Big Data Technologies

Agenda

- Spark Core
- Data warehousing
- ETL vs ELT
- Apache Airflow

Spark Core

- Refer day12 notes

RDD Persistence

- By default, RDDs are computed and resides in RAM (of cluster).
- After its use, it is auto released/destroyed.
- If same RDD is to be used for multiple actions, the same RDD can get recomputed multiple times.
- In such case, RDD can be stored temporarily.
- rdd.cache()
    - When rdd is computed first time, it will be cached in cluster RAM.
    - If RAM is less, then some partitions are discarded and recomputed when needed next time.
- rdd.persist(storageLevel)
    - MEMORY_ONLY
        - similar to cache()
    - MEMORY_ONLY_SER
        - When rdd is computed first time, it will be cached in cluster RAM in serialized/compressed format.
        - If RAM is less, then some partitions are discarded and recomputed when needed next time.
        - When used, RDD contents is deserialized.
        - This needs less RAM, but more CPU (for deserialization).
    - MEMORY_AND_DISK

        - When rdd is computed first time, it will be cached in cluster RAM.
        - If RAM is less, then some partitions are stored on disk and loaded back when needed next time.
        - Needs more Disk IO (may be slower).
    - MEMORY_AND_DISK_SER
        - Similar to MEMORY_AND_DISK. But stored in memory in serialized/compressed form.
        - This needs less RAM, but more CPU (for deserialization).
    - DISK_ONLY
        - When rdd is computed first time, it will be stored on disk.
        - Loaded back from disk, when needed next time.
- Cached/Persisted RDDs are discarded when application is terminated.

Spark Performance Tuning

- Memory Management
    - spark.driver.memory -- Memory assigned to driver process (default 1g)
        - Useful in cluster (single node/multi node) and local mode to collect huge result or process huge data
    - spark.executor.memory -- Memory assigned to executor process (default 1g)
        - Useful in cluster (single node/multi node) to process huge data
- Parallism
    - spark.default.parallism -- Default number of partitions to be created in wide transformations e.g. reduceByKey(), ...
    - spark.sql.shuffle.partitions -- Default number of partitions to be created after shuffle stage for structured APIs e.g. dataframes, spark sql, ...
- Shuffle operations
    - spark.shuffle.io.maxRetries -- If shuffle fails, how many times to retry.
- Number of Cores
    - spark.executor.cores -- Number of CPU cores assigned to each executor.
    - spark.cores.max -- Max number of CPU cores for across the cluster for current application.

Data Warehousing

Data Warehouse Fundamentals

1. Introduction

What is a Data Warehouse?

- A **Data Warehouse (DW)** is a centralized repository designed to store integrated, historical data from multiple sources to support decision-making and analytics.

Traditional vs. Modern Data Warehousing

Aspect	Traditional DW	Modern DW
Infrastructure	On-premises servers	Cloud-based solutions
Scalability	Limited	Auto-scalable
Data Types	Structured data only	Structured and unstructured
Processing	Batch processing	Real-time + batch processing
Tools	Informatica, IBM DB2	Snowflake, Redshift, BigQuery

Why Do We Need Data Warehouses?

- **Support for Analytics**: Helps in generating insights from historical data.
- **Data Consolidation**: Combines data from RDBMS, NoSQL, and external sources.
- **Improved Performance**: Optimized for read-heavy operations (OLAP queries).

2. Key Features of a Data Warehouse

1. **Subject-Oriented**: Focuses on specific business areas like sales or marketing.
2. **Integrated**: Resolves inconsistencies in data formats, naming conventions, etc.
3. **Time-Variant**: Stores snapshots of data across time for historical analysis.
4. **Non-Volatile**: Maintains data integrity by ensuring data isn't frequently updated or deleted.

3. Data Warehousing in the Big Data Era

Challenges in Traditional DW Approaches

- High infrastructure costs.
- Limited ability to handle unstructured data.
- Poor scalability for large-scale data processing.

Modern Solutions

- **Cloud Data Warehouses**:
  - Examples: Amazon Redshift, Google BigQuery, Snowflake.
  - Benefits: Scalability, flexibility, cost-effectiveness.
- **Real-Time Data Warehousing**:
  - Tools: Apache Kafka, AWS Kinesis.
  - Benefits: Ingest and process data in near real-time.

Data Lakes vs. Data Warehouses vs. Data Marts vs. Lake House

1. **Data Lake**
   - **Concept**:
     A storage repository that holds large volumes of raw, unstructured, and semi-structured data from various sources in its native format.
   - **Purpose**:
     - Centralized storage for all types of data (structured, semi-structured, and unstructured).
     - Enables data scientists and analysts to process and analyze data as needed.
   - **Applications**:
     - Machine learning model training.
     - Big Data analytics.
     - Storing IoT and streaming data.
   - **Example Tools**:
     Apache Hadoop, Amazon S3, Azure Data Lake.
2. **Data Warehouse**
   - **Concept**:
     A structured repository of processed, historical data, designed for reporting and analysis.

- **Purpose**:
  - Facilitates decision-making through business intelligence.
  - Organizes data into schemas (e.g., star or snowflake) for fast querying.
- **Applications**:
  - Generating business reports (e.g., sales, financial, and customer insights).
  - Supporting OLAP systems for complex analytical queries.
- **Example Tools**:
  Snowflake, Amazon Redshift, Google BigQuery, Microsoft Azure Synapse Analytics.

3. **Data Mart**

- **Concept**:
  A subset of a data warehouse, focused on a specific business domain or department.
- **Purpose**:
  - Provides department-level analytics for faster and more focused insights.
  - Simplifies access to data for non-technical users.
- **Applications**:
  - Sales analysis for a marketing team.
  - Financial reporting for the accounts department.
- **Example**:
  A retail company creates separate data marts for sales, inventory, and customer data.

4. **Lakehouse**

- **Concept**:
  A unified architecture that combines the benefits of data lakes (flexibility) and data warehouses (structure and analytics).
- **Purpose**:
  - Eliminates the need for maintaining separate data lakes and warehouses.
  - Supports both unstructured and structured data with ACID transactions.
- **Applications**:
  - Real-time data analytics.
  - Unified storage for machine learning and business intelligence.

- **Example Tools**:
  Databricks Lakehouse, Google BigLake, Delta Lake.

Comparison Table

Aspect	Data Lake	Data Warehouse	Data Mart	Lakehouse
Data Type	Raw (structured, semi-structured, unstructured).	Processed and structured.	Structured (subset of DW).	Unified (raw + structured).
Purpose	Storage and processing.	Reporting and analytics.	Department-specific analysis.	Unified analytics.
Cost	Cost-effective for large data volumes.	Higher storage and compute costs.	Lower cost (limited scope).	Moderate cost.
Tools/Examples	Hadoop, S3, Azure Data Lake.	Snowflake, Redshift, BigQuery.	Tableau, Power BI, Qlik.	Databricks, Delta Lake.
Primary Users	Data engineers, scientists.	Business analysts, BI teams.	Specific departments.	All users.

When to Use Each?

1. **Data Lake**:
   - When you need to store vast amounts of raw data for future use.
   - Ideal for machine learning or exploratory analytics.
2. **Data Warehouse**:
   - For structured data analysis and generating historical reports.
   - Best suited for business intelligence.
3. **Data Mart**:
   - For department-specific needs.
   - Simplifies access to relevant data for non-technical users.

4. **Lakehouse**:
    - When you want a unified platform for data storage, analytics, and machine learning.
    - Ideal for organizations managing both structured and unstructured data.

5. Comparison: OLAP vs OLTP Systems

- OLAP and OLTP are data management systems used for different purposes.
    - **OLTP**: Databases used to run the business (daily operations).
    - **OLAP**: Databases used to optimize the business (analytical operations).

Aspect	OLAP	OLTP
Purpose	Used for analyzing and querying large amounts of historical data.	Used for managing day-to-day transaction data.
Focus	Data analysis, decision-making, and business intelligence.	Transaction management and real-time data updates.
Users	Executives, analysts, and decision-makers.	Operational staff and application users.
Data Characteristics	Historical and aggregated data.	Current and highly detailed transactional data.
Operations	Complex queries involving joins, aggregations, and data mining.	Simple queries like insert, update, delete, and select.
Database Design	Designed using star, snowflake, or fact constellation schemas.	Designed using normalized schemas (3NF or higher).
Performance Metric	Optimized for read-heavy operations (queries).	Optimized for write-heavy operations (transactions).
Concurrency	Low concurrency (fewer users, complex queries).	High concurrency (many users, simple operations).
Data Volume	Large volumes of data.	Smaller volumes of data compared to OLAP.
Response Time	Slower due to the complexity of queries.	Fast, real-time responses for transactions.
Backup Frequency	Periodic backups (e.g., nightly).	Frequent backups (e.g., every few minutes or hours).
Examples	Data warehouses, reporting tools, BI systems.	Banking systems, order management, inventory systems.
Tools	Apache Hive, Amazon Redshift, Snowflake, Power BI.	MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server.

When to Use OLAP vs. OLTP?

- **OLAP**:
  - Use for strategic decisions, trend analysis, and performance monitoring.
  - Executes on a data warehouse.
  - Example: Understanding customer purchasing trends over the past five years.
    - "What is the total sales revenue for each product category in the last year across all regions?"
      - Requires aggregation, joins, and historical data.
- **OLTP**:
  - Use for operational systems requiring real-time data consistency.
  - Executes on a transactional system.
  - Example: Processing an online purchase order.
    - "Insert a new order into the system."
      - Requires quick response and data consistency.

6. Modern Trends in Data Warehousing

1. **Cloud Data Warehousing**
   - Key Players: Snowflake, BigQuery, Redshift.
   - Benefits:
     - No hardware management.
     - Pay-as-you-go pricing models.
     - Seamless integration with data pipelines and BI tools.
2. **Hybrid Architectures**
   - Combining Data Warehouses and Data Lakes into a **Lakehouse Architecture** for unified storage and analytics.
3. **Real-Time Analytics**
   - Technologies like Kafka and Flink enable real-time processing and integration with warehouses.

Data Warehouse Architecture and ETL Process

1. Data Warehouse Architecture

Core Components of Data Warehouse Architecture

1. **Data Sources**:
    - Internal systems: ERP, CRM, transactional databases.
    - External sources: APIs, third-party services, IoT devices.
2. **ETL (Extract, Transform, Load) Layer**:
    - Extract: Gather data from heterogeneous sources.
    - Transform: Clean, normalize, and integrate data.
    - Load: Store data into the data warehouse.
3. **Data Storage**:
    - Centralized repository optimized for analytics.
    - Data is stored in relational or columnar formats (e.g., Parquet, ORC).
4. **Presentation Layer**:
    - Query tools (Hive, Spark SQL) and BI tools (Tableau, Power BI).
5. **Metadata and Management Tools**:
    - Maintain data catalogs, schemas, and lineage.
    - Examples: Hive Metastore, Apache Atlas, AWS Glue Data Catalog.

2. Data Modeling in Data Warehousing

Data modeling structures the warehouse to ensure efficient querying and data integrity. **A. Star Schema**: A simple model where a central fact table is connected to multiple dimension tables.

B. Snowflake Schema: A more normalized version of the star schema where dimension tables are broken into sub-dimensions.
C. Fact Constellation (Galaxy Schema): A complex model where multiple fact tables share common dimension tables.

3. ETL/ELT Process

ETL Process

- **Extract**:
  - Retrieve data from diverse sources (e.g., relational databases, APIs).
  - Tools: Talend, Informatica, Apache NiFi.
- **Transform**:
  - Cleanse and prepare data (e.g., handling nulls, remove duplication).
  - Enrich data (e.g., calculate new metrics).
  - Apply business rules.
- **Load**:
  - Insert data into a warehouse or lakehouse.
  - Batch vs. Incremental loading.

ELT Process

- ELT (Extract, Load, Transform): Transformations happen inside the warehouse using tools like Spark or Snowflake.
- Ideal for Big Data scenarios where processing power is high.

4. Key terms in ETL Process

1. **Extracting**
   - Process of retrieving data from various sources (e.g., databases, APIs, files).
   - Can handle structured, semi-structured, and unstructured data.
   - **Example**: Using Sqoop to extract data from a MySQL database.

2. **Enrichment**
    - Enhancing raw data with additional information to add value.
    - **Example**: Adding geolocation data to a customer dataset using IP addresses.
3. **Conditioning**
    - Preparing data for analysis by formatting, transforming, or normalizing.
    - **Example**: Converting all date formats to `YYYY-MM-DD` (DATE type).
4. **Scoring**
    - Assigning a value or rank to data based on specific rules or models.
    - **Example**: Assigning credit scores to customers using a predictive model.
5. **Scrubbing (Cleansing)**
    - Removing or correcting errors, inconsistencies, and duplicate entries in data.
    - **Example**: Standardizing "NY", "NewYork", "New York" to "New York" in location fields.
6. **Loading**
    - **Full Load**: Loads all data into the target system, often used for initial setup.
    - **Incremental Load**: Updates the target system with only new or changed data.
    - **Example (Hive)**: Incrementally loading daily sales data into a partitioned table.
7. **Merging**

- Combining multiple datasets into a unified dataset.
- **Example**: Merging customer data from CRM and e-commerce platforms into a single table.

8. **Validating**
    - Ensuring the extracted and transformed data meets predefined criteria.
    - **Example**: Verifying that all mandatory fields in a dataset are non-empty.

9. **Householding**
    - Grouping data records belonging to the same entity, such as a family or company.
    - **Example**: Aggregating customer accounts under a shared household ID.
10. **Delta Updating**
    - Capturing and applying only the changes (inserts, updates, deletes) since the last ETL run.
    - **Example**: Using CDC (Change Data Capture) to update only modified rows in the target system.

5. Logical, Physical, and Cloud Architectures

Logical Architecture

- High-level view of components and their interactions.
- Includes ETL pipelines, data storage, and presentation layers.

Physical Architecture

- Details of how data is physically stored and processed.
- E.g., On-premises clusters, cloud-based storage (S3, GCS).

Cloud Architecture

- Infrastructure hosted on platforms like AWS, GCP, or Azure.
- Benefits: Scalability, reduced costs, disaster recovery.

6. Hands-On Example

Use Case: ETL Workflow Using Apache Airflow

1. **Objective**: Load and transform sample retail data.
2. **Steps**:
    - **Extract**: Read data from CSV files.
    - **Transform**: Clean data using Python scripts.

- **Load**: Insert data into a Hive table.

3. **Airflow DAG Example**:

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract_data():
    # Extract data from sources like files or databases
    pass

def transform_data():
    # Process data to fit DW model
    pass

def load_data():
    # Store data into DW
    pass

with DAG('etl_demo', start_date=datetime(2023, 1, 1), schedule_interval='@daily') as dag:
    extract = PythonOperator(task_id='extract', python_callable=extract_data)
    transform = PythonOperator(task_id='transform', python_callable=transform_data)
    load = PythonOperator(task_id='load', python_callable=load_data)

    extract >> transform >> load
```

Query Optimization Techniques in Data Warehousing

1. Partitioning in Data Warehouses

Definition

- Partitioning involves dividing a large dataset into smaller, manageable parts based on a key.

- Enhances query performance by reducing the amount of data scanned.

Types of Partitioning

1. **Horizontal Partitioning**: Divides data rows based on a condition.
    - Example: Partition sales data by year.
2. **Vertical Partitioning**: Divides columns into smaller subsets.
    - Example: Separate frequently accessed columns into a different table.

Partitioning in Hive

- **Static Partitioning**: Partition values are explicitly specified.
- **Dynamic Partitioning**: Partition values are determined during data load.

Example in Hive

```
-- Create partitioned table
CREATE TABLE sales_partitioned (
    product_id INT,
    quantity INT,
    price FLOAT
)
PARTITIONED BY (year INT, region STRING);

-- Load data into a specific partition
LOAD DATA INPATH '/data/sales/2023'
INTO TABLE sales_partitioned
PARTITION (year=2023, region='US');
```

2. Bucketing in Data Warehousing

Definition

- Bucketing further subdivides partitioned data into fixed-size buckets based on a hash function.
- Useful for optimizing joins and sampling.

Example in Hive

```
-- Create a bucketed table
CREATE TABLE sales_bucketed (
    product_id INT,
    quantity INT,
    price FLOAT
)
CLUSTERED BY (product_id) INTO 4 BUCKETS;

-- Load data
INSERT INTO sales_bucketed VALUES (1, 10, 100.0), (2, 20, 200.0);
```

Advantages:

- Reduces data shuffling in joins.
- Speeds up join query execution.

3. Indexing in Data Warehousing

Definition

- Indexing creates a reference for faster data retrieval.
- Common types:
  - **B-Tree Index**: Efficient for range queries.
  - **Bitmap Index**: Ideal for columns with low cardinality.

Indexing Example in Hive 2.x (Bitmap Index)

```
-- Create an indexed table
CREATE INDEX idx_quantity
ON TABLE sales_partitioned (quantity)
AS 'BITMAP'
WITH DEFERRED REBUILD;

-- Rebuild the index after data load
ALTER INDEX idx_quantity ON sales_partitioned REBUILD;
```

Advantages:

- Speeds up data lookups.
- Reduces full table scans.

4. Query Optimization Techniques

Overview

- Query optimization minimizes resource usage and execution time.
- Techniques include:
  - Query rewriting.
  - Using indexes, partitions, and/or materialized views.
  - Avoiding SELECT *.
  - Aggregating at the source.

Best Practices

1. Use partitioning and bucketing wisely.
2. Avoid joins on large tables without filtering.
3. Leverage materialized views for frequently accessed queries.

5. Example: Query Optimization in Hive

Scenario

Analyze sales data partitioned by year and region for a specific region and year.

Dataset Example

Product_ID	Quantity	Price	Year	Region
1	100	20.5	2023	US
2	200	30.5	2023	EU

Step 1: Create and Load Data

```
-- Create partitioned table
CREATE TABLE sales_partitioned (
    product_id INT,
    quantity INT,
    price FLOAT
)
PARTITIONED BY (year INT, region STRING);

-- Load data into specific partitions
LOAD DATA INPATH '/data/sales/2023/US'
INTO TABLE sales_partitioned PARTITION (year=2023, region='US');
LOAD DATA INPATH '/data/sales/2023/EU'
INTO TABLE sales_partitioned PARTITION (year=2023, region='EU');
```

Step 2: Query Without Optimization

```
-- Scan the entire table
SELECT SUM(quantity), AVG(price)
FROM sales_partitioned;
```

Step 3: Optimized Query with Partition Filtering

```
-- Query specific partition
SELECT SUM(quantity), AVG(price)
FROM sales_partitioned
WHERE year=2023 AND region='US';
```

6. Comparison: Partitioning vs. Bucketing

Aspect	Partitioning	Bucketing
Purpose	Reduces data scan based on conditions.	Optimizes joins and sampling.
Data Subdivision	Divides data into partitions.	Divides data into fixed-size buckets.
Query Optimization	Filters partitions during query.	Reduces shuffle during joins.
Example Use Case	Year-based partition.	User ID-based bucketing.

Slowly Changing Dimensions (SCD) and Materialized Views

1. Fact and Dimension Tables

Overview

Fact and dimension tables form the foundation of star, snowflake, and fact constellation schemas in data warehousing.

Fact Table

- **Definition**: Contains quantitative data (measurable facts) used for analysis.
- **Characteristics**:
  - Contains foreign keys linking to dimension tables.
  - Stores numerical measures (e.g., sales amount, quantity).
- **Example**: Sales Fact Table

Order_ID	Product_ID	Customer_ID	Sales_Amount	Quantity	Date_Key

Dimension Table

- **Definition**: Contains descriptive attributes about the facts.
- **Characteristics**:
  - Stores textual or categorical data.
  - Provides context to fact table measures.
- **Example**: Product Dimension Table

Product_ID	Product_Name	Category	Price

Key Points

- **Granularity**: Fact table granularity defines the level of detail (e.g., daily sales vs. hourly sales).
  - Fine grained: Detailed data is stored (May store individual records)
  - Coarse grained: Summarized data is stored (May store daily average/total)
- **Relationships**: Fact tables have many-to-one relationships with dimension tables.

2. Querying Fact and Dimension Tables

Scenario: Analyze total sales per category.

Schema Setup

- Fact Table: `sales_fact`
  | **Order_ID** | **Product_ID** | **Sales_Amount** | **Quantity** | **Date_Key** |
- Dimension Table: `product_dimension`
  | **Product_ID** | **Product_Name** | **Category** |

```
-- Create sales fact table
CREATE TABLE sales_fact (
    Order_ID INT,
    Product_ID INT,
```

```
    Sales_Amount FLOAT,
    Quantity INT,
    Date_Key DATE
);

-- Create product dimension table
CREATE TABLE product_dimension (
    Product_ID INT,
    Product_Name VARCHAR(100),
    Category VARCHAR(50)
);
```

```
SELECT
    p.Category,
    SUM(s.Sales_Amount) AS Total_Sales,
    SUM(s.Quantity) AS Total_Quantity
FROM sales_fact s
JOIN product_dimension p
ON s.Product_ID = p.Product_ID
GROUP BY p.Category;
```

3. Query Optimization: Using Materialized Views

Definition

- Materialized views store query results for faster retrieval.
- Used for pre-aggregated data or frequently accessed queries.

Example: Materialized View for Sales Aggregation

```
-- In Hive 3.x, Materialized views are allowed only on Transactional tables.
-- So create underlying table (sales_fact) with transactional property.
```

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT
    Product_ID,
    SUM(Sales_Amount) AS Total_Sales,
    SUM(Quantity) AS Total_Quantity
FROM sales_fact
GROUP BY Product_ID;

-- Query the materialized view
SELECT * FROM sales_summary WHERE Product_ID = 101;
```

Benefits

- Reduces query execution time.
- Pre-computed results minimize resource usage.

4. Slowly Changing Dimensions (SCD)

Definition

- Dimensions that change over time in a data warehouse.
- Requires tracking historical data while managing changes.

Types of SCD

1. **Type 1**: Overwrite old data with new data.
    - **Use Case**: When historical data is not required.
    - **Example**: Correcting a spelling error in a customer's name.
2. **Type 2**: Maintain historical data by adding a new record for each change.
    - **Use Case**: Track changes over time.
    - **Implementation**:

- Add effective date and expiry date columns.
- Use a current flag column (e.g., `Is_Current`).

3. **Type 3**: Add new columns to store historical data.
    - **Use Case**: Limited history tracking.
    - **Example**: Storing both old and new values of a customer's city.

5. SCD Implementation in SQL

Example: Customer Dimension (Type 2 SCD)

Customer_ID	Name	City	Effective_Date	Expiry_Date	Is_Current
101	John Doe	New York	2023-01-01	2023-06-01	N
101	John Doe	Los Angeles	2023-06-02	NULL	Y

SQL for Type 2 SCD

```
-- Create customer dimension table
CREATE TABLE customer_dimension (
    Customer_ID INT,
    Name VARCHAR(100),
    City VARCHAR(50),
    Effective_Date DATE,
    Expiry_Date DATE,
    Is_Current CHAR(1)
);
```

```
-- Update old record to set expiry date and current flag
UPDATE customer_dimension
SET Expiry_Date = CURRENT_DATE - INTERVAL 1 DAY,
```

```
    Is_Current = 'N'
WHERE Customer_ID = 101 AND Is_Current = 'Y';

-- Insert new record for updated customer data
INSERT INTO customer_dimension (Customer_ID, Name, City, Effective_Date, Expiry_Date, Is_Current)
SELECT
    Customer_ID,
    Name,
    'Los Angeles' AS City,
    CURRENT_DATE AS Effective_Date,
    NULL AS Expiry_Date,
    'Y' AS Is_Current
FROM customer_dimension
WHERE Customer_ID = 101;
```

Data Warehouse Models and Analytical Queries

1. Star Schema

- A simple schema with one fact table at the center connected to multiple dimension tables.

Characteristics

1. Denormalized structure.
2. Easy to query with simple joins.
3. Faster query performance due to fewer joins.

- **Structure**:
  - **Fact Table**: Contains metrics (e.g., sales revenue).
  - **Dimension Tables**: Contain descriptive attributes (e.g., product, customer, date).
- **Advantages**:
  - Simplicity.
  - Faster query performance due to fewer joins.
- **Disadvantages**:

- Data redundancy in dimensions.

Example Schema

- **Fact Table**: `sales_fact`
- **Dimension Tables**: `customer_dim`, `product_dim`, `time_dim`, `store_dim`.

Diagram Representation

```
            customer_dim
                 |
product_dim --- sales_fact --- time_dim
                 |
              store_dim
```

SQL Example

```sql
-- Query sales by product and store
SELECT
    p.Product_Name,
    s.Store_Name,
    SUM(f.Sales_Amount) AS Total_Sales
FROM sales_fact f
JOIN product_dim p ON f.Product_ID = p.Product_ID
JOIN store_dim s ON f.Store_ID = s.Store_ID
GROUP BY p.Product_Name, s.Store_Name;
```

2. Snowflake Schema

- A normalized schema where dimension tables are further broken into sub-dimension tables.

Characteristics

1. Normalized structure (removes redundancy).
2. Complex queries due to multiple joins.
3. Saves storage space compared to star schema.

- **Structure**:
  - Dimension tables are normalized to reduce redundancy.
- **Advantages**:
  - Saves storage space.
  - Easier to maintain.
- **Disadvantages**:
  - Slower queries due to more joins.

Example Schema

- **Fact Table**: `sales_fact`
- **Normalized Dimension Tables**: `product_dim`, `category_dim`, `customer_dim`, `region_dim`.

Diagram Representation

```
      category_dim
           |
product_dim --- sales_fact --- customer_dim
           |
        region_dim
```

SQL Example

```sql
-- Query sales by category and region
SELECT
    c.Category_Name,
    r.Region_Name,
    SUM(f.Sales_Amount) AS Total_Sales
```

```
FROM sales_fact f
JOIN product_dim p ON f.Product_ID = p.Product_ID
JOIN category_dim c ON p.Category_ID = c.Category_ID
JOIN region_dim r ON f.Region_ID = r.Region_ID
GROUP BY c.Category_Name, r.Region_Name;
```

---

3. Fact Constellation Schema

- A schema that contains multiple fact tables sharing common dimension tables.
- Also known as the **Galaxy Schema**.

Characteristics

1. Supports complex analytics involving multiple business processes.
2. More flexible than star or snowflake schemas.
3. Higher storage and query complexity.

- **Advantages**:
  - Supports multiple business processes.
- **Disadvantages**:
  - Increased complexity.

Example Schema

- **Fact Tables**: `sales_fact`, `inventory_fact`.
- **Shared Dimension Tables**: `product_dim`, `time_dim`, `store_dim`.

Diagram Representation

```
        product_dim
             |
sales_fact --- store_dim
```

```
            |
      inventory_fact
            |
         time_dim
```

SQL Example

```sql
-- Query sales and inventory data by product
SELECT
    p.Product_Name,
    SUM(s.Sales_Amount) AS Total_Sales,
    SUM(i.Stock_Quantity) AS Total_Stock
FROM sales_fact s
JOIN inventory_fact i ON s.Product_ID = i.Product_ID
JOIN product_dim p ON s.Product_ID = p.Product_ID
GROUP BY p.Product_Name;
```

4. Schema Selection Considerations

Schema	Advantages	Disadvantages
Star Schema	Simple, fast queries, intuitive design.	Redundant data increases storage.
Snowflake Schema	Saves storage, avoids redundancy.	Complex queries due to normalization.
Fact Constellation	Handles complex analytics.	High complexity, higher storage.

5. Advanced Querying with Analytical Functions

Use Case: Rank products by sales within each category.

SQL Example

Setup Schema in Hive

```
-- Fact table
CREATE TABLE sales_fact (
    Sales_ID INT,
    Product_ID INT,
    Store_ID INT,
    Sales_Amount FLOAT,
    Sales_Date DATE
);

-- Dimension table
CREATE TABLE product_dim (
    Product_ID INT,
    Product_Name STRING,
    Category STRING
);

-- Insert sample data
INSERT INTO sales_fact VALUES (1, 101, 1, 100.0, '2024-01-01');
INSERT INTO product_dim VALUES (101, 'Laptop', 'Electronics');
```

```
SELECT
    p.Category,
    p.Product_Name,
    SUM(f.Sales_Amount) AS Total_Sales,
    RANK() OVER (PARTITION BY p.Category ORDER BY SUM(f.Sales_Amount) DESC) AS Rank
FROM sales_fact f
JOIN product_dim p ON f.Product_ID = p.Product_ID
GROUP BY p.Category, p.Product_Name
ORDER BY p.Category, Rank;
```

Explanation

1. **PARTITION BY**: Divides rows into categories.
2. **ORDER BY**: Orders rows within each partition.
3. **RANK()**: Assigns rank based on sales.

ETL Process and Data Warehouse Optimization

1. Overview of ETL Process

ETL (Extract, Transform, Load) is a fundamental process in data warehousing to move and prepare data from source systems to the warehouse.

2. Phases of ETL Process

1. **Extract**
   - Extracts data from multiple source systems (e.g., RDBMS, NoSQL, APIs).
   - Data may be structured, semi-structured, or unstructured.

   Example Tools: Sqoop, Apache Nifi, AWS Glue, Talend.

   SQL Example (Extracting Data)

   ```
   -- Extract sales data from an RDBMS table
   SELECT * FROM sales_data;
   ```

2. **Transform**
   - Cleans, validates, and applies business rules to the extracted data.
   - Common operations: filtering, joining, aggregating, deduplication.

   Example Tools: Apache Spark, Python Pandas, AWS Glue.

   SQL Example (Transformation)

```sql
-- Clean and transform sales data
SELECT
    Product_ID,
    Store_ID,
    Sales_Amount * 1.1 AS Adjusted_Sales,
    TO_DATE(Sales_Date, 'YYYY-MM-DD') AS Formatted_Date
FROM sales_data
WHERE Sales_Amount > 0;
```

3. **Load**
    - Loads transformed data into a target data warehouse system.
    - Can be full or incremental.

    Example Tools: Hive, Amazon Redshift, Snowflake, BigQuery.

    SQL Example (Loading Data)

```sql
-- Insert transformed data into a data warehouse
INSERT INTO sales_fact
SELECT * FROM transformed_sales_data;
```

3. Practical ETL/ELT Workflow

Scenario: Load sales data from MySQL into Hive.

Steps

1. **Extract**: Use Sqoop to fetch data from MySQL into Hive Table.

```
sqoop import \
--connect jdbc:mysql://localhost/salesdb \
--username root --password password \
--table sales_data \
--target-dir /user/hduser/sales_data
```

```
CREATE EXTERNAL TABLE sales_data_raw (
Product_ID INT,
Store_ID INT,
Sales_Amount DOUBLE,
Sales_Date STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/user/hduser/sales_data';
```

2. **Transform**: Use Hive to clean and transform the data.

```
CREATE VIEW transformed_sales_data AS
SELECT
    Product_ID,
    Store_ID,
    Sales_Amount * 1.2 AS Adjusted_Sales,
    TO_DATE(Sales_Date, 'YYYY-MM-DD') AS Formatted_Date
FROM sales_data_raw
WHERE Sales_Amount > 0;
```

3. **Load**: Insert the transformed data into the fact table.

```
CREATE TABLE sales_fact(
Product_ID INT,
Store_ID INT,
Sales_Amount DOUBLE,
Sales_Date DATE
)
ROW FORMAT ORC;
```

```
INSERT INTO TABLE sales_fact
SELECT Product_ID, Store_ID, Sales_Amount, Sales_Date FROM transformed_sales_data
WHERE Sales_Date > (SELECT MAX(Sales_Date) FROM sales_fact)
```

4. Data Warehouse Optimization Techniques

1. **Partitioning**
   - Divides large tables into smaller, manageable pieces.
   - Improves query performance by reducing the data scanned.

   Example in Hive

```
CREATE TABLE sales_fact_partitioned (
    Sales_ID INT,
    Product_ID INT,
    Store_ID INT,
    Sales_Amount FLOAT
) PARTITIONED BY (Sales_Date DATE) ...;
```

2. **Bucketing**

- Groups data into fixed buckets based on a hash function.
- Reduces shuffle in map-reduce and improves join performance.

Example in Hive

```
CREATE TABLE sales_fact_bucketed (
    Sales_ID INT,
    Product_ID INT,
    Sales_Amount FLOAT
) CLUSTERED BY (Product_ID) INTO 4 BUCKETS ...;
```

3. **Indexing**

- Creates indexes on columns to speed up queries.
- Useful for frequently filtered columns.

SQL Example

```
CREATE INDEX sales_index ON TABLE sales_fact (Sales_Date);
```

4. **Materialized Views**

- Stores precomputed results of complex queries.
- Reduces runtime computation.

Example in Hive

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT Product_ID, SUM(Sales_Amount) AS Total_Sales
FROM sales_fact
GROUP BY Product_ID;
```

5. **Compression**
    - Compresses data to reduce storage costs and improve I/O performance.

    Example

```
SET hive.exec.compress.output=true;
SET hive.exec.compress.intermediate=true;
SET mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.SnappyCodec;
```

5. Query Optimization Techniques

1. **Predicate Pushdown**
    - Push filtering to the data source to reduce data transferred.
2. **Query Execution Plan**
    - Use `EXPLAIN` or `EXPLAIN EXTENDED` to understand query performance.

    Example

```
EXPLAIN SELECT * FROM sales_fact WHERE Sales_Amount > 1000;
```

3. **Avoid Small Files Problem**
    - Combine small files to reduce overhead.

    Example

```
SET hive.merge.smallfiles.avgsize = 256000000;
```

4. **Parallel Execution**
    - Enable parallel query execution for faster performance.

    Example

```
SET hive.exec.parallel=true;
```

Highlights

- A data warehouse is optimized for historical data storage and analytics.
- Modern data warehouses are increasingly cloud-based, scalable, and integrated with real-time tools.
- Data Lakes complement Data Warehouses in handling unstructured and raw data, while **Lakehouses** aim to combine their advantages.
- Cloud architectures are leading the way for scalable and cost-effective solutions.
- Tools like Snowflake, BigQuery, and Redshift dominate modern data warehousing solutions.
- ETL vs. ELT approaches depend on the processing environment and data volume.
- The ETL process ensures smooth data movement and transformation into the warehouse.
- Partitioning and bucketing significantly improve query performance by reducing data scans and optimizing joins.
- Indexing adds another layer of performance enhancement for lookups and range queries.
- Query optimization requires a combination of techniques like filtering, leveraging partitions, and avoiding unnecessary scans.
- Fact and dimension tables are critical to data warehousing and should be carefully designed for scalability and performance.
- Slowly Changing Dimensions (SCD) are essential for tracking historical changes in dimension data.
- Query optimization techniques, including materialized views, can greatly improve query performance.
- Fact and dimension tables are central to schema design and efficient querying.
- Data modeling is crucial for efficient querying and analytics in data warehouses.
- Choose between star, snowflake, or constellation based on business needs and scalability.
- Analytical functions enhance querying capabilities.

Cron

IO Redirection (Linux)

- output redirection: command > file
- output redirection append: command >> file
- error redirection: command 2> file
- input redirection: command < file
- output of first command as input for second command: command1 | command2
- output on terminal as well as in a file: command | tee file
- error redirection on stdout: 2>&1
    - terminal> ls /hom > out.txt 2>&1

CRON

- CRON is process scheduler for Linux.
- It can schedule process execution periodically or at fixed time.
    - Run a task: on 31-Dec-2022 00:00:00
    - Run a task daily: at 12:30 AM
    - Run a task on each Sunday: at 8.00 AM
    - Run a task monthly: on 10th date at 12:00 PM

Write CRON Job

- Install cron on your system.
    - terminal> sudo apt install cron
- Cron job are written in text file using command "crontab -e".
- On first execution, it asks for the editor.
    - Recommended: vim.basic
- Cron job are executed in background by "cron" service/daemon.
    - terminal> sudo systemctl status cron
    - terminal> ps aux | grep "cron"
- Cron expressions: https://crontab.guru/

- Cron example

```
*/2 *   * * *   /usr/bin/date >> /tmp/dates.log
5   0   * * *   /path/to/ingest.sh
```

- ingest.sh (example)

```
#!/bin/bash

export HADOOP_HOME=/path/of/hadoop
export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH

# download weather data from weather station website
/usr/bin/wget https://ncdc/todays -o /tmp/weather.txt

# upload into hive staging table (external table)
hadoop fs -put /tmp/weather.txt /user/nilesh/ncdc_staging
```

Apache Airflow

- A platform created by community to programmatically author, schedule and monitor workflow.
- Airflow is used to develop, orchestrate and monitor complex ETL pipelines.
- Airflow is completely developed in Python, so it can work with any of the Python library.
- Development
  - Initiated by Airbnb in 2014
  - Open-sourced under Apache in 2016.
  - Apache top level project in 2019.
- Used by more than 250 companies world-wide - Amazon, Citi, JPMorgan, Salesforce, Drillinginfo, Unitedhealth, ...
- Managed services in GCP and AWS cloud.

Traditional ETL

- Extract, Transform and Load
- Common schedulers: cron, oozie (hadoop), luigi (spotify)
- BigData has complex pipelines.
- Cron limitations
    - Error handling
    - Not maintainable (runs on a machine)
    - Execution dependency (delayed tasks)
    - Transperency (No centralized log)
    - Task tracking (No centralized monitoring)
    - Handle historical data
- Oozie
    - Error handling (n retries)
    - Execution dependency
    - Better tracking & monitoring
    - XML based jobs - difficult to read/understand

Airflow Applications

- Task scheduler
- ETL pipelines
- Periodic backups
- Automate DevOps operations
- ML jobs
- Recommendation engines