TASK-4 (next)

Database using SQL

1. Database:

- A database is a collection of data.

- It is stored in such a way that we can store, mandge, and daccess data edsily.

2. SQL (Structured Query Langudge):

SQL is a language used to interdct with databases.

Using SQL, we can Insert, Update, Delete, and Retrieve data.

3. Main SQL Commands:

0

CREATE DATABASE / TABLE → Create a new database or table.

- INSERT INTO → Add data.

0

SELECT → Retrieve/view data.

- UPDATE → Modify existing data.

- DELETE Remove

Keys in Database:

Primdry Key → Unique identifier of a record.

· Foreign Key → Used to link two tables.

5. Jvins:

· Used to combine data from multiple tables.

· Types: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN.

6. Functions in SQL:

O

Aggregate Functions: SUM(), AVG(), COUNTO), MINO), MAXO).

String Functions: CONCAT(), UPPER(), LOWERO.

Date Functions: NOW(), DATE_ADD(), DATEDIFFO.

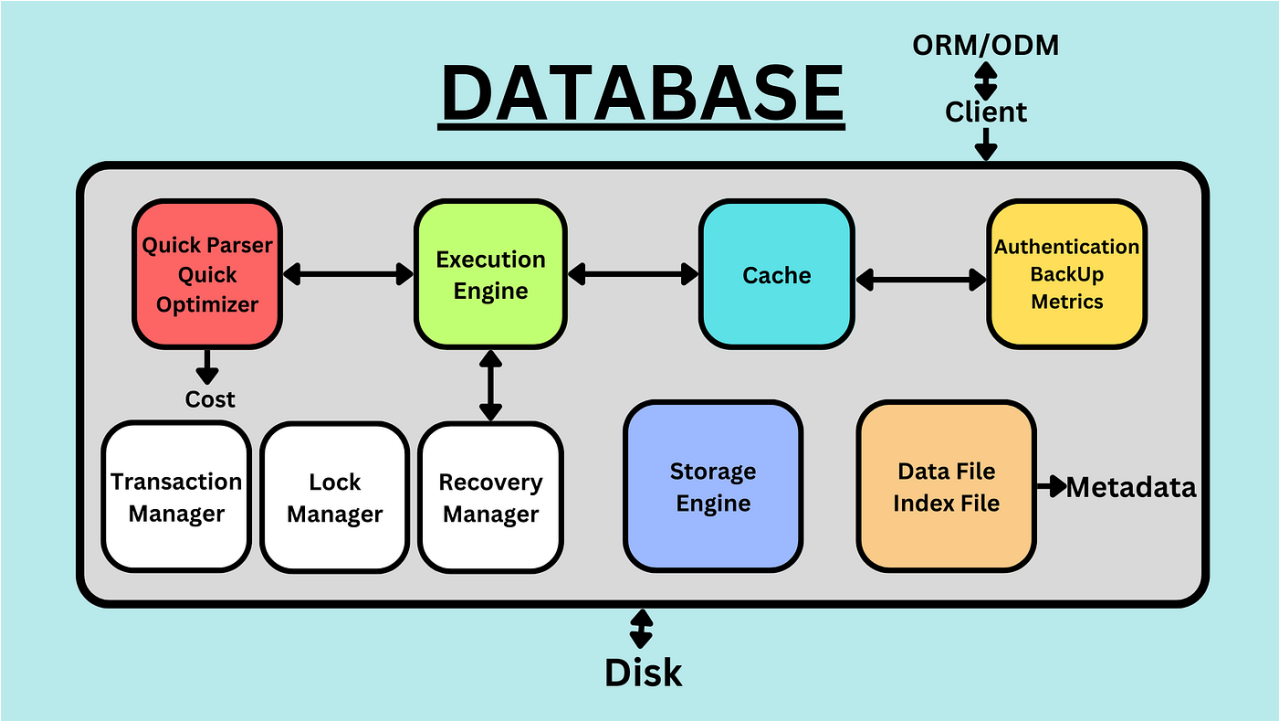
7. Views:

Save a query and use it like a virtual table.

8. Indexes:

O

Used to speed up uata sedrchi



Database — deeper view

What it is: a structured store for persistent data (tables, indexes, constraints, transactions).

Types: relational (RDBMS: MySQL, PostgreSQL, SQLite), columnar (analytics), NewSQL/distributed, and NoSQL. Use RDBMS when relationships, transactions and strong consistency matter.

Modeling: start with an ER diagram (entities, relationships, attributes). Capture cardinality (1:1, 1:N, N:M) and optionality. Good modeling prevents slow queries later.

SQL (role & categories)

DDL (Data Definition Language):
CREATE
,
ALTER
,

DROP
— schema design, migration.
DML (Data Manipulation Language):
INSERT
,
UPDATE
,
DELETE
,
SELECT
— day-to-day data operations.
TCL (Transaction Control):
BEGIN/START
,
COMMIT
,
ROLLBACK
— atomic changes.
DCL (Data Control):
GRANT
,
REVOKE
— access control.
Best practice: Keep DDL changes in version control (migration scripts).

Main SQL commands — practical examples & notes

CREATE TABLE (define types, constraints)

```
CREATE TABLE customers (
  customer_id SERIAL PRIMARY
KEY,

  name VARCHAR(200) NOT NULL,
  email VARCHAR(255) UNIQUE,
```

```
  created_at TIMESTAMP DEFAULT
NOW()
);
```

INSERT

```
INSERT INTO customers (name,
email) VALUES ('Asha',
'asha@example.com');
```

SELECT

```
SELECT customer_id, name FROM
customers WHERE email LIKE
'%@example.com' ORDER BY
created_at DESC LIMIT 10;
```

UPDATE / DELETE

```
UPDATE customers SET name =
'A. Gupta' WHERE customer_id =
5;
DELETE FROM customers WHERE
created_at < '2020-01-01';
```

DROP: use with caution — usually via migration scripts with backups.

Keys & constraints

Primary key: unique row id — prefer narrow (small integer) for performance.
SERIAL
/
BIGSERIAL
or
UUID
.
Foreign key: enforces referential integrity; can cascade
ON DELETE/UPDATE
or restrict.
Unique, NOT NULL, CHECK: ensure data consistency at DB-level (defensive programming).
Tradeoff: Constraints add CPU overhead on writes but prevent data corruption and simplify app logic.

Joins & subqueries — semantics & performance

INNER JOIN: returns rows matching both tables.
LEFT JOIN: keep all rows from left, matching or NULL from right.
RIGHT JOIN / FULL JOIN: use when appropriate; some RDBMS (MySQL) lack FULL JOIN.

Subqueries: correlated (run per outer row) vs non-correlated. Correlated can be slow; often rewrite as JOIN or use EXISTS
.
Example:

```
-- INNER JOIN
SELECT o.id, c.name, o.total
FROM orders o

JOIN customers c ON
o.customer_id = c.customer_id
WHERE o.total > 100;

-- Aggregation + HAVING
SELECT c.customer_id, COUNT(*)
AS orders, SUM(o.total) AS
total_spent
FROM customers c

JOIN orders o ON o.customer_id
= c.customer_id
GROUP BY c.customer_id
HAVING SUM(o.total) > 1000;
```

Performance tip: ensure join columns are indexed; avoid functions on join columns.

Functions & aggregates — use and optimization

Aggregates:
SUM, AVG, COUNT, MIN, MAX
. Use
GROUP BY
.
Window functions:
ROW_NUMBER(), RANK(), SUM()
OVER(PARTITION BY ...)
— powerful for running totals, top-N per group, without collapsing rows.

```
SELECT customer_id, order_id,
total,
       SUM(total) OVER
(PARTITION BY customer_id
ORDER BY order_date) AS
running_total
FROM orders;
```

Tip: grouping many rows is expensive—pre-aggregate if possible, use materialized views for frequently used aggregates.

Views & materialized views

Views: saved queries acting like virtual tables. Good for encapsulating complexity and implementing security (restrict columns).

Materialized views: store results physically — faster reads, but must be refreshed (manually or scheduled). Ideal for expensive aggregations.
Example:

```
CREATE VIEW active_customers AS

SELECT customer_id, name FROM
customers WHERE active = TRUE;


-- Materialized view (Postgres)
CREATE MATERIALIZED VIEW
monthly_sales AS
SELECT date_trunc('month',
order_date) AS month,
SUM(total) AS total
FROM orders GROUP BY 1;
```

Tradeoff: materialized views speed reads but add storage & refresh complexity.

Indexes — types & strategy

Common types: B-tree (default), Hash, GiST/GIN (Postgres: for full-text, arrays), BRIN (for very large ordered data).
Create index

```
CREATE INDEX
idx_orders_customer ON
orders(customer_id);
```

Composite index: order matters.
CREATE INDEX idx ON table(a,b);
helps queries filtering
a
or
a AND b
, but not only
b
.

Covering index: include columns so the query can be satisfied from the index alone (
INCLUDE
in Postgres/MSSQL).
Partial/index on expressions: index only a subset (
WHERE

clause) or expression (e.g., lower(email)).
Pitfalls: indexes speed reads, slow down writes. Too many indexes increases insert/update cost. Monitor index usage with
EXPLAIN
/statistics.

Selectivity matters: index columns with high distinct values (high cardinality).

Transactions & isolation

ACID: Atomicity, Consistency, Isolation, Durability.
Isolation levels: (from weak to strict)
READ UNCOMMITTED
,
READ COMMITTED
,
REPEATABLE READ
,
SERIALIZABLE

. Higher isolation reduces anomalies but increases locking/latency.
Example (Postgres style):

```
BEGIN;

UPDATE accounts SET balance =
balance - 100 WHERE id = 1;

UPDATE accounts SET balance =
balance + 100 WHERE id = 2;
COMMIT;
```

Concurrency & locking: keep transactions short, avoid user input inside a transaction, use optimistic concurrency when possible (version columns), watch for deadlocks and use retry logic.

Schema design — normalization vs denormalization

Normalization: 1NF/2NF/3NF reduces redundancy and ensures integrity. Use it for OLTP workloads.

Denormalization: stores derived/duplicate data to speed reads (useful for read-heavy analytics). Adds complexity (need to keep duplicates in sync).

Partitioning & sharding: partition large tables by range/list/hash to speed queries and maintenance. Shard when dataset and throughput exceed single node capacity.

Query optimization checklist

1. Run `EXPLAIN` / `EXPLAIN ANALYZE` to see actual plan and cost. (`EXPLAIN ANALYZE` runs the query in Postgres).
2. Avoid `SELECT *` — choose only needed columns.

3 Use proper indexes (WHERE, JOIN, ORDER BY, GROUP BY columns).

4 Avoid functions on indexed columns (e.g., WHERE LOWER(name) = 'x' prevents index use unless you index the expression).

5 Use LIMIT + sensible pagination (cursor-based for large offsets).

6 Batch inserts/updates (bulk operations) rather than many single-row operations.

7 Cache heavy, frequently-read results (Redis, materialized views).

8 Maintain statistics (ANALYZE , VACUUM in Postgres).

9 Monitor slow query logs and optimize top offenders.

Security & best practices

Least privilege: grant only required permissions to DB roles.
Use parameterized queries / prepared statements to prevent SQL injection.
Example (pseudo):

```
-- application code
cursor.execute("SELECT * FROM users WHERE id = %s", (user_id,))
```

Encryption: encrypt connections (TLS), and sensitive data at-rest when required.

Audit & logging: enable logs for DDL, suspicious access, and monitor them.

Passwords & secrets: use secret managers, rotate credentials, use separate accounts for apps vs admins.

Scalability & high availability

Read replicas: scale reads by sending queries to replicas. Ensure replication lag is acceptable.
Master-master vs master-slave: choose based on write patterns. Master-master increases complexity for conflict resolution.

Sharding: split data horizontally when single node capacity is exceeded — application-level routing complexity.
Caching: use an in-memory cache (Redis / Memcached) to reduce DB load for frequent reads.
Connection pooling: essential for web apps (PgBouncer for Postgres, ProxySQL for MySQL).

Maintenance & monitoring

Backups: full & incremental, test restores regularly.

Maintenance tasks: vacuum/optimize, rebuild indexes, refresh materialized views during off-peak hours.

Monitoring metrics: QPS, latency, connection count, cache hit ratio, slow queries, index usage, replication lag. Use tools (Prometheus, Grafana, built-in RDBMS metrics).

Tools & workflows

Migrations: use tools (Flyway, Liquibase, Alembic) to manage schema changes.

CI/CD: run migrations in CI, test against staging DB.

ORMs: increase developer productivity but be aware of generated SQL; profile queries and use raw SQL for performance-sensitive paths.

Quick practical SQL examples (collected)

```
-- Create tables with PK + FK
CREATE TABLE products (
  product_id SERIAL PRIMARY
KEY,
  name TEXT NOT NULL,

  price NUMERIC(12,2) NOT NULL
);
CREATE TABLE orders (

  order_id SERIAL PRIMARY KEY,
  customer_id INTEGER
REFERENCES
customers(customer_id),
  order_date TIMESTAMP DEFAULT
NOW(),
  total NUMERIC(12,2)
);

-- Join + aggregation

SELECT c.name,
COUNT(o.order_id) AS orders,
SUM(o.total) AS spent
FROM customers c

LEFT JOIN orders o ON
o.customer_id = c.customer_id
GROUP BY c.name
ORDER BY spent DESC;

-- Index for faster lookup
```

```
CREATE INDEX
idx_customers_email ON
customers (lower(email));

-- Transaction
BEGIN;

UPDATE accounts SET balance =
balance - 100 WHERE id = 1;

UPDATE accounts SET balance =
balance + 100 WHERE id = 2;
COMMIT;

-- EXPLAIN (Postgres)

EXPLAIN ANALYZE SELECT * FROM
orders WHERE customer_id = 123;
```

Practical checklist before you ship

Schema reviewed with constraints and indexes in place.
Critical queries run with EXPLAIN ANALYZE ; slow queries optimized.

Backups configured and restore tested.
Monitoring and alerts for slow queries/replication lag.
Least-privilege roles for applications.
Transactional boundaries and retry logic implemented in app.
If you want, I can:

analyze a specific schema or slow query you have (paste table schemas + query), or

produce optimized SQL examples for a sample ecommerce schema (customers, products, orders, order_items), or

A database is a collection of data.

It is stored in such a way that we can **store, manage, and access** data easily.

SQL is a language used to interact with databases.
Using SQL, we can **Insert, Update, Delete, and Retrieve** data.

CREATE DATABASE / TABLE → Create a new database or table.
INSERT INTO → Add data.
SELECT → Retrieve/view data.
UPDATE → Modify existing data.
DELETE → Remove data.
DROP → Delete a table or database completely.

Primary Key → Unique identifier of a record.

Foreign Key → Used to link two tables.

Used to combine data from multiple tables.
Types: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN.

Aggregate Functions: SUM(), AVG(), COUNT(), MIN(), MAX().
String Functions: CONCAT(), UPPER(), LOWER().
Date Functions: NOW(), DATE_ADD(), DATEDIFF().

Save a query and use it like a virtual table.

Used to speed up data searching.

To ensure safe and controlled changes in a database.
Commands: COMMIT, ROLLBACK, SAVEPOINT.