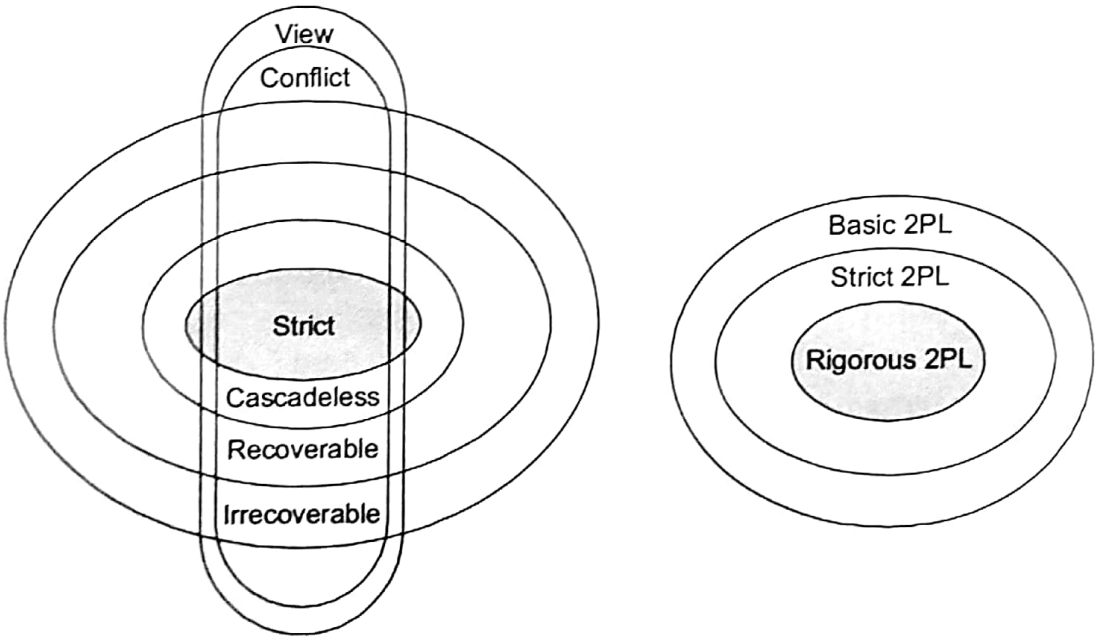
Strict 2PL Protocol

- Basic 2PL with all exclusive locks should be hold until commit/roll back.
- It ensures serializability strict recoverable.
- **Problems with strict 2PL:** Starvation and irrecoverability.

Rigorous 2PL protocol

- Basic 2PL with all locks (S/X) should be hold until commit.
- Equivalent serial schedule based on order of commit.

Class Diagram of Schedules with Serializability



Time Stamp Ordering Protocols

Assigns global unique time stamp to each transaction and produces order for transaction submission. Advantages with time stamping: Free from Deadlock and Ensuring serializability.

Basic Time Stamp Ordering Protocol

1. **Transaction T issues R(A) option:**
   (a) If WTS(A) > TS(T) then roll back T
   (b) Otherwise allow to execute R(A) successfully.
   Set RTS(A) = max {TS(T), RTS(A)}

2. Transaction T issues W(A) operation:
   (a) If RTS (A) > TS (T) then roll back.
   (b) If WTS (A) > TS (T) then roll back
   (c) Otherwise execute W(A) option set WTS (A) = {TS(T)}

Thomas Write Rule Stamp Ordering Protocol

1. Transaction T issues R(A) option:
   (a) If WTS (A) > TS (T) then roll back T
   (b) Otherwise execute successfully.
   Set RTS (A) = max {TS (T), RTS (A)}
2. Transaction T issues W (A) operation:
   (a) If RTS (A) > TS (T) then roll back T.
   (b) If WTS (A) > TS (T) then ignore W(A) and continue execution of trans T.

Strict Time Stamp Ordering Protocol

A transaction T_2 that issues a R(A) or W(A) such that $TS(T_2) > WTS(X)$ has its read write option delayed until the transaction T_1 that write the value X has committed or rolled back.

Wait Die Protocol

- Write the transaction in ascending order of time stamp values.
- If T_1 required resource that is hold by T_2, T_1 wait for T_2 to unlock.
- If T_2 required resource that is hold by T_1, then roll back T_2 and restart with same time stamp value.

Wound Wait Protocol

- Write the transaction in ascending order of TS values.
- If T_1 required resource that is hold by T_2 then roll back T_2 and restart with same time stamp value.
- If T_2 required resource that is hold by T_1 than T_2 wait for T_1 to unlock.
- Both wait die and wound wait protocols may have starvation.

■■■