GeeksforGeeks

GATE फर्रें CSE

DBMS

SHORT NOTES

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

STAR MENTOR CS/DA

KHALEEL SIR
ALGORITHM & OS
29 YEARS OF TEACHING EXPERIENCE

CHANDAN SIR
DIGITAL LOGIC
GATE AIR 23 & 26 / EX-ISRO

SATISH SIR
DISCRETE MATHEMATICS
BE in IT from MUMBAI UNIVERSITY

MALLESHAM SIR
M.TECH FROM IIT BOMBAY
AIR – 114, 119, 210 in GATE
(CRACKED GATE 8 TIMES)
14+ YEARS EXPERIENCE

VIJAY SIR
DBMS & COA
M. TECH FROM NIT
14+ YEARS EXPERIENCE

PARTH SIR
DA
IIIT BANGALORE ALUMNUS
FORMER ASSISTANT PROFESSOR

SAKSHI MA'AM
ENGINEERING MATHEMATICS
IIT ROORKEE ALUMNUS

SHAILENDER SIR
C PROGRAMMING & DATA STRUCTURE
M.TECH in Computer Science
15+ YEARS EXPERIENCE

AVINASH SIR
APTITUDE
10+ YEARS OF TEACHING EXPERIENCE

AJAY SIR
PH.D. IN COMPUTER SCIENCE
12+ YEARS EXPERIENCE

Module 1. Database design

Limitations of File System (Short Notes)

No Data Abstraction: Users must manage physical data access manually (no data independence).
Not Scalable: Suitable only for small datasets; inefficient for large databases.
No Concurrency Control: Cannot handle multiple users or concurrent transactions safely.
Single-User Access: Lacks multi-user support; no in-built locking or transaction management.
Data Redundancy & Inconsistency: No central control over data integrity or duplication.

Relational Model Terminology

Tuple : A tuple is a single row in a relation (table), representing a single record in that table.

Attribute : An attribute refers to a column in a relation. It defines the properties or characteristics of the relation.

Domain : The domain of an attribute is the set of all possible values that an attribute can take. It defines the data type and constraints of the attribute.

Degree : The degree of a relation is the total number of attributes (columns) it contains.

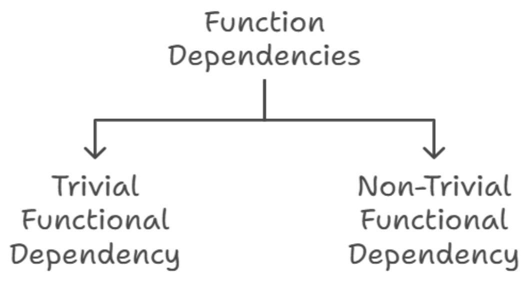
Cardinality : The cardinality of a relation is the total number of tuples (rows) present in it.

Relational Instance : A relation instance is a specific set of tuples that exist in a relation at a particular moment. It is a finite set that changes over time.

Relation Schema: A relation schema describes the structure of a relation, including its relation and the names and data types of its attributes.

Functional dependencies & Normalization

X —-> Y or X determines Y


Function Dependencies
Trivial Functional Dependency
Non-Trivial Functional Dependency

1. Trivial FD : Always valid, $X \rightarrow Y$ is trivial iff $X \supseteq Y$. Ex $AB \rightarrow A$
2. Non Trivial FD: $X \rightarrow Y$ is non trivial FD if $X \cap Y = \Phi$ and $X \rightarrow Y$ must satisfy FD definition. Ex $A \rightarrow B$

Armstrong's Axioms / Inference Rules

Armstrong's Axioms (or inference rules) provide a **sound and complete** set of rules for reasoning about **functional dependencies (FDs)** in relational database design.

Let X, Y, Z, W be sets of attributes. The following **three fundamental axioms** form the basis of all functional dependency inference:

Rule Name	Rule
Reflexivity	If $X \supseteq YX$, then $X \rightarrow Y$
Augmentation	If $X \rightarrow Y$, then $XZ \rightarrow YZ$
Transitivity	If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
Decomposition	If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
Union	If $X \rightarrow Y$, and $X \rightarrow Z$, then $X \rightarrow YZ$
Pseudotransitive	If $X \rightarrow Y$, and $WY \rightarrow Z$, then $WX \rightarrow Z$

Attribute closure $[X]^+$ Let x be the attribute set of relation R set of all possible attributes which are logically or functionally determined by X is called attribute closure of X $[X]^+$

Key Concept

Primary key: Randomly chosen candidate key is the primary key which is Unique and Not Null.

Alternate Key: All other keys are alternate keys from the set of candidate keys, except the primary key.

Super-key: If all attributes of relation R is determind by attribute (attribute set) closure of X $[X]^+$. The superset of the candidate key is known as the super key in a relation. Then X is super key.

Example: consider Emp-id as a candidate key in a relation Employee.
Then, {Emp-id, Emp-name} is a super-key.

Candidate key: The minimal superkey is defined as the candidate key.
R(ABCDE) [$AB \to C$, $C \to D$, $B \to EA$]
$[AB]^+ = [ABCDE]$, therefore AB is super key,
$[A]^+ = [A]$
$[B]^+ = [ABCDE]$, therefore B is the candidate key.

Key / Prime attribute: Set of attributes which belongs in any candidate key.

Non Key / Non Prime attribute: Set of attributes which does not belong in any candidate key.
Key/prime attribute = [B]
Non key/ Non Prime attribute = [A, C, D, E]
Note: To find the candidate key, first find the super key, then check the minimum of that super key.

Membership set
Let F be a given set of functional dependencies.
A functional dependency $X \to Y$ is said to be a member of F^+ (i.e., logically implied by F) iff
$X \to Y \in F^+ \Leftrightarrow Y \subseteq X^+$
Where X^+ is the **attribute closure** of X with respect to the set F.

Equality between 2FD set:
Let two FD set F & G give

$$\boxed{F \equiv G} \text{ iff } \begin{cases} \text{FCovers G : True} \\ \text{G Cover F : True.} \end{cases}$$

Finding Multiple candidate Key : First find any one candidate key in relation R, the attribute present in that candidate jey is key/prime attribute.

If $X_{\text{Attribute}} \to$ {Prime/Key Attribute}, then multiple candidate keys are possible.

Relation	F ⊃ G	G ⊃ F	F = G	Uncomaparable
F Covers G	True	False	True	False
G Covers F	False	True	True	False

Minimal cover: To eliminate redundant FD.

Minimal / Canonical cover: Sets of FD may have redundant dependencies that can be inferred from others.

Procedure to find canonical cover:
1. Split the FD such that RHS contain single attribute Ex. $A \to BC$: $A \to B$, $A \to C$
2. Find the redundant attribute on LHS and delete them.
3. Find the redundant FD and delete them from the FD set. { $A \to B$, $B \to C$, $A \to C$ } : { $A \to B$, $B \to C$ }

Note: Minimal cover may or may not be unique i.e. we can have more than one.

Finding number of super key: Let R be the relational schema with n attributes $A_1, A_2, \ldots A_n$
Total number of super keys = $\mathbf{2^{n-1}}$
Ex. With only CK A1, A2
Maximum number of candidate key = $^{n}C_{[n/2]}$, Where n is number of attributes.

Normalization

Why Normalisation?
To eliminate the following anomalies.
- Insertion Anomalies
- Updation Anomalies
- Deletion Anomalies

Properties of Decomposition

Lossless decomposition:

i) If ($R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n$) = R then it is **lossless join decomposition**

ii) If ($R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$) ⊃ R then it is **lossy**.

Dependency preserving decomposition:

i) If $\{F_1 \cup F_2 \cup F_3 \ldots \cup F_n\}$ = F, then decomposition is **dependency preserving**.

ii) If $\{F_1 \cup F_2 \cup F_3 \ldots \cup F_n\} \subset F$, then it is **not dependency preserving**.

Normal Forms

First Normal Form (1NF)

- A relation is in **First Normal Form (1NF)** if **it does not contain any multivalued or composite attributes**.
- All attributes must contain **atomic (indivisible) values**.
- By default, **RDBMS relations are in 1NF**.

  Example:
  If an attribute contains a list like {Math, English}, it's **not** in 1NF.
  Instead, split into multiple rows or use separate records.

Second Normal Form (2NF)

The second normal form (2NF) is based on the concept of *full functional dependency*. A functional dependency **X** → **Y** is a **full functional dependency** if removal of any attribute **A** from **X** means that the dependency does not hold any more; that is, for any attribute **A** ∈ **X**, (**X** − **{A}**) **does not functionally determine Y**. A functional dependency **X** → **Y** is a **partial dependency** if some attribute **A** ∈ **X** can be removed from **X** and the dependency still holds; that is, for some **A** ∈ **X**, (**X** − **{A}**) → **Y**. In Figure 15.3(b), {**Ssn, Pnumber**} → **Hours** is a full dependency (neither **Ssn** → **Hours** nor **Pnumber** → **Hours** holds). However, the dependency {**Ssn, Pnumber**} → **Ename** is **partial** because **Ssn** → **Ename** holds.

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

Third Normal Form (3NF)

- A relation is in **Third Normal Form (3NF)** if:

1. It is in **Second Normal Form (2NF)**, and
2. For every **non-trivial** functional dependency X→YX \rightarrow YX→Y, at least one of the following holds:
   Either X is a **superkey**, or
   Y is a **prime attribute** (i.e., part of some candidate key).

   Non-trivial FD: X→Y is non-trivial if X ∩ Y = Φ & must satisfy FD definitions

Why 3NF?

Eliminates **transitive dependencies**.
Allows some redundancy to preserve **dependency preservation** and **lossless decomposition**.

Boyce-Codd Normal Form (BCNF)

- A relation is in **BCNF** if:
  For **every non-trivial functional dependency** X→Y
  → X must be a **superkey**.

  This is **stricter than 3NF** because it does **not allow any dependency** where the determinant is **not** a superkey, even if Y is a prime attribute.

Quick Note:

- If a relation has **only two attributes**, it is always in **BCNF** regardless of the dependency.

Design goal	1NF	2NF	3NF	BCNF
0% Redundancy	No	No	No	YES(Suffers from multivalued attributes)
Lossless Join	yes	yes	yes	yes
Dependency Preservation	yes	yes	yes	May or may not be preserved

Note:

2NF Checking	3NF Checking	BCNF Checking
[Proper Subset] of Ck. → [non key] [attribute] ∴ Not in 2NF.	R is in 3NF if every non-trivial FD must satisfy the following: Either x: Super Key , or y: key/prime attribute	R is in BCNF if every X → A non-trivial FD must satisfy the following Condition: X: Super Key.

Module: 2 ER Model

A description of data in terms of a data model is called schema.
Entity relationship set: It is used to relate two or more entity sets.
Entity: Real-world object (e.g., Student).
Strong Entity: Exists independently; has a primary key.
Weak Entity: Depends on strong entity; no primary key.

Attributes: Describe properties of an entity.
Simple (Atomic), **Composite**, **Derived**, **Multivalued**.

Relationship: Association among entities.
Degree of Relationship Set: Specifies the number of Entity Set participate in a relationship set.
Mapping Cardinality: 1:1, 1:N, M:1, M:N.

Participation:
Total (Double line): Every entity must participate.
Partial (Single line): Optional participation.

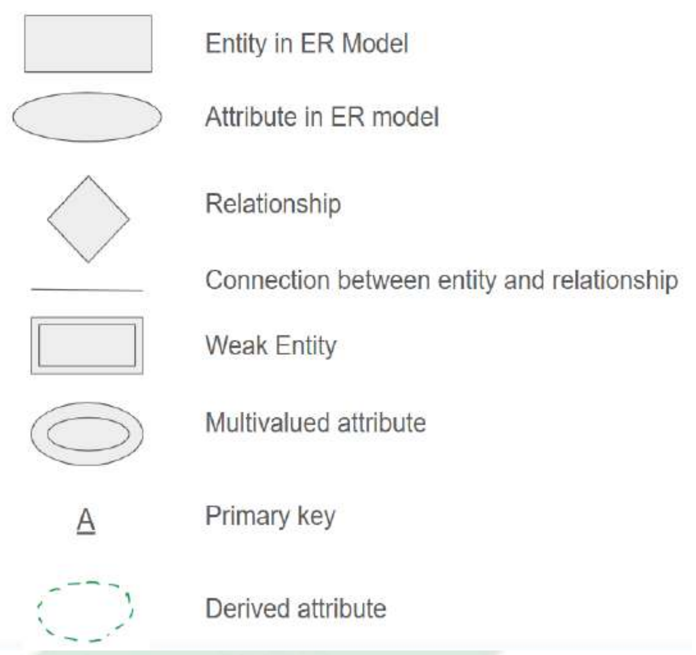
ER to Relational Mapping

Strong Entity → Relation with primary key.
Weak Entity → Include partial key + key attribute of strong(Owner) Entity set
1:1 Relationship → Foreign key in either entity.
1:N Relationship → Foreign key in "N" side.
M:N Relationship → Create separate relations with foreign keys.
Multivalued Attribute → Create new relation.

Aggregation → Treat relationship set as entity.

Keys:
Primary Key: Unique identifier for entity.
Discriminator/Partial Key: Used in weak entity sets.

Partial participation on both side of binary relationship



ER	RDBMS
One : One	Merge relationships set any one side. 2 table required
1 : M	Merge relationship set towards many sides. 2 table required
M : M	Separate table for each entity set and relationship set. 3 table required
M : 1	Merge relationship set towards many side. 2 table required

Full participation on "one" side of many to one relationship
Merge the entities and relationships set into a single relational table. So, 1 table.

Full participation on "Many" side of Many-to-one relationship
Merge relationship set towards many sides. So, 2 relational tables.
Full participation on any "one" side in one-to-one relationship
Merge the entity sets and relationship sets into a single table. So, 1 table.

Full participation on any "Many" side of Many-to-Many relationship
Merge relationship set towards any "Many" side of relationship. So, 2 table.

Module - 3 Relational Model and SQL

Derived Operator
1. JOIN & its type
2. Intersection
3. Division

Relational Algebra (RA)
Basic Operators:
Selection (σ), Projection (π), Rename (ρ)
Set Operations:
Union (∪), Intersection (∩), Difference (−)
Cartesian Product (×)
Joins:
Natural Join (⋈), Theta Join, Outer Join (Left, Right, Full)

$\pi_{\text{attribute\_name}}$ (R): It is used to project required attributes from relation R.

σcondition(p) (R): It is used to select records from relation R, those satisfied by the condition (P).

Cross product (×): R × S - It results in all attributes of R followed by all attributes of S, and each record of R paired with every record of S.

Degree (R × S) = Degree (R) + Degree (S)

Note: Relation R with n tuples and Relation S with 0 tuples then number of tuples in R × S = 0 tuples

Join (⋈):
1. Natural join (⋈)
R⋈S distinct attributes(equality between common attributes of R and S (R × S))
deg(R ⋈ S) = deg(R) + deg(S) - number of common attribute.

Note: Natural join is equal to cross-product if the join condition is empty.
Number of n-ary relation that can be formed over n-domains having n elements : n^n
Number of equivalent relation
Conditional Join ($\bowtie_c$)
$R \bowtie_c S \cong \sigma_c (R \times S)$

Outer Joins:
(a) LEFT OUTER JOIN
R ⟕ S : It produces
(R ⋈ S) {Records of R those are failed join condition with remaining attributes null}
(b) RIGHT OUTER JOIN (⟖)
R ⟖S : It produces]
(R ⋈ S) {Records of S those are failed join condition with remaining attributes null}
(C) FULL OUTER JOIN (⟗)
R ⟗ S = (R ⟕ S) (R ⟖ S)

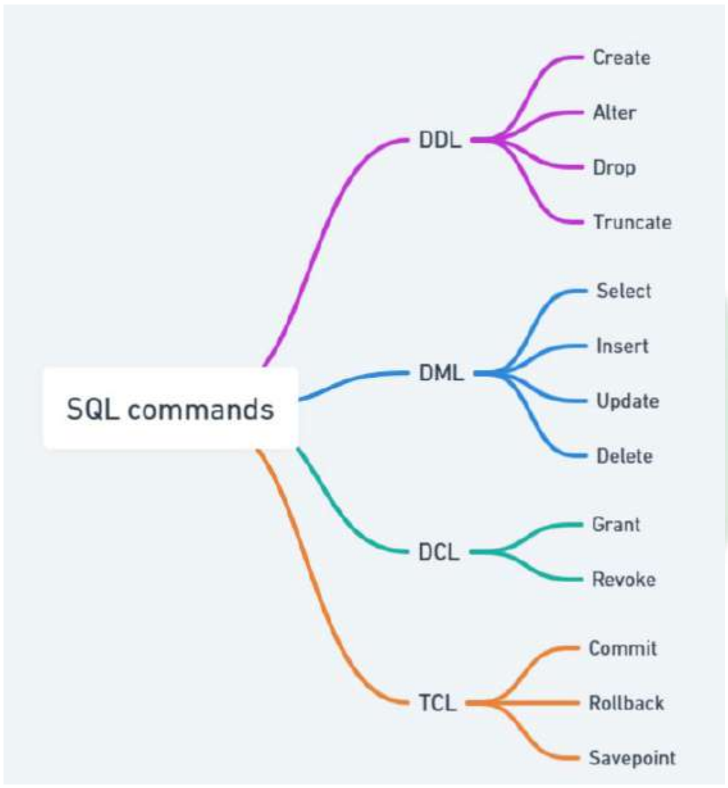
Rename(ρ): It is used to rename table names and attribute names for query processing.
Example:
(I) Stud (Sid, Sname, age)
ρ(Temp, Stud) : Temp (Sid, Sname, age)

Division: The **division** operation is used in relational algebra to find tuples in one relation that are associated with **all** tuples in another relation.
Given two relations: **R(A, B), S(B)**
The operation **R ÷ S** returns those values of **A** such that for **every** value of **B** in **S**, the pair **(A, B)** is present in **R**.
Expansion of Division Using Basic Operators:
1. Compute the **cross product** of all student IDs with all course IDs: **πsid(Enroll)×πcid(Course)**
   This gives all possible enrollments that should exist if each student were enrolled in every course.
2. Subtract the actual Enroll relation: **(πsid(Enroll)×πcid(Course))−Enroll**
   This yields the set of (sid, cid) pairs that *do not exist* in the Enroll relation.
3. Project the student IDs from the result of Step 2: **πsid((πsid(Enroll)×πcid(Course))−Enroll)**
   These are the students who are not enrolled in every course
4. Subtract this result from the set of all student IDs:
   πsid(Enroll)−πsid((πsid(Enroll)×πcid(Course))−Enroll)
   This final expression gives the **students who are enrolled in every course**.

Result : **$\pi sid(Enroll) - \pi sid((\pi sid(Enroll) \times \pi cid(Course)) - Enroll)$**

Set Operator:
Union:
R and S relations union compatible iff-
(i) Arity of R equal to Arity of S and
(ii) Domain of attributes of R must be the same as domain of attributes of s respectively.

Structural Query language



DDL (Data Definition Language): Used to **define or modify the structure** of database schema objects (like tables, views, indexes).
CREATE: Creates a new table, view, or other object.
ALTER: Modifies an existing object (e.g., add a column).
DROP: Deletes an existing object.
TRUNCATE: Removes all rows from a table quickly (no rollback).

DML (Data Manipulation Language): Used to manipulate data in existing tables.
SELECT: Retrieves data from one or more tables.
INSERT: Adds new rows into a table.
UPDATE: Modifies existing rows in a table.
DELETE: Removes rows from a table.

DCL (Data Control Language): Used to **control access and permissions** to the database.
GRANT: Gives privileges to users.
REVOKE: Withdraws previously granted privileges.

TCL (Transaction Control Language): Used to **manage transactions** and control their execution.
COMMIT: Saves all changes made in the current transaction.
ROLLBACK: Undoes all changes since the last commit.
SAVEPOINT: Sets a point within a transaction to which a rollback can be done.

Relational Query and SQL query
SQL : SELECT DISTINCT A1, A2, ... An
FROM R_1, R_2, ... R_n
WHERE P;
Equivalent RA: $\pi_{A1, A2.....An} (\sigma(R_1 \times R_2 \times ....R_n)$

SQL
SELECT DISTINCT A1, A2, ... An
FROM R1, R2, ... Rn
WHERE condition
GROUP BY (attributes)
HAVING condition
ORDER BY (attributes) (DESC)
Execution Flow
FROM cross-product of relations
WHERE Selection operator () to apply condition for each record
GROUP BY
HAVING
SELECT
DISTINCT ORDER BY

GROUP BY:
• It is used to group records data based on specific attributes.
• It GROUP BY clause used then
(a) Every attribute of the GROUP BY clause must be selected in SELECT clause.

(b) Not allowed to select any other attribute in the SELECT clause.
Allowed to select aggregate function along with GROUP BY attribute in SELECT Clause.

HAVING Clause

• HAVING clause must be followed by GROUP BY clause.
• HAVING clause used to select groups that are satisfied having clause condition.
• HAVING clause condition must be over aggregation function such as some ( ), Every ( ) etc. but not allowed direct attribute
comparison

Note: Every Query which is written by using HAVING clause can be re-written by using WHERE clause.

Nested Query Without Co-relations: Inner query independent of outer query.

SQL Constraints
Key Properties:

Constraints can be defined:
During **table creation** (CREATE TABLE)
After table creation using **ALTER TABLE**
If any data violates a constraint, the database system **rejects the operation** (insert/update).

1. Not Null: Ensures that a column **must always hold a value**
2. Unique: Ensures **all values are distinct** in the column or column set
3. Primary Key: Combines NOT NULL + UNIQUE Uniquely identifies each record. Can consist of one or more columns (composite key). Each table can have **only one** PRIMARY KEY
4. Foreign Key: Maintains **referential integrity** Enforces a relationship between columns in two tables
   The referencing column(s) must match values in the referenced table's PRIMARY or UNIQUE key. Can be NULL if not marked NOT NULL
5. Check :Restricts the **range or condition** of values in a column
6. Default: Specifies a **default value** when none is provided during insert

ALTER TABLE Statement

The ALTER TABLE command is used to modify the structure of an existing table.
Add a new column
Drop (delete) a column
Modify column data type, rename columns, rename table (DBMS-specific)
Aggregate Functions: Aggregate functions perform **calculations over a set of rows** and return a **single summary value**.

1. Avg() - Mean value of numeric column
2. Count() - Returns the **number of rows** that match a condition.
3. Sum() - Returns the total values
4. Min() - Returns the smallest value
5. Max() - Returns the largest value

Nested Query Without Co-relations: It is independent of the outer query.

Inner Query → Runs independently and returns a result.
Outer Query → Uses the inner result to complete execution.
Example:

```
SELECT name
FROM employees
WHERE dept_id IN (
   SELECT id
   FROM departments
   WHERE location = "Delhi"
);
```

Co-related Nested Query
In Nested Co-related query inner query uses attributes from outer query tables.
In Co-related Nested query inner query allowed in WHERE, HAVING clause of outer query.
Example:

```
SELECT A
        FROM R
        WHERE (SELECT count(*)
               FROM S
               WHERE S.B < R.A) < 5;
```

NOTE: If co-relation in WHERE clause then inner query re-computes for each record of outer query From

clause. If correlation in HAVING clause then inner query re-computes for each group of outer query.
Function used for Nested Query

1. IN / Not IN:
2. ANY: Compares a value with **each value returned by the subquery**. Used with comparison operators: =, <, >, <=, >=, <>
3. ALL: Compares a value with **all values** returned by the subquery.
4. EXIST / NOT EXISTS: Check **whether a subquery returns any rows**. (Result empty or non-empty)

SQL Join query

```
SELECT DISTINCT T1.eid
FROM Emp T1
JOIN Emp T2 ON T1.sal > T2.sal
WHERE T1.gen = 'Female' AND T2.gen = 'Male';
```

SQL Nested Query

```
SELECT eid
FROM Emp
WHERE gen = 'Female'
  AND sal > ANY (
      SELECT sal
      FROM Emp
      WHERE gen = 'Male'
  );
```

SQL Co-related Nested Query

```
SELECT eid
FROM Emp T1
WHERE T1.gen = 'Female'
  AND EXISTS (
      SELECT *
      FROM Emp T2
      WHERE T2.gen = 'Male'
        AND T1.sal > T2.sal
  );
```

Module - 4 File Structure & Indexing (B and B+ tree)

Indexing Types

Block Factor of Index = ⌊ (B - H) / (k + P) ⌋ entries/block
where , B = Block size (in bytes)
H = Overhead per block (e.g., block headers, pointers, etc.)
k = Number of fields per index entry
P = Size of one field (in bytes), or size of one index entry

1. Single-Level Index

A single-level index contains a sorted list of index entries, each pointing to a block or record in the data file.
(a) Primary Index (key + ordered file)
(b) Clustered index (non key + ordered file)
(c) Secondary Index (key/non key + unordering file)

At most **one primary index** per relation. Any number of **secondary indexes** can be created on non-primary key attributes.
A relation can have **either** a **primary (sparse)** index **or** a **clustered** index, but **not both** on the same attribute.
A clustered index determines the physical order of records; hence only one clustered index per relation.
If the number of levels = k, and we also access the data block, Total block accesses = k +1
Then total number of block accesses to locate a record is k + 1

Multilevel Index

A **multilevel index** treats the index file itself as an ordered file (first or base level). A **second-level index** is then created on this first-level index, acting like a **primary index** using **block anchors**—one entry per block of the first level. Since all index entries are of the same size, the **blocking factor** remains the same across all levels.

B Tree

Spanned organisation	Unspanned organisation
A record stored in more than 1 block	Record can be stored in a particular block
Blocking factor = Block size / Record size	Blocking factor = ⌊Block size / Record size⌋
No memory wastage but block access cost increase	Block access cost reduced but wastage of memory

Every internal node except the root node contains at least (min) ⌈P/2⌉ block pointers (min keys ⌈P/2⌉ - 1) and maximum P block pointers (maximum keys P - 1).
Root can contain at least 2 block pointers (min 1 key in root node) and maximum P block pointers (max P - 1 keys).
Keys within the node should be in ascending order.
Each leaf node should be at the same level.
A **B-tree** maintains balance and prevents excessive space waste due to deletion by enforcing specific constraints.
Internal Node structure <P1, <K1, Pr1>, P2, <K2, Pr2>, ..., <Kq−1, Prq−1>, Pq>
Pi is a **tree pointer** (points to a subtree).
Kj is a **search key**.
Prj is a **data pointer** (points to the actual data record or block for key Kj).
q is the number of keys in the node (q ≤ p).

B - Tree Formulas:

Order : P

	Minimum	Maximum
Root (B_P)	2	P
Non - Root (B_P)	⌈P/2⌉	P
Root (Keys)	1	P - 1
Non Root (Keys)	⌈P/2⌉ - 1	P - 1

Order P:
P * B_P + P(Keys + R_P) <= Block size.
(P + 1) B_P + P(Keys + R_P) <= Block Size
Note: Maximum level: At each level min # key & min # Bp (⌈P/2⌉−1, ⌈P/2⌉)
Minimum level: At each level maximum # key & max # Bp (P−1, P)
Applicable for both B and B+ Tree

B+ Tree

All keys are available at leaf node

Internal node: No read pointer.
Leaf node contains Key & Record pointer + 1 Block pointer

Structure of Internal node:

B_1	K_1	B_2	K_2	...	B_{p-1}	K_{p-1}	B_p

Left biasing ⇒
If $x \le K_1$
If $K_1 < x \le K_2$

Right biasing ⇒
If $x < K_1$
If $K_1 \le x < K_2$

Formula: $P * BP + (P - 1) * Key \le$ Block Size
Structure of Leaf node:

K_1R_1	K_1R_2	....	$K_{p-1}\ R_{p-1}$	Bp

Formula: $(P - 1) *[\ Key + R_p] + 1 * BP \le$ Block Size
In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer.
In a B+ tree, data pointers are stored only at the leaf nodes of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes.
B+ Tree is best suited for sequential access to a range of data records.

Chapter 5: Transaction and Concurrency Control

A transaction is a collection of operations that forms a single logical unit of work.

Operation in Transaction

Read (A) - This operation transfers the current value of data item **A** from the database (typically stored on disk) into a local variable in the transaction's main memory workspace. It enables the transaction to access and use the value of **A** during its execution.

Write (A) - This operation stores the modified value of data item **A** from the transaction's local workspace (in main memory) back into the database (on disk), thereby making the changes made by the transaction visible in the database.

BEGIN TRANSACTION - Marks the starting point of a transaction. Allocates resources and begins logging.
END TRANSACTION - Marks the logical end of the transaction. Actual commit or abort happens later.
Commit - Makes all changes made by the transaction **permanent** in the database.
Abort - Undoes all changes made by the transaction and restores the database to its previous consistent state.

Atomicity : Atomicity ensures that a transaction is *all-or-nothing* — either all operations of the transaction are executed successfully, or none of them are.

Goal: Prevent **partial updates** that can lead to data inconsistency.

Maintained by: Recovery subsystem using **undo** (rollback) operations.

Operation	Purpose	Affects
Redo	Reapply changes of committed transactions	Committed Transactions
Undo	Revert changes of uncommitted transactions	Uncommitted Transactions
Checkpoint	Limit redo/undo scope for recovery	All transactions since last checkpoint

System crash - If System crash/failure happen, required operation to recover are
(I) All committed transactions until the previous checkpoint will perform Redo.
(II) All uncommitted transactions in the entire system will perform undo.
(III) Clean all log entries.
Roll back - Undo modification of database files which are done by failure transaction.
Durability - If a transaction completes successfully and the user is notified, the changes made by the transaction must **persist**, regardless of what happens next.

Durability Mechanisms

Write-Ahead Logging (WAL): Log records are written before actual data is updated.

Redo Logging: During recovery, the system reapplies the effects of committed transactions using the log.

Stable Storage: Assumes existence of a reliable storage medium that survives crashes (or is emulated using replication, RAID, etc.).

Consistency - It ensures all integrity constraints are preserved.

Isolation - Each transaction should appear to execute in isolation, even when multiple transactions are running concurrently.

Isolation issues - Dirty read, Non-repeatable read, Phantom read.

Schedules - Time order execution sequence of two or more transactions.

Schedules

Schedule: Sequence of operations from multiple transactions.
Serial Schedule: One transaction executes completely before another.
Concurrent Schedule: Interleaved operations of transactions.
Serializable Schedule: Equivalent to a serial schedule.

Types of Serializability

Conflict Serializability

Uses Precedence Graph.

No cycles ⇒ Conflict serializable. (CNC)

Conflict Operations:

Read–Write Problem

T1	T2
W(A)	
	R(A)

- Write–Read

T1	T2
R(A)	
	W(A)

- Write–Write

T1	T2
W(A)	
	W(A)

View Serializability

- Three checks:
  - Initial read
  - Updated read
  - Final write

- Conflict serializable ⇒ View serializable, but not vice versa.

Why Concurrent Execution?

- **Increases throughput** of the system
- **Maximizes resource utilization**
- **Reduces waiting time** for users

T1	T2
W(A)	
	R(A)

Schedule type	Property
Irrecoverable	T2 reads dirty data from T1 and commits before T1
Recoverable	T2 commits after T1 commits.
Cascading Rollback	T2 reads from uncommitted T1; failure of T1 rolls back T2.
Cascadeless	T2 reads data from committed T1 only.
Strict Schedule	No overwrites on uncommitted data. Guarantees atomicity.

Concurrency Control with Locks

Lock-Based Protocols

- **Locks** ensure *mutual exclusion*, allowing only one transaction to access a data item in a conflicting mode at a time.
- A transaction **must acquire a lock** on a data item before accessing it.
- Locks are **automatically released** when the transaction commits or aborts.
- **Locking protocols** are sets of rules that govern how locks are acquired and released to **ensure serializability**.
- Improper use of locks may lead to issues like **deadlocks**, **starvation**, and reduced concurrency.
- Locking restricts **non-serializable schedules** by design.

Lock Granularity

Granularity refers to the **size of the data item** that can be locked. Coarser granularity reduces overhead but limits concurrency, while finer granularity increases overhead but allows higher concurrency.

Database-Level Locking

- Locks the entire database.
- **Use Case**: Suitable for batch jobs or bulk updates.

- **Drawback**: Poor concurrency – only one transaction can access the DB.

Table-Level Locking

- Locks the entire table.
- **Use Case**: Useful when transactions access entire tables.
- **Drawback**: Reduces concurrency if multiple transactions access different rows.

Page-Level Locking

- Locks a **disk page** (unit of storage, typically 4KB–8KB).
- A page may hold **multiple rows** or **parts of a table**.
- **Use Case**: Balanced trade-off between concurrency and overhead.
- Most **popular in multi-user DBMS**.

Tuple/Row-Level Locking

- Locks individual tuples (rows).
- **Use Case**: Allows maximum concurrency.
- **Drawback**: Higher overhead (more locks, more tracking).

Attribute-Level Locking

- Locks a specific column/attribute.
- **Rarely used** in practice due to high complexity.

Row-Level Locking

- Locks individual **tuples (rows)** in a table.
- Allows **maximum concurrency**, as multiple transactions can access different rows.
- Even if rows are on the **same page**, they can be accessed in parallel.
- **Overhead**: High, due to managing a large number of locks.
- Common in high-concurrency systems like **banking** and **real-time OLTP**.

Field-Level Locking (Attribute-Level)

- Locks **specific attributes/columns** within a tuple.
- Allows **highest concurrency**, since multiple transactions can access different fields of the same row.
- **Rarely used in practice** due to:
  - Complexity in implementation
  - Very **high CPU and memory overhead**
- Mostly **theoretical** and not supported in common DBMS implementations.

Lock types

1. Binary locks

0 → **Unlocked** (available)
1 → **Locked** (unavailable)
Every transaction must lock before accessing a data item and unlock after operation.

Issues :

Doesn't distinguish between read/write intent.
Leads to **low concurrency**.
Susceptible to **deadlocks** and **irrecoverability**.

2. Shared/Exclusive Locks

Shared (S Mode)
Allows **read-only** access to a data item.
Multiple transactions can hold shared locks **simultaneously**.
No conflict between transactions if all are **reading only**.
Conflict when transaction wants to read an exclusive lock not held on item.

Lock compatibility matrix

- A transaction may be granted a lock on a data item **if the lock mode is compatible** with the existing locks held by other transactions.
- **Multiple transactions** can simultaneously hold **Shared (S)** locks on the same item (read-only).
- If **any transaction holds an Exclusive (X)** lock, **no other transaction** can obtain either shared or exclusive lock on that item.

Request ↓ /Held →	S	X
S	Lock can be granted	Lock request must wait
X	Lock request must wait	Lock request must wait

Problem with locking

1. **Non-Serializable Schedules**
   Problem: Concurrent execution might lead to **inconsistent** results if not carefully managed.
   Solution: Enforce **serializability** using two-phase locking protocol.
2. Schedule may create deadlocks.
   Problem: Two or more transactions wait **indefinitely** for each other's locks.
   Solution: Use **deadlock handling techniques**: Deadlock detection and deadlock prevention

Issue	Cause	Solution
Non-Serializable Schedule	Improper interleaving	Two-Phase Locking (2PL)
Deadlock	Circular wait on locks	Detection or Prevention Techniques

2 Phase locking (2PL) : Two-Phase Locking ensures **conflict serializability** by dividing the transaction's locking behavior into two phases:
Growing Phase: Transaction **acquires locks** (shared or exclusive).
No lock can be released during this phase.
Shrinking Phase :
Transaction **releases locks**.
No new lock can be acquired once this phase begins.

Governing Rules of 2PL:

For a transaction to follow **Basic 2PL**, the following must hold:

1. **No conflicting locks** can be held by different transactions on the same data item.
2. **No unlock operation** can occur before all required locks are acquired.
3. **No data item is modified** until **all necessary locks** are successfully obtained.

Once a lock is released, no further locks can be requested — this defines the **lock point** of the transaction.

Basic 2PL Protocol

- Ensures **conflict serializability**.
- **Lock point** determines equivalent **serial order**.
- Does **not ensure recoverability or cascading abort freedom**.

Problems:

- **Irrecoverable Schedules**: Transactions might read uncommitted changes.
- **Deadlocks**: Due to circular waits.
- **Starvation**: Transaction may never acquire needed locks.

Strict 2PL Protocol

- Extends Basic 2PL by holding **all exclusive locks until commit/abort**.
- Ensures:
  - **Conflict serializability**
  - **Strict recoverability** (no cascading aborts)
- Commonly used in real-world DBMS (e.g., Oracle, SQL Server)

Problems:

- **Starvation is still** possible (solution: fairness policy).
- **Not deadlock-free**

NOTE:
Every schedule **allowed by 2PL** is always **conflict serializable**.
But **not every conflict serializable schedule** is allowed by 2PL.
Lock point of each transaction determines the serial order in 2PL.

Rigorous Two-Phase Locking
Rules:

- A transaction holds **all its locks (Shared or Exclusive)** until it **commits or aborts**.
- Lock acquisition follows standard 2PL (growing then shrinking phase).
- Ensures:
  - **Conflict serializability**
  - **Strict recoverability**
  - **Cascadeless schedules**

Still has:

- Deadlock possibility
- Starvation possibility (can be handled with wait-time policies)

Difference Strict and rigorous 2pl

	Strict 2PL	Rigorous 2PL
Holds exclusive locks	Till commit	Till commit
Holds shared locks	May release early	Till commit

```
View Serializable
 └── Conflict Serializable
     └── Strict
         └── Cascadeless
             └── Recoverable
                 └── Irrecoverable
```

All **conflict serializable** schedules are **view serializable**.
All **strict schedules** are **cascadeless** and **recoverable**.
Rigorous 2PL ensures the strictest class: **strict, conflict-serializable, and recoverable**

Rigorous 2PL (Two-Phase Locking) Protocol

In Rigorous 2PL, a stricter version of 2PL:

All locks (both Shared S and Exclusive X) are held until the transaction commits or aborts.
This guarantees strict schedules, which are both conflict-serializable and cascadeless.

Strict 2PL holds only exclusive locks until commit.
Rigorous 2PL ⊆ Strict 2PL ⊆ Basic 2PL (in terms of restrictiveness and guarantee of serializability).

Timestamp Ordering Protocols

For each data item X, the system maintains:
WTS(X): The largest timestamp of any transaction that successfully wrote X.
RTS(X): The largest timestamp of any transaction that successfully read X.

Let TS(T) be the timestamp of transaction T.

1. If T issues R(X) (read operation):

If TS(T) < WTS(X):
Reject the read; rollback T (as a newer transaction has already written X).

Else:
Allow the read.
Update RTS(X) = max(RTS(X), TS(T)).

2. If T issues W(X) (write operation):

If TS(T) < RTS(X):
Reject the write; rollback T (a younger transaction has already read X).
If TS(T) < WTS(X):

Reject the write; rollback T (a younger transaction has already written X).
Else:
Allow them to write.
Set WTS(X) = TS(T).

Thomas's Write Rule (TWR)

1. **If T issues R(X):**
   If WTS(X) > TS(T):
   rollback T.
   else
   execute
   Set RTS(X) = max{ TS(T), RTS(X)}

2. **If T issues W(X):**
   If TS(T) < RTS(X):
   Reject the write; rollback T.
   If TS(T) < WTS(X):
   Ignore the write (it is obsolete write).
   Else:
   Perform the write.
   Set WTS(X) = TS(T).

Strict Timestamp Ordering Protocol

A transaction Tj that issues a read or write (R(X) or W(X)) operation must wait until the transaction Ti that last wrote to X has committed or aborted, if TS(Tj) > WTS(X).

Deadlock Prevention Using Timestamp-Based Schemes

1. **Wait Die protocol (Preemptive)**
   Transactions are ordered by their timestamps.
   Smaller timestamp → Older transaction
   Larger timestamp → Younger transaction
   Transactions are ordered by their timestamps:
smaller timestamp = older transaction.
   If a transaction T_1 requests a lock held by T_2:
      If $TS(T_1) < TS(T_2)$ (i.e., T_1 is older):
         → T_1 waits.
      If $TS(T_1) > TS(T_2)$ (i.e., T_1 is younger):
         → T_1 is rolled back (dies) and
restarted with the same timestamp.
No deadlocks occur.
Starvation is possible for younger transactions, as they may be repeatedly rolled back if older transactions continuously block them.

2. **Wound Wait Protocol (Non- Preemptive)**

   Transactions are ordered by their **timestamps**.
   Older transactions have higher priority over
younger ones.
   Transactions are again ordered by **timestamp**.
   If a transaction T_1 requests a lock held by T_2:
      If $TS(T_1) < TS(T_2)$ (i.e., T_1 is older):
         → **T_1 preempts T_2** — T_2 is
   rolled back (wounded), and T_1 gets the lock.
      If $TS(T_1) > TS(T_2)$ (i.e., T_1 is younger):
         → **T_1 waits**.

No deadlocks occur.
Starvation is possible for younger transactions, as they may be forced to wait if older transactions repeatedly preempt them.