File System to RDBMS

File System

A file system stores data in individual files managed by the operating system.

Data is typically organized into directories without any inherent relationships between files.

Characteristics

- Data is stored in plain files (text, CSV, binary, etc.).
- Each file is independent; no centralized control or structure.
- Security is mostly limited to OS-level permissions.
- Data retrieval requires custom programs or scripts.
- High chances of data redundancy, inconsistency, and duplication.

Relational Database Management System (RDBMS)

An RDBMS organizes data into **tables** consisting of rows and columns. Relationships between tables are maintained using primary keys and foreign keys.

Characteristics

- Structured storage with predefined schemas.
- SQL is used for querying and data manipulation.
- Supports constraints such as primary key, foreign key, unique, and check.
- Provides ACID (Atomicity, Consistency, Isolation, Durability) properties for transactions.
- Advanced security and role-based permissions.

Key Differences

Feature	**File System**	**RDBMS**
Data Redundancy	High	Reduced through normalization
Data Integrity	Hard to maintain	Enforced via constraints
Data Access	Custom programs needed	SQL makes it simple and standardized
Security	Basic OS-only	Strong user-level control & encryption
Backup/Recovery	Mostly manual	Built-in automated mechanisms
Relationships	Not supported	Full relationship management

Limitations of File System

- No standard query language.
- Poor data integrity and consistency.
- Does not support concurrency well.
- Not suitable for large-scale or multi-user applications.
- Difficult to manage and maintain over time.

Advantages of RDBMS

- Ensures data consistency and integrity.
- Reduces redundancy using normalization.
- Supports multiple users with proper concurrency control.
- Provides strong security, backup, and recovery features.
- Easily scalable for large data volumes.
- Offers powerful query capabilities using SQL.

Why Shift from File System to RDBMS?

Organizations move to RDBMS because it provides:

- Centralized, structured data management
- Secure, reliable, and scalable storage
- Faster querying and reporting
- Reduced redundancy and improved integrity
- Support for multi-user environments

What is a Data Warehouse?

A Data Warehouse (DW) is a centralized repository that stores large volumes of historical data collected from multiple sources. It is designed for querying, analysis, and reporting rather than day-to-day transaction processing.

Characteristics

- **Subject-oriented**: Organized around business themes (sales, finance, HR).
- **Integrated**: Combines data from multiple heterogeneous sources.
- **Time-variant**: Stores historical data for trend analysis.
- **Non-volatile**: Data remains stable and not frequently modified.

Purpose

- Supports decision-making.
- Enables trend, pattern, and multi-dimensional analysis.
- Provides a "single source of truth."

Data Warehouse Architecture

a. Source Systems

Operational databases, CRM, ERP, logs, flat files.

b. ETL / ELT Process

- **Extract** data from sources.
- **Transform** (clean, validate, aggregate).
- **Load** into the warehouse.

c. Storage Layers

- **Staging Area** – temporary storage for raw data.
- **Data Warehouse** – central historical store.
- **Data Marts** – department-specific subsets (sales mart, finance mart).

d. Presentation/Access Layer

Reporting tools, dashboards, OLAP, ad-hoc queries.

OLAP (Online Analytical Processing)

- Enables multi-dimensional analysis using **cubes** (dimensions & measures).
- Supports operations like **slice**, **dice**, **drill-down**, **roll-up**, and **pivot**.

Business Intelligence (BI)

Business Intelligence refers to technologies, tools, and processes used to transform raw data into actionable insights for decision-making.

Components of BI

- Reporting & Dashboards
- Data Mining & Predictive Analytics
- OLAP Tools
- Visualization Tools (Power BI, Tableau, Looker)
- Self-service BI

BI Goals

- Improve decision quality.
- Identify trends and patterns.
- Monitor business performance (KPIs).
- Enable real-time insights.

Data Warehouse vs BI

Aspect	Data Warehouse	Business Intelligence
Purpose	Stores integrated historical data	Converts data into insights
Focus	Data storage & organization	Reporting, analysis, prediction
Users	Data engineers, DBAs	Managers, analysts, executives
Tools	ETL tools, DBMS, OLAP	Visualization, dashboards, analytics

Benefits of Data Warehouse + BI

- Better strategic and operational decision-making.
- Centralized and consistent data for analysis.
- Reduced reporting time.
- Increased productivity and performance monitoring.
- Ability to forecast future trends using analytics.

Real-World Examples

- Sales forecasting and demand prediction.
- Customer segmentation and churn analysis.
- Financial reporting and risk analysis.
- Supply chain optimization.

OLTP – Online Transaction Processing

OLTP (Online Transaction Processing) refers to systems designed to manage **real-time transactional data**. These systems handle a large number of short, atomic operations such as insert, update, and delete.
Typical use cases include:

- Banking transactions
- Retail POS systems
- Ticket booking systems
- E-commerce order processing

Key Characteristics of OLTP

- **High Transaction Volume**: Processes thousands to millions of transactions per second.
- **Short and Simple Queries**: Focus on quick reads/writes, not complex analysis.
- **Real-Time Processing**: Immediate response is required.
- **Data Integrity**: Ensured via **ACID properties**.
- **Normalized Schema**: Reduces redundancy and improves update efficiency.
- **Concurrent Access**: Supports multiple users accessing the database simultaneously.

OLTP heavily relies on ACID rules to ensure correctness:

- **Atomicity** – Each transaction is all-or-nothing.
- **Consistency** – Data must always remain valid after any transaction.
- **Isolation** – Concurrent transactions do not affect each other.
- **Durability** – Once committed, data persists permanently.

OLTP Architecture Components

1. **User/Application Layer** – Web/apps initiating transactions.
2. **Transaction Manager** – Ensures ACID compliance, locking, and isolation.
3. **Database Engine** – Executes SQL operations and maintains indices.
4. **Storage** – Optimized for fast reads/writes.

OLTP	OLAP
Run day-to-day operations	Analytical querying & reporting
Current/real-time	Historical/aggregated
Short & simple	Long & complex
Highly normalized	Denormalized (star/snowflake)
Very high volume	Relatively low
Clerical staff, applications	Analysts, managers

Columnar Databases

A columnar database (or column-oriented database) stores data column by column instead of row by row. This design is optimized for analytics, aggregation, scanning large datasets, and OLAP workloads.
Examples: Amazon Redshift, Apache Parquet, Snowflake, BigQuery, Apache Cassandra (hybrid).

How it works:
In a row-based DB (e.g., OLTP systems), data is stored like:

Row 1: [Name, Age, City]
Row 2: [Name, Age, City]

In a columnar DB:

Column Name: [Amit, John, Sara]
Column Age: [25, 32, 29]
Column City: [Delhi, NYC, London]

This reduces I/O and speeds up analytical queries.

Key Characteristics

- Column-wise storage
- Highly compressed data due to similar column values
- Optimized for read-heavy workloads
- Vectorized execution (process many values at once)
- Efficient for aggregations (SUM, AVG, COUNT)
- Supports distributed and large-scale architectures

Why Columnar Databases are fast

1. Reads only relevant columns, not entire rows → less disk I/O
2. Better compression (run-length, dictionary) → fits more data in memory
3. Vectorized processing → CPU processes batches of data
4. Optimized for scans → common in BI and data warehouse queries

Row-Oriented (OLTP)	Column-Oriented (OLAP)
By rows	By columns
Transactions	Analytics
Insert/Update/Delete	Scan, Aggregate, Filter
Low	Very high
Fast for small read/write ops	Fast for large analytical queries
MySQL, PostgreSQL, Oracle	Redshift, Snowflake, BigQuery

ACID and BASE

ACID is a set of properties that ensure reliable and consistent transaction processing in traditional relational databases (OLTP systems).

Atomicity

- A transaction is all-or-nothing.
- If any step fails, the entire transaction rolls back.

Consistency

- Ensures the database moves from one valid state to another.
- All rules, constraints, and validations must be satisfied.

Isolation

- Multiple transactions occurring simultaneously do not interfere with each other.
- Achieved using locks or MVCC.

Durability

- Once a transaction is committed, the data is permanently saved.
- Survives crashes, power failures, and errors.

Some Application areas:

- Banking systems
- Payment gateways
- E-commerce order transactions
- Inventory updates
- Reservation systems

BASE is a model used by NoSQL and distributed systems that prioritize scalability and availability over strict consistency.

BASE stands for:

Basically Available

- The system guarantees availability even during failures or partitions.
- Uses replication and distributed nodes.

Soft State

- The system's state may change over time even without new input.
- Data is not instantly consistent across systems.

Eventual Consistency

- Data will eventually become consistent across nodes.
- Temporary inconsistencies are acceptable.

Some Application areas:

- Social media feeds
- Messaging systems
- Product catalog services
- IoT platforms
- Real-time analytics and big data systems

ACID	**BASE**
Strong	Eventual
Reliability, correctness	Availability, scalability
Transactions (OLTP)	Distributed/Big Data systems
Higher	Lower
Stable	Soft state
MySQL, PostgreSQL, Oracle	Cassandra, DynamoDB

What is a Data Lake?

A Data Lake is a centralized storage system that allows organizations to store all types of data (structured, semi-structured, and unstructured) at any scale.

Characteristics

- Stores raw data in its native format ("schema-on-read").
- Supports diverse data formats: JSON, CSV, images, logs, videos.
- Low-cost and highly scalable storage (e.g., AWS S3, Azure Data Lake).
- Flexible for advanced analytics, ML, and big data processing.
- Lacks strong governance and performance optimizations.

Limitations

- Data becomes a "data swamp" without proper governance.
- Slow querying due to lack of transactional guarantees.
- Poor data quality management.
- Complex to integrate for BI/analytics use cases.

What is a Lakehouse?

A Lakehouse is a modern data architecture that combines the low-cost storage and flexibility of a Data Lake with the data management features of a Data Warehouse.

Key Characteristics

- Provides ACID transactions on data lake storage.
- Unifies batch and streaming analytics.
- Supports schema enforcement + governance.
- Optimized for BI, SQL analytics, and machine learning.
- Uses open formats like Parquet, Delta Lake, Iceberg, Hudi.

Why Move from Data Lake to Lakehouse?

Issue in Data Lake	Improvement in Lakehouse
No ACID transactions	ACID support ensures reliable updates
Poor data quality	Schema enforcement & governance
Slow SQL performance	Optimized query engines & caching
Data duplication	Unified single storage layer
Hard for BI tools	SQL-friendly & warehouse-like performance

Key Components of a Lakehouse

a. Storage Layer

- Object storage with open data formats (Parquet, ORC).

b. Metadata / Transaction Layer

- Delta Lake, Apache Iceberg, or Apache Hudi provide:
  - ✓ ACID transactions
  - ✓ Versioning & time travel
  - ✓ Schema evolution/enforcement

c. Compute Layer

- Engines such as Spark, Trino, Presto, Databricks SQL, Snowflake.
- Supports both real-time and batch processing.

d. Governance & Security

- Unified catalog (Unity Catalog, AWS Glue, Hive Metastore).
- Access control, lineage, audit trails.

Benefits of the Lakehouse Architecture

- Unified platform for BI, AI, ML, and analytics.
- High performance SQL queries on data lake storage.
- Cost-efficient (compute and storage are decoupled).
- Open and interoperable (supports open formats).
- End-to-end governance across all data types.
- Eliminates duplication between lake + warehouse systems.

Popular Lakehouse Technologies

- Delta Lake (Databricks)
- Apache Iceberg (Netflix, AWS, Snowflake)
- Apache Hudi (Uber)
- Databricks Lakehouse Platform
- AWS Lake Formation
- Snowflake's Unistore

Why Modern Data Warehouses
Modern data warehouses differ from traditional warehouses because they are built to handle massive data volumes, real-time processing, cloud scalability, and advanced analytics. They support today's business needs far better than old on-premise, rigid warehouse systems.

Cloud-Native & Highly Scalable
Modern warehouses (Snowflake, BigQuery, Redshift) are built for the cloud.
They allow:

- Elastic scaling (scale up/down automatically)
- Separation of compute & storage
- Pay-as-you-use model, reducing cost
- No hardware maintenance

This flexibility makes them ideal for big data and unpredictable workloads.

Support for Semi-Structured Data
Traditional warehouses only supported rows and columns.
Modern warehouses support:

- JSON
- Avro
- Parquet
- XML

This allows storage and querying of diverse data formats used by modern applications.

High Performance with MPP
Modern warehouses use **Massively Parallel Processing (MPP)**:

- Workload is split across multiple nodes
- Queries run in parallel
- Extremely fast for analytical workloads

This enables complex reports and dashboards at high speed.

Built-in Automation & Zero Maintenance
Modern warehouses handle:

- Automatic scaling
- Automatic optimization

- Automatic backup and recovery
- Automatic query tuning

This reduces the need for DBAs and manual configuration.

Strong Integration with BI, Data Lakes & ML
Modern warehouses are designed to plug into:
- Data lakes (Lakehouse architecture)
- BI tools (Power BI, Tableau)
- Data engineering tools (Spark, Airflow)
- Machine learning tools (Databricks, SageMaker)

This makes them part of a unified analytics ecosystem.

Integration with DataLakes:
Modern analytics architectures often combine **Data Lakes** (flexible raw storage) and Data Warehouses (structured analytics). Integrating them creates a unified environment for end-to-end data analytics, BI, and machine learning.

Why Integrate Data Lakes and Data Warehouses?

- Data lakes store massive raw data cheaply (logs, IoT, images, JSON).
- Warehouses provide fast SQL analytics on curated data.
- Integration enables:
    - Unified data strategy
    - Seamless movement from raw → refined → analytics
    - Lower storage costs + higher performance

Different Integration options:

ETL (Extract → Transform → Load) from Data Lake to Warehouse
1. Raw data lands in the data lake (S3, ADLS, GCS).
2. Data is cleaned, modeled, and transformed.
3. Curated datasets are loaded into the warehouse (Snowflake, Redshift, BigQuery, Synapse).

When to use:
- Strong governance and curated schemas needed.
- BI reporting requires structured data.

ELT (Extract → Load → Transform)
1. Data lake stores raw data.
2. Raw or semi-structured data is directly loaded into the warehouse.
3. Transformations happen inside the warehouse (SQL-based).

Advantages:
- Faster ingestion
- Uses warehouse compute for transformations

Direct Query from Warehouse to Data Lake
Modern warehouses support querying files directly in the data lake.
Examples:
- Snowflake External Tables
- Redshift Spectrum
- BigQuery external tables
- Synapse Serverless SQL

Benefits:
- No need to load data into warehouse

- Keeps large datasets in low-cost storage
- Simplifies architecture

Lakehouse Integration

Using open formats like Parquet, Delta Lake, or Iceberg to unify lake + warehouse.

- Data lake stores the data
- A Lakehouse engine provides:
  - ACID transactions
  - Schema enforcement
  - Versioning
- Warehouse can read/write these formats directly

Tools:

- Databricks Lakehouse
- Snowflake Iceberg Tables
- AWS Glue + Apache Iceberg
- Delta Lake with Spark

3. Integration Components

a. Storage Layer

- S3 / ADLS / GCS
- Open formats: Parquet, Avro, Delta, Iceberg, Hudi

b. Metadata Layer

- Hive Metastore
- AWS Glue Catalog
- Unity Catalog
- Snowflake Catalog

c. Ingestion Layer

- Batch: Airflow, AWS Glue
- Streaming: Kafka, Kinesis, Pub/Sub

d. Processing Layer

Spark, Databricks, Flink , dbt, Warehouse SQL engines

PostgreSQL is an open-source, object-relational database management system known for:

- Strong ACID compliance
- Advanced SQL capabilities
- Extensibility (custom functions, types, operators)
- Support for structured + semi-structured data (JSONB)
- Performance, indexing strategies, and reliability

It is widely used in finance, e-commerce, analytics, government systems, and SaaS applications.

Key Features

- ACID transactions
- MVCC (Multi-Version Concurrency Control) for high concurrency
- JSON/JSONB support for semi-structured data
- Extensible architecture (functions, operators, plugins)
- Rich indexing: B-tree, Hash, GIN, GiST, BRIN
- Replication & partitioning

PostgreSQL Architecture

- **Client Layer** – applications, APIs
- **Postgres Server** – handles SQL parsing, planning, execution
- **Storage Layer** – manages tables, indexes, WAL logs
- **WAL (Write-Ahead Log)** – ensures durability and crash recovery

Create a database

```
CREATE DATABASE company_db;
```

Create a table

```
CREATE TABLE employees (
   emp_id SERIAL PRIMARY KEY,
   name VARCHAR(100),
   department VARCHAR(50),
   salary NUMERIC(10,2),
   join_date DATE
);

INSERT INTO employees (name, department, salary, join_date)
VALUES ('Amit Kumar', 'IT', 75000, '2023-05-01');
```

Query data

```
SELECT name, salary FROM employees WHERE department = 'IT';
```

Some Example Queries:

```
Whenever an employee's salary increases by more than 20%, log the change in an audit table.
Create audit table
CREATE TABLE salary_audit (
    emp_id INT,
    old_salary NUMERIC,
    new_salary NUMERIC,
    changed_at TIMESTAMP DEFAULT NOW()
);

Handling Semi-Structured Data (JSONB Example)
Use Case: Store and query customer preferences (dynamic data).
Create table with JSONB
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    preferences JSONB
);

Insert JSON data
INSERT INTO customers (name, preferences)
VALUES ('Rahul', '{"theme": "dark", "notifications": {"email": true, "sms": false}}');

Query nested JSON values
SELECT name
FROM customers
WHERE preferences->'notifications'->>'email' = 'true';

CREATE INDEX idx_preferences_gin
ON customers USING GIN (preferences);
Improves performance for heavy JSON queries.

Useful for logs, orders, or time-series data.
Create a partitioned table
CREATE TABLE orders (
    order_id SERIAL,
    order_date DATE NOT NULL,
    amount NUMERIC
) PARTITION BY RANGE (order_date);
Create monthly partitions
CREATE TABLE orders_2025_01 PARTITION OF orders
FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');

This improves performance and makes data management easier.
```

Two users update the same row at the same time:

- PostgreSQL keeps multiple versions of the row
- Readers never block writers

- Ensures high concurrency

```
Example transaction:
BEGIN;
UPDATE employees SET salary = salary + 5000 WHERE emp_id = 1;
COMMIT;
MVCC ensures this doesn't block read operations from other users.
```

MPP (Massively Parallel Processing)

MPP is a distributed computing architecture where a large query or job is split into many smaller tasks, and each task runs in parallel on different nodes in a cluster to achieve very high performance.

Key Characteristics of MPP

Multiple Nodes Working Together

An MPP system uses many servers (nodes).

Each node has:

- Its own CPU
- Its own memory
- Its own storage

Nodes work independently but collaborate to finish the query.

Parallel Query Execution: A big SQL query is broken into sub-queries and executed across all nodes simultaneously.

Shared Nothing Architecture: Nodes do not share memory or disk.

This avoids bottlenecks and improves performance.

Linear Scalability: If you double the number of nodes, you almost double performance.

Suitable for Data Warehouses

MPP is the core engine for:

Redshift, Snowflake, Google BigQuery, Azure Synapse, Greenplum, Teradata

Why MPP is fast

a. Data Is Distributed

- Large tables are partitioned across multiple nodes.
- Each node stores only a slice of data.

b. Queries Run in Parallel: When a query runs, each node processes its part of the data simultaneously.

c. No Resource Contention: Because nodes don't share memory/disk, they don't block each other.

Example Scenario:

E-commerce Sales Analytics: You have a 2 TB sales table and want to run a query

```
SELECT category, SUM(amount)
FROM sales
WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31'
GROUP BY category;
```

Without MPP (Normal PostgreSQL / MySQL)

- One server scan 2 TB, sorts, groups, and aggregates.
- Takes hours.

With MPP (e.g., Redshift / Snowflake / BigQuery)

Data is distributed like this:

Node	Data Portion
Node 1	500 GB
Node 2	500 GB
Node 3	500 GB
Node 4	500 GB

How MPP works here

1. Query is sent to all nodes in parallel.
2. Each node processes only its 500 GB.
3. Each node produces a partial result like:
    - Category: Electronics → ₹10,00,000
    - Category: Books → ₹5,00,000
4. Final node combines results and returns final output.

Time reduces drastically
Hours → seconds or minutes.

When to Use MPP

MPP is perfect when:

- Data is very large (hundreds of GB → PB)
- Queries involve complex aggregations
- You need fast analytics
- Real-time dashboards
- BI workloads (Power BI, Tableau)
- Data warehouse workloads

Not ideal for:

- OLTP
- High-frequency small transactions

Snowflake – A Quick Overview

Snowflake is a modern cloud-based data warehousing platform designed for high performance, scalability, and ease of use. It is offered as a fully managed service on cloud providers like AWS, Azure, and Google Cloud.

Snowflake Architecture overview

Snowflake has 3 layers:

1. Cloud Storage Layer
    - Stores all data in compressed, columnar format
    - Supports structured + semi-structured (JSON, Avro, Parquet)
    - Infinite storage
2. Compute Layer (Virtual Warehouses)
    - Independent compute clusters
    - Scale up (more power) or scale out (more clusters)
    - Auto-suspend and auto-resume
    - Multiple teams can query the same data without performance conflict
3. Cloud Services Layer
    - Metadata
    - Access control
    - Query optimization
    - Security, transactions, governance

Why Snowflake is popular

- No infrastructure management
- Pay-as-you-go pricing
- High performance with automatic optimization
- Suitable for analytics, BI, ML workflows
- Works with many tools (Power BI, Tableau, dbt, etc.)

Key Features
Separation of Storage and Compute

You can scale them independently:

- "Storage grows" without affecting compute
- "Compute costs" only when queries run

Zero Copy Cloning: Create a copy of a database/table instantly without duplicating data. Useful for dev/test environments.
Time Travel: Query historical data (1–90 days):

```
SELECT * FROM sales AT(OFFSET => -2);
```

Helps recover deleted or changed data.
Semi-Structured Data Handling
Native support: JSON, Parquet, Avro, ORC Using VARIANT column type.

Data Sharing
Share data securely with: Partners, Customers, Other teams, without data duplication.

Automatic Optimization
No need for:

- Indexing
- Vacuum
- Manual tuning

Feature	**Traditional DW**	**Snowflake**
Scalability	Limited	Infinite
Admin	High	Zero-maintenance
Semi-structured data	Poor	Native support
Concurrency	Low	High via multi-cluster
Cost	Fixed	Pay-as-you-go
Deployment	On-prem	Multi-cloud

Amazon Redshift
Amazon Redshift is a fully managed, cloud-based MPP data warehouse service built on AWS.
It is designed for fast analytical queries, large-scale data storage, and BI workloads.

Redshift Architecture
Redshift uses a **cluster-based architecture**:
Leader Node

- Coordinates the query execution
- Receives SQL queries from clients
- Generates execution plans
- Aggregates results and sends back to users
- Does NOT store data

Compute Nodes

- Store data in columnar format
- Perform actual query execution
- Each node has CPU, memory, storage
- Can have multiple nodes in a cluster

More compute nodes = higher performance and storage.
Columnar Storage

- Data stored column-by-column

- Greatly reduces I/O
- Ideal for analytical queries (SUM, COUNT, GROUP BY)

MPP (Massively Parallel Processing)
- Data is distributed across compute nodes
- Queries are split and run in parallel
- Great for large tables / analytics

Use Case: Credit Card Fraud Detection Analytics Platform
Bank wants:
- Daily & hourly transaction analytics
- Fraud detection using rules
- Real-time dashboards
- Integration with S3 data lake

Data Sources
- Transaction logs (CSV/hourly batch) → stored in S3
- Customer info → from operational DB
- GPS/Device logs → JSON
- Fraud rules → JSON configs

Load into Redshift
Load transactions:
```
COPY fact_transactions FROM 's3://bank/transactions/2024/' IAM_ROLE
'redshift-role' CSV;
```

Load JSON logs:
```
COPY stg_device_logs FROM 's3://bank/devices/' IAM_ROLE 'redshift-role'
FORMAT AS JSON 'auto';
```

Create Warehouse Schema
Dimensions
- dim_customer
- dim_location
- dim_device

Fact table
```
CREATE TABLE fact_transactions (
    txn_id BIGINT,
    customer_id INT,
    amount DECIMAL(10,2),
    device_id VARCHAR,
    location VARCHAR,
    txn_timestamp TIMESTAMP
)
DISTKEY(customer_id)
SORTKEY(txn_timestamp);
```

Fraud Detection Analytics
High-value suspicious transactions:

```
SELECT
    customer_id,
    txn_id,
    amount, location FROM fact_transactions
```

```
WHERE amount > 50000
AND location <> (SELECT home_location FROM dim_customer WHERE id =
customer_id);
```

Location anomaly detection:

```
SELECT
    customer_id,
    COUNT(*) AS abnormal_txn_count
FROM fact_transactions
WHERE location <> expected_region
GROUP BY customer_id
HAVING COUNT(*) > 10;
```

Querying Data Lake with Redshift Spectrum

```
Transactions archived in S3:
SELECT *
FROM spectrum.archive_transactions
WHERE year=2024
 AND amount > 10000;
No need to load older data → cost-efficient.
```

Medallion Architecture

The **Medallion Architecture** is a **multi-layered data design** used in data lakes/lakehouses to improve data quality, structure, governance, and performance.

It organizes data into **three layers**:

1. **Bronze Layer** – Raw data
2. **Silver Layer** – Cleansed & transformed data
3. **Gold Layer** – Business-ready, analytics-focused data

This ensures a smooth flow of data from **ingestion → refinement → consumption**.

Bronze Layer (Raw Layer)

Store *raw, unprocessed* data exactly as it arrives.

Characteristics

- Contains uncleaned, unvalidated data
- Can be in any format: CSV, JSON, logs, parquet, images, etc.
- Used for auditing or replaying pipelines
- Large volume, low quality

Examples of Bronze Data

- Transaction logs from banking systems
- IoT sensor stream
- API output (JSON)
- Kafka streaming messages
- CSV/Parquet files from various sources

Example SQL / Pseudocode

```
CREATE TABLE bronze_transactions AS SELECT * FROM external_s3_raw;
```

Silver Layer (Cleaned & Enriched Layer)

Convert raw data into **clean, structured, and enriched** form.

Characteristics

- Cleaned: duplicates removed, null handled
- Typed and validated
- Joined with reference data
- Semi-structured normalized
- Ready for intermediate analytics

Examples of Data Cleaning

- Convert string timestamps → proper datetime
- Remove corrupted records
- Standardize names, phone numbers
- Fix schema mismatches
- Filter invalid transactions

Example SQL

```
CREATE TABLE silver_transactions AS
SELECT txn_id, CAST(timestamp AS TIMESTAMP) AS txn_timestamp,
   amount, customer_id, location FROM bronze_transactions WHERE amount > 0;
```

Gold Layer (Business / Analytics Layer)
Provide business-ready, aggregated, modelled data that BI teams and data scientists use.

Characteristics

- Fact and dimension models (Star Schema)
- Used for dashboards, ML features, reporting
- Highly curated, best quality

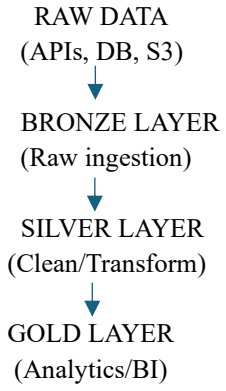
Examples

- Customer 360 table
- Daily revenue summary
- Fraud risk scoring table
- Monthly aggregated transactions
- Marketing segmentation dataset

Example SQL

```
CREATE TABLE gold_daily_revenue AS
SELECT DATE(txn_timestamp) AS txn_date, SUM(amount) AS total_revenue
FROM silver_transactions GROUP BY txn_date;
```

How Data Flows in Medallion Architecture



Why Medallion Architecture is Important

- Standardizes data quality
- Makes pipelines modular and scalable
- Supports governance and lineage
- Easy rollback and debugging
- Enables incremental data processing
- Perfect for lakehouse platforms (Databricks, Snowflake, Redshift Spectrum)

Financial Services Fraud Analytics
If we take the same example scenario financial application the layers go as fallows

Bronze Layer
Raw data ingested:

- Card transactions
- Customer logs
- Device metadata

Silver Layer

- Remove duplicates
- Clean timestamps
- Validate transactions

Gold Layer

- Customer risk scoring table
- Daily fraud summary dashboard

File Formats for Modern Data Lakes
Modern data lakes store diverse and large-scale data. Choosing the right file format is critical for performance, cost, compression, schema evolution, and analytics.

File formats are generally categorized into:
1. Raw Formats
2. Columnar Formats (optimized for analytics)
3. Row-based Formats (optimized for streaming / transactions)

Raw File Formats
CSV (Comma Separated Values)

- Simple, human-readable
- Supported by all tools
- No compression, no schema
- Slow to process for big data

Use case:
Exporting logs, transactional data from OLTP systems for initial ingestion.

JSON

- Semi-structured
- Good for nested data
- Heavy and not optimized for analytics

Use case:
Storing API responses, clickstream logs, IoT payloads.

XML

- Strict hierarchical structure
- Verbose
- Used in enterprise legacy systems

Use case:
Banking/insurance systems export regulatory or batch data using XML.

Columnar File Formats (Modern Analytics-Optimized)
These are the most important for modern data lakes & lakehouses.

Parquet

- Columnar
- Very high compression
- Supports schema evolution
- Optimized for Spark, Hive, Presto, Snowflake, Redshift Spectrum
- Most recommended format for analysis

Use case:
Storing fact tables, historical data, analytics datasets.

ORC (Optimized Row Columnar)

- Columnar
- Faster read performance than Parquet in Hive ecosystems
- Good for write-heavy workloads
- Also supports schema evolution

Use case:
Large Hive/EMR clusters, big ETL pipelines on Hadoop.

Avro

- Row-based but still used for big data
- Schema stored with the file
- Excellent for streaming systems

Use case:
Kafka → HDFS/S3 ingestion pipelines
Good for write-heavy pipelines.

Binary Formats for specific Use Cases
Delta Lake Format (Databricks)

- Built on Parquet
- Supports ACID transactions
- Time travel
- Schema enforcement
- Ideal for lakehouse architecture

Use case:
Maintain bronze → silver → gold tables with reliability.

Apache Iceberg

- Table format over Parquet/ORC
- ACID + snapshots + branching
- Vendor-neutral (Spark, Flink, Snowflake, Athena)

Use case:
Data lakes requiring SQL-like reliability and version control.

Apache Hudi

- Designed for streaming ingestion and incremental pipelines
- Supports upserts and deletes

- Used by Uber

Use case:
Real-time IoT, change data capture (CDC), upsert-heavy pipelines.

Which Format to Use?

Requirement	Best File Format
Analytics, BI, dashboards	Parquet
High compression & fast reads	Parquet / ORC
Streaming ingestion	Avro / Hudi
Upserts in data lake	Hudi / Delta
ACID + time travel	Delta / Iceberg
Legacy enterprise export	CSV / XML
Nested JSON structures	JSON / Parquet after flattening

Apache Spark
Apache Spark is an open-source, distributed computing engine designed for big data processing, real-time analytics, and machine learning. It is widely used in modern data engineering and analytics pipelines because of its speed, scalability, and unified API.

Why Spark?
In-memory processing
- Spark keeps data in RAM rather than disk → 10–100x faster than Hadoop MapReduce.
- Unified analytics engine
- Supports multiple workloads: Batch processing, Streaming, Machine learning, Graph processing, SQL analytics

Language support
APIs available in: Python (PySpark), Scala, Java, R, SQL (Spark SQL)
Fault-tolerant & scalable
Runs on large clusters; automatically handles node failures.
Works with many storages systems
Supports: HDFS**,** S3**,** Azure Data Lake**,** GCS**,** Local filesystem**,** JDBC sources

Spark Core Components
Spark Core
- Foundation for all Spark modules
- Handles distributed task scheduling, memory management, and fault tolerance

Spark SQL
- Module for SQL queries
- Creates DataFrames & Datasets
- Supports Hive Metastore
- Includes Catalyzer optimizer for query optimization

Example:

```
SELECT customer_id, SUM(amount) FROM transactions GROUP BY customer_id;
```

Spark Streaming (Structured Streaming)
- Real-time or near-real-time data processing
- Handles data from Kafka, Kinesis, Sockets, Flume

Spark Architecture

Driver Program

- Coordinates execution
- Converts your code into tasks
- Sends tasks to cluster

Cluster Manager

- YARN, Kubernetes, Mesos, or Spark Standalone

Executors

- Run tasks
- Store data (memory + disk)
- Exchange data via shuffles

Important Spark Concepts

RDD (Resilient Distributed Dataset)

- Low-level API
- Immutable distributed collection of objects
- Supports transformations (map, filter) and actions (count, collect)

DataFrame

- High-level API (like SQL table)
- Optimized by Catalyst engine
- Most widely used today

Dataset

- Type-safe DataFrame (Scala/Java)

Spark Execution Flow

- User writes code → Driver program
- Driver creates DAG (Directed Acyclic Graph)
- DAG Scheduler creates task stages
- Tasks are distributed to Executors
- Executors execute tasks in parallel
- Results returned to driver or written to storage

Business Problem

A bank wants to:

Process real-time credit card transactions for monitoring and audit logging.

Generate daily summary reports for:

- Total transactions
- Total amount
- Transactions by region / branch / card type

a. Explain how Apache Spark can be used to design this end-to-end pipeline.

In your answer, describe the role of:

i. Spark Structured Streaming
ii. Batch processing with Spark SQL
iii. The use of bronze, silver, and gold layers in a data lake
iv. Parquet as a storage format
v. Output for BI or reporting layers

Data Sources

Streaming source

- Credit card transaction stream coming from **Kafka**

  Example fields:

  ```
  transaction_id, timestamp, card_no, merchant_id, amount,
  location
  ```

Batch source

- Daily customer master data
- Stored as **CSV** or **Parquet** in S3/Data Lake

```
customer_id, name, branch, city
```

Ingestion Layer
Spark listens to Kafka topic:

```
df_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "broker:9092") \
    .option("subscribe", "transactions") \
    .load()
```

Transformation Layer
Real-time transformations

- Parse JSON
- Add derived fields (hour, date, region)
- Basic validations (amount > 0, timestamp not null)

Batch transformations

- Clean customer master data
- Convert CSV → Parquet
- Standardize column names

Storage Layer in Data Lake
Bronze Layer (Raw)

- Store transaction stream **as-is** in JSON

Silver Layer (Cleaned)

- Write cleaned streaming data into **Parquet**

```
df_cleaned.writeStream.format("parquet").outputMode("append")
```

Gold Layer (Aggregated)
Daily ETL job using Spark batch:

- Total number of transactions per day
- Total transaction amount per day
- Transactions by city/branch/merchant

Results saved as:

- **Parquet tables**
- Or loaded into **Redshift/Snowflake**

Daily Summary Report Generation

```
SELECT
    date(timestamp) AS txn_date,
    COUNT(*) AS total_transactions,
    SUM(amount) AS total_amount,
    city
FROM silver_transactions
GROUP BY date(timestamp), city;
```

Write output to:

- Power BI / Tableau dashboard
- S3 as Parquet
- Warehouse table for reporting

Modern Enterprise Data Stack

The Modern Enterprise Data Stack is a cloud-native architecture used by organizations to collect, store, process, analyze, and govern data efficiently. It replaces traditional monolithic systems with modular, scalable components.

Key Characteristics of a Modern Data Stack

- Cloud-native (AWS, Azure, GCP)
- Fully scalable & elastic
- Decoupled compute and storage
- Supports batch + streaming
- Low maintenance & automated
- Integration with BI, AI, and data apps
- Pay-as-you-go cost model

Core Components of the Modern Enterprise Data Stack

A. Data Ingestion Layer

Tools: Kafka, AWS Glue

Supports:

- Batch ingestion
- Real-time streaming ingestion
- CDC (Change Data Capture)

Used to Bring data from databases, apps, APIs, IoT devices into the data platform.

Storage Layer (Data Lake / Lakehouse)

Cloud storage: S3, Azure Data Lake Gen2, Google Cloud Storage

File formats: Parquet, ORC, Avro, Delta, Iceberg, Hudi

Used to Store raw + cleaned + transformed data at low cost.

Data Warehouse / Query Engine

Modern compute engines: Snowflake, BigQuery, Redshift, Databricks SQL, Athena

Used for fast SQL analytics, BI reporting, dashboards.

Transformation Layer (ELT)

Tools: Spark, Databricks, AWS Glue ETL

Used to transform raw data into analytics-ready tables
(bronze → silver → gold model).

Orchestration Layer

Tools: Airflow

Used to schedule, coordinate, and monitor pipelines.

BI / Analytics Layer

Tools: **Power BI**, **Tableau**

Used for Dashboards, reporting, self-service analytics.

Workflow of the Modern Enterprise Data Stack

1. **Data Ingestion**
    - Kafka streams real-time data (transactions, logs)
2. **Landing Zone → Data Lake (Bronze)**
    - Store raw data in S3/ADLS/GCS in JSON/CSV
3. **Cleaning & Standardization (Silver)**
    - Spark/dbt cleans, validates, transforms
    - Output stored in Parquet/Delta
4. **Curated Gold Layer**
    - Business-friendly tables (facts & dimensions)
    - Stored in Delta/Parquet/Snowflake schema
5. **Warehouse Analytics**
    - Fast SQL queries using Snowflake/Redshift/BigQuery
6. **BI Dashboards**
    - Reports built in Power BI/Tableau
7. **Governance & Lineage**
    - Catalog, policy enforcement, auditing

Retail / E-commerce Scenario

- Ingest customer, order, website logs via Kafka
- Store raw data in S3 (Bronze)
- Transform using Spark/dbt to create customer and sales facts (Silver/Gold)
- Query with Snowflake for BI reporting
- Power BI dashboard shows sales trends and inventory status

Apache Iceberg
Apache Iceberg is a high-performance table format designed for data lakes. It brings ACID transactions, schema evolution, time travel, and high query performance to data stored in files like Parquet, ORC, and Avro. It enables building reliable Lakehouse architectures without locking into a single vendor.

The Problem It Solves:
Traditional data lakes have limitations:

- No ACID transactions
- Difficulty handling deletes/updates
- Slow query performance
- Lack of versioning
- Schema mismatch issues
- Poor management of huge tables

Iceberg fixes all these issues while keeping data in the lake (e.g., S3, GCS, ADLS).

Schema Evolution (without rewriting data)
Supports:

- Add / drop columns
- Rename columns
- Reorder columns

Iceberg stores schema metadata separately. No need to rewrite files.
Time Travel
You can query data *as it was* at a previous snapshot.

Example:

```
SELECT * FROM sales TIMESTAMP AS OF '2024-01-01';
```

When to choose Iceberg

- Heavy read and analytics workloads
- Multi-engine compatibility needed
- Long-term historical data management

Apache Hudi

Apache Hudi is a data lake framework designed for real-time ingestion, upserts, and incremental processing on large datasets.

Hudi Storage Types

- **Copy-on-Write (CoW):**
  - Reads are fast
  - Updates rewrite files
- **Merge-on-Read (MoR):**
  - Faster writes
  - Reads merge base + log files

Best Use Cases: Streaming data ingestion**,** Change Data Capture (CDC)**,** Upsert-heavy workloads**,** Event-driven systems.

Example:
Updating customer profile data arriving every minute from Kafka → Hudi handles upserts efficiently.

When to choose Hudi

- Frequent updates and deletes
- Near real-time analytics
- Streaming-first architectures

Feature	Apache Iceberg	Apache Hudi
Primary Focus	Analytics & BI	Streaming & Upserts
Update Pattern	Batch-oriented	Upsert-heavy
Time Travel	Strong snapshot support	Limited (commit-based)
Streaming Support	Moderate	Excellent
Partition Handling	Hidden partitions	Explicit record keys
Best Fit	Warehouses on lakes	Real-time pipelines

Data Ingestion Patterns

Data ingestion is the process of **collecting data from source systems and loading it into a data platform** (data lake, lakehouse, or warehouse).

Batch Ingestion: Data is collected and ingested at **fixed intervals** (hourly, daily, weekly).

- Simple to implement
- Suitable for large volumes
- Higher latency (not real-time)

Example:

- Daily sales CSV files from POS systems

Streaming (Real-Time) Ingestion: Data is ingested **continuously as events occur**.

- Low latency
- Supports real-time analytics
- More complex infrastructure

Some Examples:

- Clickstream data via Kafka
- Credit card transactions streaming

Micro-Batch Ingestion: Data ingested in **small time windows** (every few seconds/minutes).

- Balance between batch and streaming
- Easier than pure streaming
- Used by Spark Structured Streaming

Hybrid Ingestion: Combination of batch + streaming.

Examples:

- Customer master data (batch)
- Transactions (streaming)

2. Change Data Capture: CDC is a technique to **capture and propagate only data changes** (INSERT, UPDATE, DELETE) from source databases.

Why CDC is Needed

- Avoid full table reloads
- Reduce load on OLTP systems
- Enable near real-time replication

How it works in simple
Reads database logs (WAL, binlog, redo logs) → Captures row-level changes → Sends changes to Kafka or data lake.

Example:

Employee table update:

```
UPDATE employees SET salary = 90000 WHERE emp_id = 101;
```

CDC captures:

```
emp_id=101, old_salary=75000, new_salary=90000
```

Lambda Architecture
Lambda Architecture is a data processing design that combines batch processing and real-time processing to handle both historical and real-time data.

Core Layers
Batch Layer
- Processes full historical data, High accuracy, High latency

Speed Layer
- Processes real-time streaming data, Low latency, Approximate results

Serving Layer
- Merges batch + speed results, Serves queries to applications

Kappa Architecture
Kappa Architecture is a simplified alternative to Lambda that uses only a streaming pipeline.

Core points:
- Treat all data as a stream
- Reprocess data by replaying streams
- No separate batch layer

Components
- Kafka (event log)
- Stream processing engine (Spark/Flink)
- Serving layer (DB/warehouse)

- Not ideal for very large historical reprocessing
- Requires strong streaming infrastructure

Features	**Lambda**	**Kappa**
Processing	Batch + Streaming	Streaming only
Complexity	High	Low
Code Duplication	Yes	No
Latency	Low + High	Low
Maintenance	Difficult	Easier
Best for	Mixed workloads	Streaming-first systems

Data Cleaning Techniques
Data cleaning is the process of detecting and correcting inaccurate, incomplete, inconsistent, or irrelevant data to improve data quality for analysis and reporting.

Common Data Quality Problems: Missing values**,** Duplicate records**,** Inconsistent formats**,** Invalid values**,** Outliers**,** Incorrect data types

Handling Missing Values

- Identify null or blank values
- Decide whether to remove or fill them

- Delete rows
- Replace with: Mean / Median (numerical data), Mode / default value, Previous value (time-series)

Removing Duplicates

- Identify repeated records using keys
- Keep one valid record

- DISTINCT
- Deduplication using primary keys or window functions

Data Type Standardization: Convert columns to correct data types

- String → Date
- String → Integer

Format Standardization: Ensure consistent data formats

- Date formats (YYYY-MM-DD)
- Phone numbers
- Currency symbols

Handling Inconsistent Values: Replace different representations with standard values

Examples

- M / Male / male → Male
- yes / Yes / Y → Yes

Validating Range and Constraints: Ensure values fall within valid ranges

Examples

- Age ≥ 0
- Salary > 0

Different Joins in Apache Spark

A join combines rows from two DataFrames based on a common key.

Inner Join: Returns only matching rows from both DataFrames. When data must exist in both tables.

Spark Syntax

```
df1.join(df2, on="id", how="inner")
```

Left Outer Join: Returns all rows from left DataFrame, Matching rows from right DataFrame, Non-matching → NULL

Spark Syntax

```
df1.join(df2, on="id", how="left")
```

Right Outer Join: Returns all rows from right DataFrame, Matching rows from left DataFrame, Non-matching → NULL

Spark Syntax

```
df1.join(df2, on="id", how="right")
```

Full Outer Join: Returns all rows from both DataFrames, Matches where possible, Non-matches → NULL on either side

Spark Syntax

```
df1.join(df2, on="id", how="outer")
```

Cross Join: Combines every row of df1 with every row of df2, Output size = N × M

Spark Syntax

```
df1.crossJoin(df2)
```

Left Semi Join: Returns rows from left DataFrame, Only checks existence in right DataFrame, Does not include right table columns

Spark Syntax

```
df1.join(df2, on="id", how="left_semi")
```

Left Anti Join: Returns rows from left DataFrame, Where there is NO match in right DataFrame

Spark Syntax

```
df1.join(df2, on="id", how="left_anti")
```

Self Join: A DataFrame joins with itself, Requires aliasing

Spark Syntax

```
from pyspark.sql.functions import col

e1 = emp.alias("e1")
e2 = emp.alias("e2")
e1.join(e2, col("e1.manager_id") == col("e2.emp_id"), "inner")
```

Example

Employee–manager relationship.

Broadcast Join: Broadcasts **small DataFrame** to all executors**,** Avoids shuffle**,** Improves performance

Spark Syntax

```
from pyspark.sql.functions import broadcast

df_large.join(broadcast(df_small), on="id", how="inner")
```

Data Quality (DQ) Checks and Monitoring

Data Quality checks are rules applied to data to ensure it is accurate, complete, consistent, valid, and timely before it is used for analytics or reporting.

Why DQ Checks are Important

- Prevent wrong business decisions
- Detect data pipeline failures early
- Ensure trust in dashboards and reports

Common Data Quality Checks

Null / Missing Value Check: Check for mandatory columns having NULLs

Example
```
SELECT COUNT(*) FROM orders WHERE order_id IS NULL;
```

Uniqueness Check: Ensure primary key columns have no duplicates

Example
```
SELECT order_id, COUNT(*) FROM orders GROUP BY order_id HAVING COUNT(*) >
1;
```

Range Check: Validate values fall within expected limits
Example
```
SELECT * FROM employees WHERE salary < 0;
```

Referential Integrity Check: Foreign key values must exist in parent table
Example
```
SELECT * FROM orders o LEFT JOIN customers c ON o.customer_id =
c.customer_id
WHERE c.customer_id IS NULL;
```

5. Format Check: Validate date, email, phone formats
Example
```
SELECT * FROM users WHERE email NOT LIKE '%@%';
```

Freshness / Timeliness Check: Ensure data is updated within expected time

Example
```
SELECT MAX(load_date) FROM sales;
```

Volume Check: Validate record count vs expected volume
Example
```
SELECT COUNT(*) FROM daily_sales;
```

DQ Monitoring: Continuous tracking of **data quality metrics over time** and alerting when thresholds are breached.

What is Monitored

- % null values
- Duplicate count
- Record count trends
- Data arrival delays
- Failed DQ rules

DQ Monitoring Flow

1. Run DQ checks
2. Store results in DQ metrics table
3. Compare with thresholds
4. Trigger alerts (email/dashboard)

Apache Kafka
Apache Kafka is a **distributed event-streaming platform** used to **publish, store, and consume streams of data in real time**.

Used for: High-throughput data ingestion**,** Real-time processing**,** Decoupling producers and consumers
Reliable event storage

Core Kafka Components
Producer: Sends messages (events) to Kafka topics
Topic: Logical category where messages are stored
Partition: Topic is split into partitions and enables parallelism and scalability
Broker: Kafka server that stores data
Consumer: Reads messages from topics
Consumer Group: Multiple consumers share workload and each partition → one consumer in a group

Kafka Message Structure
- Key (optional)
- Value (actual data)
- Offset (message position)
- Timestamp

Kafka Data Flow
1. Producer publishes event to topic
2. Kafka stores event in partition
3. Consumer reads event using offset
4. Message retained for configured time

Kafka Retention Model
- Messages are **not deleted after consumption**
- Deleted based on:
  - Time
  - Size

Kafka Use Cases
- Clickstream ingestion
- Log aggregation
- CDC pipelines
- Real-time analytics

Scenario: Clickstream Data Processing

An e-commerce company wants to track user activity on its website such as:
- Home page visits
- Product page views
- Add-to-cart events
- Checkout events

The goal is to:
- Monitor real-time website traffic
- Generate daily and monthly reports
- Identify most visited pages and conversion counts

Data Source
Each user action generates a click event:
user_id, page, event_time, device, session_id

Example:
101, home, 2025-12-12 10:01:05, mobile, s1
101, product, 2025-12-12 10:01:20, mobile, s1

Architecture
Web App → Kafka → Spark Streaming → Data Lake (Iceberg/Hudi) → Warehouse → BI

Workflow

Ingestion (Kafka)

- Web application sends click events to Kafka topic clicks
- Kafka handles high-volume, real-time ingestion

Stream Processing
Spark reads data from Kafka to:

- Parse events
- Add timestamps
- Perform basic aggregations (page counts)

Example:

- Page views per minute
- Unique sessions per page

Handles streaming + batch in one engine.

Storage in Data Lake (Iceberg / Hudi)
Raw Layer (Bronze)

- Stores raw clickstream events
- Append-only
- Used for replay or audit

Clean Layer (Silver)

- Remove invalid events
- Standardize timestamps
- Deduplicate clicks

Aggregated Layer (Gold)

- Daily page view counts
- Session-level summaries

Serving Layer (Warehouse / BI)

- Aggregated data loaded into Snowflake / Redshift
- BI tools (Power BI) query:
    - Top pages
    - Daily traffic trends

Apache Airflow

Apache Airflow is an open-source workflow orchestration tool used to schedule, manage, and monitor data pipelines.

Airflow Concepts

DAG (Directed Acyclic Graph)

- Defines workflow structure
- Tasks + dependencies
- Written in Python

Task: A single unit of work

Example: run Spark job, load data, run SQL

Operator: Template for tasks

- **Common operators:**
  - PythonOperator
  - BashOperator
  - SparkSubmitOperator

Scheduler: Decides when tasks run

Executor: Executes tasks (Local, Kubernetes)

Web UI: Used to monitor DAG runs, view logs, and retry failed tasks.

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

with DAG(
    dag_id="daily_sales_pipeline",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@daily",
    catchup=False
) as dag:

    extract = BashOperator(
        task_id="extract_data",
        bash_command="python extract.py"
    )

    transform = BashOperator(
        task_id="transform_data",
        bash_command="python transform.py"
    )

    load = BashOperator(
        task_id="load_data",
        bash_command="python load.py"
    )

    extract >> transform >> load
```

Airflow Execution Flow

DAG scheduled → Scheduler triggers tasks → Executor runs tasks → Logs stored → Status shown in UI