Hive Optimization Techniques

Athira George - 22011102010
Ananya Ganesam – 22011102018
Iot A

What is Hive?

Hive is a **data warehouse system** built on top of **Hadoop**.

Uses **Hive Query Language (HQL)** (similar to SQL).

Processes data using **MapReduce, Tez, or Spark**.

Why Use Hive?

- **Handles large datasets efficiently**.
- **Simplifies querying** using SQL-like syntax.
- **Integrates with Hadoop** (HDFS, MapReduce, Tez, Spark).

How Hive Works?

- **Stores data in HDFS** using various formats like ORC, Parquet, and Avro.
- **Uses Hive Query Language (HQL)** to run SQL-like queries.
- **Converts queries into execution plans** using MapReduce, Tez, or Spark.
- **Processes data in batches or in-memory**, depending on the execution engine.
- **Retrieves and returns structured results**, making it useful for analytics and reporting.

Hive vs. Traditional RDBMS

Feature	Hive	Traditional RDBMS
Query Language	HiveQL (SQL-like)	SQL
Storage	HDFS	Disk-based storage
Execution	Batch Processing (MapReduce, Tez, Spark)	Row-based transactions
Speed	Optimized for Big Data	Faster for small datasets
ACID Transactions	Limited (Supported in Hive 2.0+)	Fully Supported

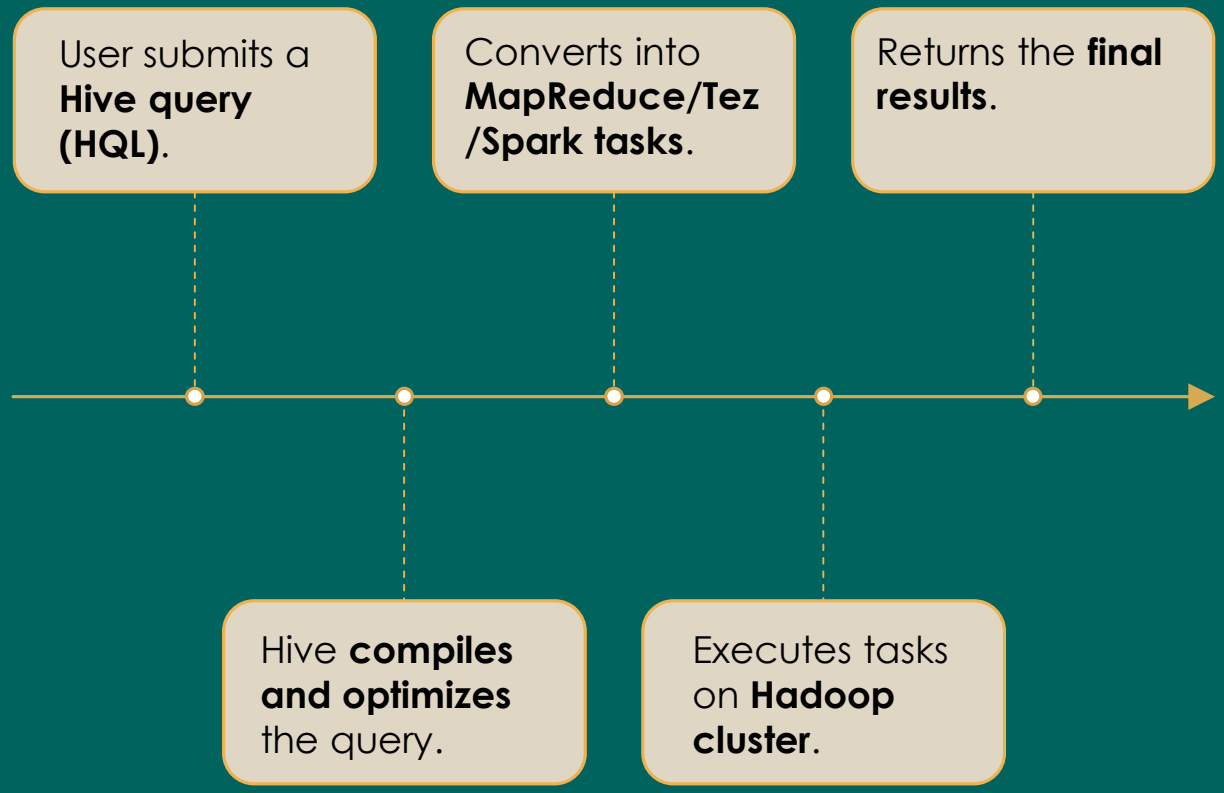
Key Takeaways:

- Hive is **not a replacement** for RDBMS.
- Best for **batch processing and analytical workloads**.
- **Not ideal** for real-time applications.

Hive Architecture Overview

Components of Hive:

- **Metastore** – Stores table metadata (Default: Derby, can use MySQL).
- **Execution Engine** – Converts queries into **MapReduce, Tez, or Spark jobs**.
- **Storage Layer** – Uses **HDFS** for distributed storage.



Hive Query Execution Flow

How Queries Are Processed in Hive:

- **Query Parsing:** Converts HQL into **Abstract Syntax Tree (AST)**.
- **Logical Plan Generation:** Breaks query into execution steps.
- **Optimization & Compilation:** Uses **Cost-Based Optimization (CBO)**.
- **Execution Plan Generation:** Converts logical plan into execution tasks.
- **Execution & Results Retrieval:** Runs the query and retrieves results.

Example Query Execution:

```
SELECT COUNT(*) FROM sales_data;
```

- **Simple SELECT query** runs without MapReduce.
- **Aggregations (e.g., COUNT)** trigger MapReduce jobs.

Why Does This Matter?

Understanding query execution helps in **query optimization**.

Query Optimization Techniques

Understanding Query Optimization:

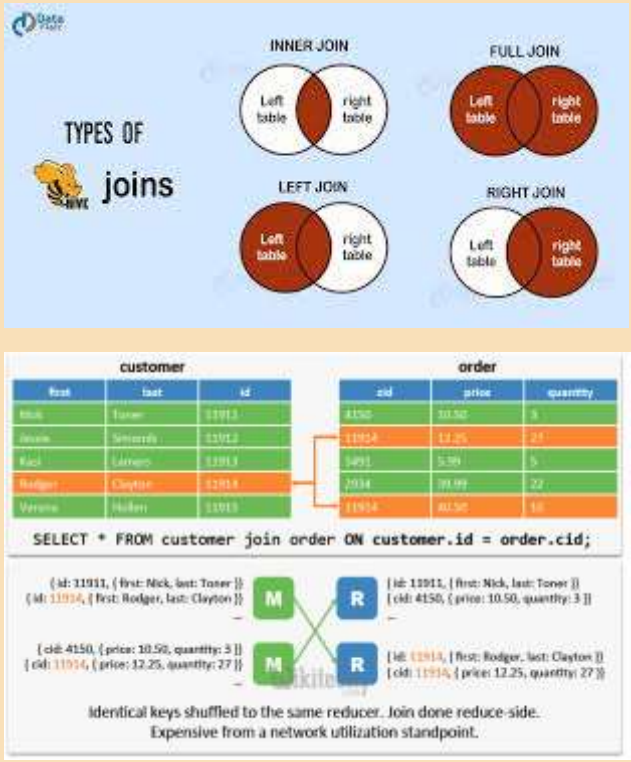
- Hive **automatically optimizes** queries.
- Manual tuning can **enhance performance**.
- **Cost-Based Optimization (CBO)** improves efficiency.

Optimization Strategies:

- **Predicate Pushdown:** Filters data early to reduce workload.
- **Column Pruning:** Select only necessary columns.
- **Reducing Shuffling:** Minimize data movement across nodes.
- **Parallel Execution:** Enable parallel processing.

Optimizing Joins in Hive

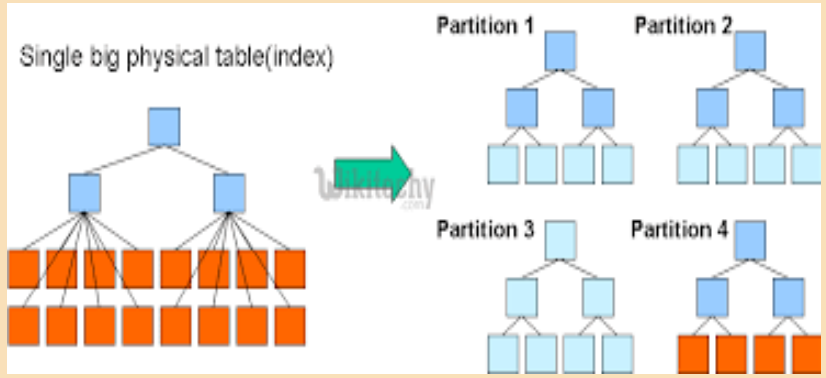
- **Types of Joins in Hive**
- **Inner Join:** Matches records in both tables.
- **Left/Right Join:** Includes records from one table even if no match exists.
- **Full Outer Join:** Returns all records from both tables.

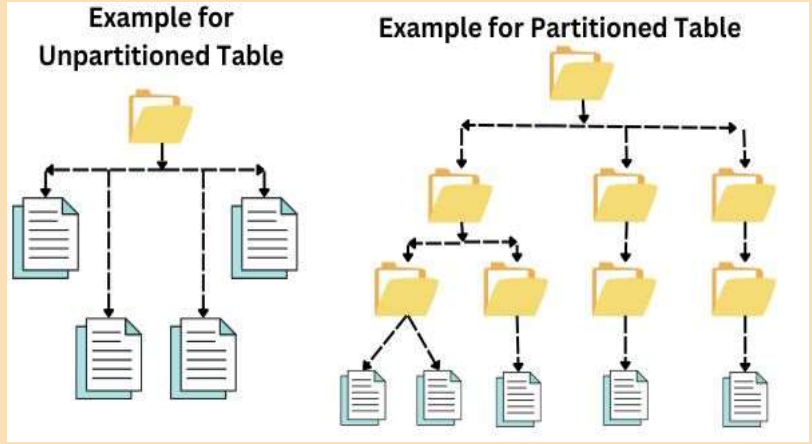


Efficient Join Strategies

1. **Map Join**
   (`hive.auto.convert.join=true`)
   s smaller tables into memory.
2. **Bucket Join**
   (`hive.enforce.bucketing=true`)
   Uses bucketing for efficient joins.
3. **Skew Join**
   (`hive.optimize.skewjoin=true`)
   Prevents uneven data distribution.

Partitioning in Hive





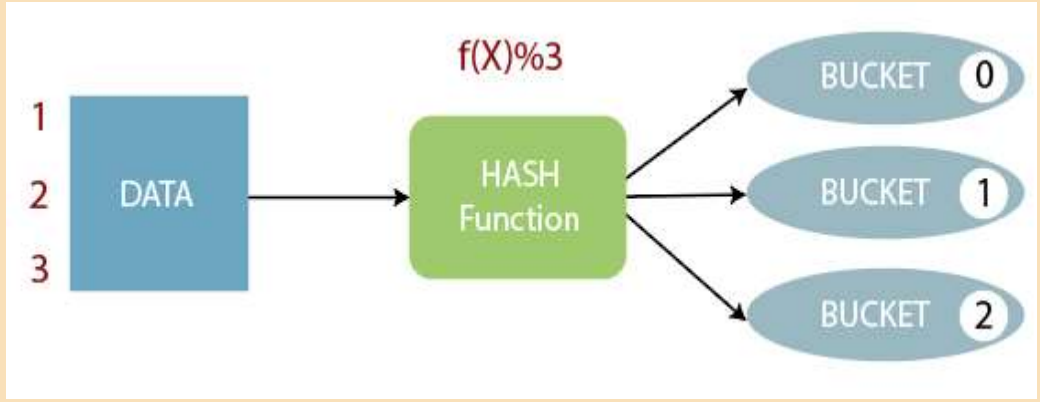
Why Partitioning?

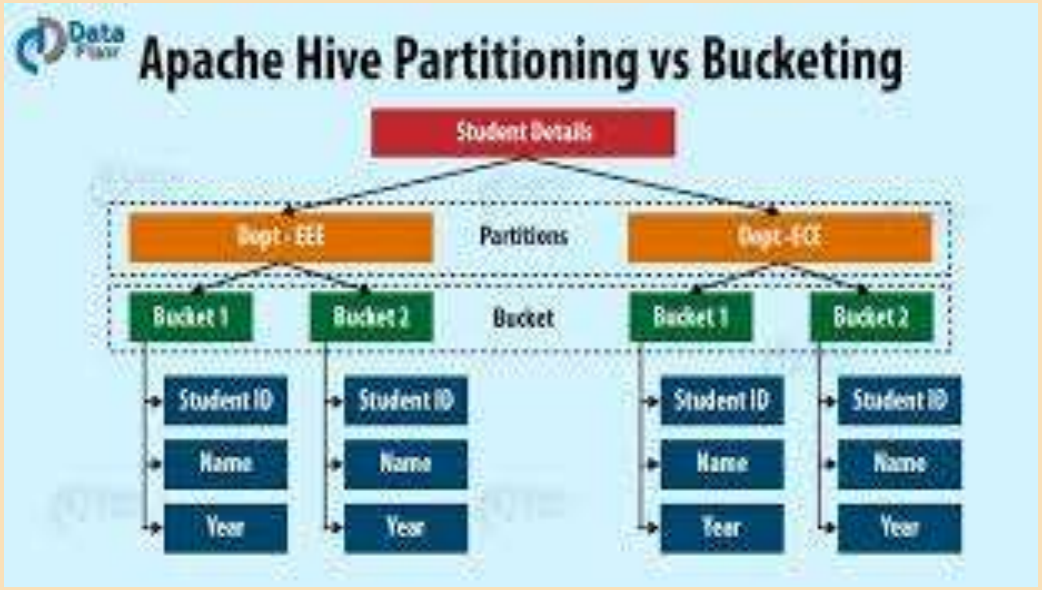
- Improves **query speed**.
- Reduces **unnecessary data scans**.

Types of Partitioning

- **Static Partitioning** – User manually defines partitions.
- **Dynamic Partitioning** – Partitions are created automatically during data insertion.

```
CREATE TABLE sales (sale_id INT, amount FLOAT)
PARTITIONED BY (country STRING);
```





Bucketing in Hive

Divides data into fixed-sized parts (buckets) based on a **hash function.**

Optimizes **joins** and **sampling.**

Each bucket is stored as a **separate file**

Why Use Bucketing?

- Speeds up **sampling and joins**.
- Organizes data into **fixed-sized buckets**.

```
H(column) MOD NumBuckets = BucketNumber
```

```
CREATE TABLE sales_buckets (
    sale_id INT,
    amount FLOAT
)
CLUSTERED BY
```

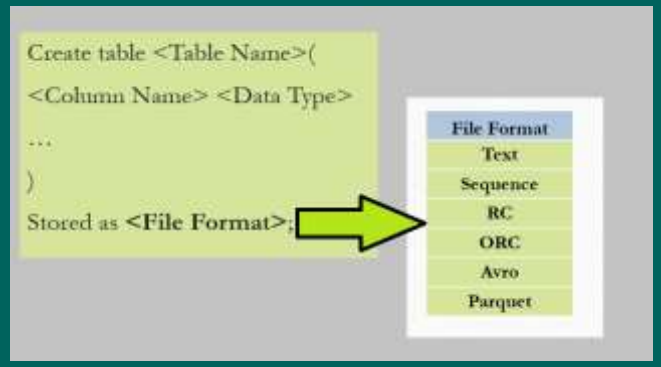
File Format Selection for Optimization

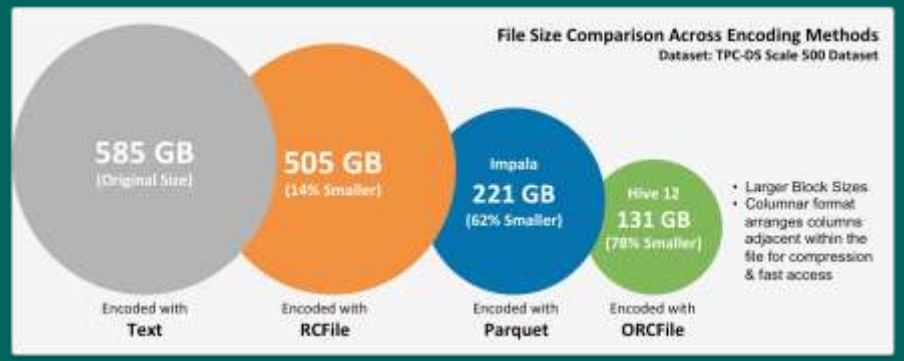
Best File Formats:

- **ORC (Optimized Row Columnar)** – Best for Hive, supports indexing.
- **Parquet** – Efficient columnar storage.
- **Avro** – Schema evolution support.

Example:

```
CREATE TABLE sales_orc (sale_id INT, amount FLOAT)
STORED AS ORC;
```





Types of indexing in Hive

Compact indexing

- Stores row IDs (RID) and helps speed up query performance.
- Uses a separate table to store index metadata.

Bitmap Index

- Efficient for columns with low cardinality (fewer distinct values).
- Uses bitmaps to represent column values, reducing storage and lookup time.

```
SET hive.optimize.index.filter=true;
```

Improves query performance by allowing Hive to automatically use the index.

Creating a compact index

```
CREATE INDEX idx_example_compact
ON TABLE sales (sale_id)
AS 'COMPACT';
```

Creates a compact index on the `sale_id` column of the `sales` table to optimize space and query performance.

Creating a bitmap index

```
CREATE INDEX idx_example_bitmap
ON TABLE sales (country)
AS 'BITMAP';
```

Creates a bitmap index on the `country` column of the `sales` table to improve query performance for low cardinality data.

Small Files Problem & Solution

Why Small Files Slow Down Queries

- **High Overhead**: In Hadoop, each file incurs overhead in terms of storage and computation. The larger the number of files, the more overhead there is, leading to performance degradation.
- **Increased Metadata**: Each small file requires metadata management in the Hadoop Distributed File System (HDFS), which can add significant latency when dealing with a large number of files.
- **Task Parallelism Limits**: When queries are executed, each small file often results in an individual map task, which can lead to task overload and inefficient resource usage. This reduces the parallelism of the job.
- **Low Data Locality**: Small files may not fully utilize the available data locality, causing the system to read data from multiple nodes, which impacts query speed.

Solution: Combining Small Files

- **CombineHiveInputFormat**: This is a solution to the small file problem in Hive. It allows for the combination of many small files into fewer larger ones before the actual query execution.

  How it Works:

  - It groups multiple small files into fewer, larger splits.
  - This minimizes the overhead and improves parallelism by reducing the number of map tasks.
  - It helps Hive leverage Hadoop's distributed processing more effectively by reducing metadata management and improving data locality.

- **Benefits**:
  - **Reduced Job Overhead**: Fewer map tasks are required, resulting in lower job overhead.
  - **Improved Query Performance**: The system performs better by handling fewer, larger files rather than many small ones.
  - **Better Resource Utilization**: By reducing the number of tasks, resources are more efficiently allocated across the cluster.

Parallel Execution & Resource Tuning

Enabling Parallel Execution

- By default, Hive queries execute sequentially, but enabling parallel execution can improve performance.
- **Configuration:**
  - Set `hive.exec.parallel=true` to allow multiple stages of a query to run in parallel.
  - Useful for large datasets and complex queries with multiple stages (e.g., joins, aggregations).
  - Consider data skew and cluster resource availability before enabling.

2. Tuning Number of Mappers & Reducers

- The number of **mappers and reducers** directly impacts query performance.
- **Reducers:**
  - Defined by `hive.exec.reducers.bytes.per.reducer`.
  - Too few reducers → Bottleneck in execution.
  - Too many reducers → Excessive overhead and under-utilization of resources.
  - Use `set mapreduce.job.reduces=<num>` to manually specify reducer count if needed.
- **Mappers:**
  - Determined by input splits (HDFS block size).
  - Use larger files or adjust mapreduce.input.fileinputformat.split.minsize to optimize.

Adjusting Memory Allocation

- Proper memory tuning prevents **out-of-memory (OOM) errors** and improves job efficiency.

- Key configurations:
  - **Map Task Memory:** `mapreduce.map.memory.mb` (default: 1024MB)
  - **Reduce Task Memory:** `mapreduce.reduce.memory.mb` (default: 2048MB)
  - **Heap Size:** `mapreduce.map.java.opts=-Xmx<size>m` (typically 80% of map memory)
  - Increase values based on **dataset size** and **available cluster resources**.
  - Monitor memory usage using **YARN Resource Manager**.

Data Skew

occurs when **data is unevenly distributed** across reducers, causing **some reducers to handle significantly more data than others**

Common causes

- Highly repetitive values in a column (e.g., NULLs, common categories, etc.)
- Uneven data partitioning in GROUP BY, JOIN, or ORDER BY operations
- Small number of keys assigned to few reducers
- Large variations in dataset size per partition

Handling skew

```
SET hive.groupby.skewindata=true;
```

- When set to `true`, Hive **automatically detects skewed keys**.
- Instead of sending all data with the same key to a single reducer, Hive **splits the data into multiple reducers**.
- After initial processing, another round of **aggregation combines partial results**, preventing reducer overload.

ACID transactions in Hive

Hive supports **ACID transactions** starting from **Hive 2.0+**,enabling **INSERT, UPDATE, DELETE, and MERGE** operations on tables.

Atomicity – Ensures transactions are fully completed or fully rolled back.

Consistency – Maintains integrity constraints and correct state of data.

Isolation – Prevents concurrent transactions from interfering with each other

Durability – Ensures committed transactions remain permanent.

Transactions **only work with ORC file format** and **transactional tables**:

```
CREATE TABLE employee (
    emp_id INT,
    name STRING,
    department STRING,
    salary DOUBLE
)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');
```

`transactional=true` ensures the table supports ACID properties.

Solution: Combining Small Files

- **CombineHiveInputFormat**: This is a solution to the small file problem in Hive. It allows for the combination of many small files into fewer larger ones before the actual query execution.

  How it Works:

  - It groups multiple small files into fewer, larger splits.
  - This minimizes the overhead and improves parallelism by reducing the number of map tasks.
  - It helps Hive leverage Hadoop's distributed processing more effectively by reducing metadata management and improving data locality.

- **Benefits**:

  - **Reduced Job Overhead**: Fewer map tasks are required, resulting in lower job overhead.
  - **Improved Query Performance**: The system performs better by handling fewer, larger files rather than many small ones.
  - **Better Resource Utilization**: By reducing the number of tasks, resources are more efficiently allocated across the cluster.

insert

```sql
INSERT INTO employee VALUES (1, 'Alice', 'HR', 50000);
INSERT INTO employee VALUES (2, 'Bob', 'IT', 70000);
```

update

```sql
UPDATE employee
SET salary = 75000
WHERE emp_id = 2;
```

delete

```sql
DELETE FROM employee WHERE emp_id = 1;
```

Tez and Spark

Tez is an advanced **DAG-based execution engine** designed to optimize Hive queries.
It minimizes **disk I/O, improves task parallelism**, and reduces job execution time.

Spark is a **fast, in-memory** processing engine that can replace MapReduce for Hive queries.
It **processes data in memory**, reducing latency for **iterative** queries and complex analytics.

```
SET hive.execution.engine=tez;
```

```
SET hive.execution.engine=spark;
```

Why Use Tez & Spark for Hive Execution?

- Hive originally used **MapReduce**, which had high **latency and disk I/O overhead**.
- **Tez and Spark** provide **faster query execution** by **reducing disk writes, optimizing DAG execution, and improving parallel processing**.
- Switching to **Tez or Spark** enhances **performance, scalability, and resource utilization** in Hive.

Benefits of Hive on Tez

- **Faster execution** – Tez eliminates unnecessary disk writes between stages.
  In-memory processing – Reduces data movement and improves performance.

- **Optimized DAG processing** – Queries execute in a streamlined fashion.
  Better resource utilization – Efficiently manages YARN resources.

Benefits of Hive on Spark

- **In-Memory Computation** – Spark processes data in RAM, reducing disk I/O.
- **Faster for Complex Queries** – Performs well on **joins, aggregations, and analytics**.
- **Scalability** – Handles **large datasets efficiently** using Spark's distributed computing.
- **Supports ML & Graph Processing** – Integrates with **MLlib and GraphX** for advanced analytics.

Materialized Views & Caching

What is Query Caching?

- Hive **caches query results** to **reuse them for repeated executions**, reducing unnecessary computation.
- This is useful for **dashboards, reporting, and frequent analytical queries**.

```
SET hive.cache.expr.evaluation=true;
SET hive.exec.reducers.bytes.per.reducer=256000000;
SET hive.vectorized.execution.enabled=true;
```

- **What Are Materialized Views?**
- A **materialized view (MV)** stores **precomputed query results** as a physical table.
- When the same query runs again, **Hive retrieves the results from the MV instead of recomputing**.
- This is useful for **aggregations, joins, and complex analytical queries**.

```
SELECT * FROM mv_sales_summary;
```

```
CREATE MATERIALIZED VIEW mv_sales_summary
AS
SELECT region, SUM(sales) AS total_sales
FROM sales_data
GROUP BY region;
```

Why Use Materialized Views & Caching?

1.Repeated queries on large datasets can be **slow and resource-intensive**.

2.Query caching and materialized views help **improve performance** by avoiding redundant computations.

3.This leads to **faster response times and optimized resource utilization**.

Benefits of Materialized Views & Caching

Faster Query Execution – Avoids recalculating results for repeated queries.
Reduced Computation Overhead – Saves CPU and memory usage for complex aggregations.
Optimized Resource Utilization – Minimizes redundant processing, improving cluster efficiency.

Hive vs. Other Big Data Query Engines

Feature	Hive	Spark SQL	Presto	Impala
Latency	High (Slow)	Low (Fast)	Low (Fast)	Very Low (Fastest)
ACID Support	✅ Yes (Hive 2.0+)	❌ No	❌ No	✅ Yes
File Format Support	ORC, Parquet	Parquet, Avro	ORC, Parquet	Parquet
Execution Mode	Batch Processing	Interactive	Interactive	Interactive
Best For	Large-scale ETL & batch analytics	Real-time analytics & ML	Ad-hoc, federated queries	Low-latency BI reporting
Engine Type	MapReduce / Tez / Spark	In-Memory Processing	Distributed Query Engine	MPP (Massively Parallel Processing)
Query Performance	Slower due to job scheduling	Faster with in-memory computing	Optimized for federated queries	Fastest for BI and OLAP workloads
Use Cases	Data Warehousing, Batch Processing	Data Science, Streaming Analytics	Querying multiple databases	BI Dashboards, Real-time Analytics

Thank You