SQL/Relational/B/B+Tree

SQL (Structured Query Language) is a language used to work with databases.
SQL is used to store, read, update, and delete data in a database.

Think of a school record system:

- Student names
- Roll numbers
- Marks

All this data is stored in a database.
To:

- get student details
- add new students
- update marks
- delete records

we use SQL.

What can SQL do?

- Insert data
- Read data
- Update data
- Delete data
- Create tables and databases

Where is SQL used?

- Websites (login, signup systems)
- Mobile apps
- Backend development
- Banking systems
- Big companies (Google, Amazon, Netflix)

Popular SQL databases

- MySQL
- PostgreSQL
- Oracle
- SQL Server
- SQLite

Master–Slave Architecture in SQL (also called Primary–Replica architecture) is a database setup where one database handles writes and others copy and read the data.

- Master (Primary) ➝
  Handles WRITE operations (INSERT, UPDATE, DELETE)
- Slave (Replica) ➝
  Handles READ operations (SELECT)

Slaves continuously copy data from the master.

Why do we need Master–Slave?

Main problems it solves

1. High traffic (many users)
2. Slow reads
3. Backup & safety
4. Scalability

How it works (step by step)

1. Client sends a WRITE query
   ➝ goes to Master
2. Master updates its database
3. Master writes changes to a log (binary log / WAL)
4. Slave reads this log
5. Slave replicates the same changes
6. Client sends READ queries
   ➝ go to Slaves

```
    WRITE
Client ────────▶ Master
         |
         |  Replication
         ▼
     Slave 1
     Slave 2
     Slave 3

    READ ◀────── Client
```

Example

Write (only on Master)

```
INSERT INTO users VALUES (1, 'Vijay');
```

Read (from Slave)

```
SELECT * FROM users;
```

Replication types

1. Asynchronous
   - Master doesn't wait for slaves
   - Fast
   - Small risk of data lag
2. Synchronous
   - Master waits for slave confirmation
   - Strong consistency
   - Slower
3. Semi-synchronous
   - Balance of both

Advantages

- Faster reads
- Better performance
- Easy scaling
- Backup from slaves
- Reduced load on master

Disadvantages

- Replication lag
- Master is a single point of failure
- More complex setup
- Possible stale reads

Where it is used?

- MySQL Replication
- PostgreSQL Replicas
- Large-scale apps (e-commerce, social media)
- Banking read-heavy systems

Data Block / Data Page

Smallest unit of data storage in a database.

- Database does NOT read one row at a time
- It reads blocks/pages
- Size is fixed (example: 8 KB in many DBs)

Think of it like:

A page of a notebook that stores many records.

Disk

└──── Data Page (8KB)

├──── Header

├──── Records

└──── Offset Table

2.Data Page Header

Metadata about the page (info about the page itself)

Header stores:

- Page ID
- Table ID
- Page type (data / index)
- Free space available
- Number of records
- Checksum (for corruption check)

Header helps DB answer:

"What kind of page is this and what's inside it?"

| Header | Records | Offsets |

3.Data Records (Rows)

Actual table rows are stored here.

Each record contains:

- Column values (id, name, age, etc.)
- Row metadata (row id, transaction id)
- NULL bitmap
- Version info (for MVCC)

Example record:

(1, 'Vijay', 24)

(2, 'Aashu', 26)

Records are usually variable length.

4.Offset Table (Slot Directory)

A list of pointers that tell where each record starts.

Why needed?

- Records move when rows are updated
- Offsets keep record order intact
- Faster row access

Offset table is usually stored at the end of the page.

Example:

```
Offset Table
-------------
Slot 0 → byte 120
Slot 1 → byte 180
Slot 2 → byte 260
```

So DB knows:

"Row 2 starts at byte 260"

5.How a Data Page is structured (visual)

```
+----------------------+
| Page Header          |
+----------------------+
| Record 1             |
| Record 2             |
| Record 3             |
| ...                  |
| Free Space           |
+----------------------+
```

| Offset Table (Slots) |

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

6.Why this design is important

- Fast disk I/O
- Efficient memory usage
- Easy record movement
- Supports transactions & rollback
- Index pointers work at page level

Database	Block/Page Size
MySQL (InnoDB)	16 KB
PostgreSQL	8 KB
Oracle	8 KB / 16 KB
SQL Server	8 KB

A data page is a container
Header = page info
Records = actual rows
Offsets = pointers to rows

Real-World Example: School Register Book

Imagine a school register book

1.Data Block / Data Page = One Page of the Register

- The whole register = Database
- One page of the register = Data Page / Data Block

Teacher never opens one student at a time
She opens one full page → same as DB reading a data page.

2.Page Header = Page Information at the Top

At the top of the page, teacher writes:

- Class: 10th A
- Date: 3 Feb

- Total students on this page: 30
- Page number: 12

This is the Page Header

It doesn't store students, only info about the page.

3.Data Records = Student Entries

Below the header, you have:

Roll Name Marks

1 Vijay 90

2 Aashu 85

3 Sanya 88

Each student row = Data Record

This is the actual data we care about.

4.Offset Table = Index at the Bottom of the Page

At the bottom of the page, teacher writes:

Roll → Line Number

1 → Line 5

2 → Line 6

3 → Line 7

This is the Offset Table

It tells:

"Student with roll 2 starts at line 6"

5.Why Offset Table is needed?

Suppose:

- Student "Aashu" name is updated to "Aashu Sharma"
- His row becomes longer

Teacher just:

- Shifts lines
- Updates the line number at the bottom

No need to rewrite the whole page
Same idea in databases

Database Term	Real World Example
Database	Register Book
Data Page / Block	One Page
Page Header	Page info at top
Data Record	One student row
Offset Table	Line numbers at bottom

Page = Notebook page
Header = Page details
Records = Written lines
Offsets = Line numbers

What is Indexing?

Indexing is a data structure that helps the database find data faster.

Without an index:

- Database scans every row
  With an index:
- Database jumps directly to the data

Real-World Example

Book without Index

You want to find “Binary Search” in a 500-page book.

What do you do?

- Start from page 1
- Read every page

Very slow.

Book with Index

At the back of the book:

Binary Search → Page 213

You:

- **Open page 213 directly**

Very fast.

This back-of-book index = Database index

What problem does indexing solve?

1.Full Table Scan Problem

Without index:

SELECT * FROM users WHERE email = 'a@gmail.com';

DB checks:

- **Row 1**
- **Row 2**
- **Row 3**
- **...**
- **Row 1,000,000**

With index:

- **Direct jump to the matching row**

2.Performance Problem (Speed)

- **Search time drops from seconds → milliseconds**
- **Very important for:**
  - **Login systems**
  - **Payments**
  - **Search features**

3.Scalability Problem

- **Small data → scan is ok**
- **Big data (millions of rows) → scan is impossible**

Index makes large data usable.

How indexing works internally (high level)

- Index stores:
  - Column value
  - Pointer to data page / row

Example:

email_index

----

a@gmail.com → Page 12, Slot 3

b@gmail.com → Page 45, Slot 1

DB:

1. Search index (fast)
2. Jump to exact page
3. Read the row

Types of Indexes

- B-Tree (most common)
- Hash index (exact match)
- Composite index (multiple columns)
- Unique index
- Clustered index
- Non-clustered index

But indexing has a cost

Indexes are not free.

Problems indexing creates:

1. Extra storage
2. Slower INSERT / UPDATE / DELETE
   (index must also be updated)
3. Too many indexes = bad performance

Over-indexing is dangerous

When should you use indexing?

Index columns that are:

- Used in WHERE

- Used in JOIN
- Used in ORDER BY
- Used in GROUP BY
- Frequently searched

Don't index:

- Very small tables
- Columns with same values (low cardinality)

Why do we even need B / B+ Trees?

Databases:

- Store millions / billions of rows
- Data is on disk (slow)
- Need fast search, insert, delete

Binary Search Tree problem:

- Too tall
- Too many disk reads

Solution: B / B+ Trees

- Short height
- Very few disk I/Os
- Perfect for databases

1.What is a B-Tree?

A B-Tree is a self-balancing multi-way search tree where:

- Each node can have many keys
- All leaf nodes are at same level
- Tree stays short and wide

Optimized for disk-based storage

B-Tree Properties

Let order = m

- Each node can have:
  - Max m children
  - Max m−1 keys
- Minimum keys = ceil(m/2) − 1
- Always balanced

Example node:

| 10 | 20 | 30 |

Children:

<10   10-20   20-30   >30

2.B-Tree Structure (Visual)

```
        [20 | 40]

       /   |    \

    [5 | 10] [25 | 30] [45 | 50]
```

- **Keys are sorted**
- **Internal nodes store keys + pointers**
- **Leaf nodes store actual data pointers**

3.How search works in B-Tree

Search for 30:

1. **Start at root [20 | 40]**
2. **30 lies between 20 & 40 → go middle child**
3. **Found in leaf**

Time complexity:
$O(\log_m N)$ (very small)

4.Insert in B-Tree

Case 1: Node has space

- **Insert key in sorted order**

Case 2: Node is full

- **Split the node**
- **Middle key goes up**
- **Tree height increases only when root splits**

This keeps tree balanced

5.Delete in B-Tree

Deletion may cause:

- **Borrow key from sibling**
- **Or Merge nodes**

Tree still stays balanced

6.Problems with B-Tree in Databases

Internal nodes store:

- **Keys**
- **Pointers**
- **Data pointers**

This reduces:

- **Number of keys per node**
- **Disk efficiency**

Solution → B+ Tree

What is a B+ Tree?

A B+ Tree is an improved version of B-Tree where:

Internal nodes:

- **Store ONLY keys**
- **No data**

Leaf nodes:

- **Store keys + actual data pointers**
- **Linked like a linked list**

1.B+ Tree Structure

```
          [20 | 40]

        /   |    \

     [5 | 10] [25 | 30] [45 | 50]

       ↓      ↓       ↓

  (Leaf) (Leaf) (Leaf) (Leaf)
```

↔ ↔ ↔ ↔ (Linked)

All actual data is only in leaf nodes

2.Why B+ Tree is better than B-Tree

1.Faster range queries

SELECT * FROM users WHERE age BETWEEN 20 AND 30;

Because:

- Leaf nodes are linked
- Sequential scan is fast

2.Better disk utilization

- Internal nodes are smaller
- More keys per node
- Tree height is even smaller

3.Consistent search path

- Every search goes to leaf
- Predictable performance

Feature	B-Tree	B+ Tree
Data stored	Internal + Leaf	Leaf only
Internal nodes	Keys + data	Keys only
Leaf nodes linked	No	Yes
Range queries	Slower	Faster
Used in DBs	Rare	Yes

Most databases use B+ Tree

Database	Index Type
MySQL (InnoDB)	B+ Tree
PostgreSQL	B+ Tree
Oracle	B+ Tree
SQL Server	B+ Tree

Library system

- Floors = Tree levels
- Cupboards = Nodes
- Index cards = Keys
- Books = Actual data
- All books stored only on last floor
- Last floor corridors are connected (linked list)

That's B+ Tree

B-Tree: Balanced tree for disk
B+ Tree: Optimized B-Tree used in databases
B+ Tree = faster search + faster range queries

Clustered Index

What is a Clustered Index?

A clustered index decides the physical order of data in the table.

- Table data is stored in index order
- Actual rows are inside the leaf nodes
- Only ONE clustered index per table

Think:

"Table itself is the index"

Real-World Example

Students sorted by Roll Number

Roll | Name

----

1 | Vijay

2 | Sanya

3 | Aman

Data is physically stored in roll-number order.

This is Clustered Index on Roll

Internal view (B+ Tree)

Root

↓

Internal Nodes

↓

Leaf Nodes → ACTUAL DATA ROWS

Leaf node contains:

[Roll=1 | Vijay]

[Roll=2 | Rahul]

[Roll=3 | Aman]

Example SQL

CREATE CLUSTERED INDEX idx_roll ON students(roll);

(In MySQL InnoDB, PRIMARY KEY is clustered by default)

Advantages

- Very fast range queries
- No extra lookup needed
- Efficient disk reads

Disadvantages

- Only one allowed
- Inserts can cause page splits
- Changing clustered key is expensive

Non-Clustered Index

What is a Non-Clustered Index?

A separate structure that stores:

- **Indexed column**
- **Pointer to actual row**

Data order does NOT change

Real-World Example

Phone directory:

Name → Phone Number

But actual people live anywhere.

Directory = Non-Clustered Index
Houses = Actual data

Internal view

Non-Clustered Index (B+ Tree)

Leaf Node:

[Name=Vijay → Pointer → Data Page]

Then:

1. **Search index**
2. **Jump to data page**
3. **Fetch row**

Example SQL

CREATE NONCLUSTERED INDEX idx_name ON students(name);

Advantages

- **Multiple allowed**
- **Faster search on many columns**
- **Flexible**

Disadvantages

- Extra storage
- Extra lookup (key lookup / bookmark lookup)
- Slightly slower than clustered for reads

Feature	Clustered Index	Non-Clustered Index
Physical order	Yes	No
Data in leaf	Yes	No
Pointer needed	no	yes
Number allowed	1	Many
Range query	Very fast	Slower
Storage	Less	More

In InnoDB (MySQL):

- Primary Key = Clustered Index
- Non-clustered index leaf stores:
  - Index key
  - Primary key value
- Then DB uses PK to find row

When to use what?

Use Clustered Index when:

- Column is frequently searched
- Range queries (BETWEEN, ORDER BY)
- Primary key

Use Non-Clustered Index when:

- Multiple search columns
- WHERE + JOIN conditions
- Secondary lookups

Clustered = Data sorted physically
Non-clustered = Separate lookup table