# Limitations of Hive Engine and strategies to overcome them

In the last lecture, we learned about a few of the optimizations in Hive, in this lecture, we are going to deep dive into a few more of them and understand even after these optimizations why other technologies are replacing Hive slowly.

## Partition Pruning

### What is Partition Pruning?

Partition pruning allows Hive to skip unnecessary partitions when running a query. If a table is partitioned (e.g., by date or region), Hive only reads the relevant partitions instead of scanning the entire dataset. This reduces the amount of data read and processed, speeding up query execution.

### Detailed Example:

Suppose we have a table `sales` partitioned by `region` and `order_date`:

```
CREATE TABLE sales (
  order_id INT,
  amount DOUBLE
)
PARTITIONED BY (region STRING, order_date STRING);
```

If you query this table with conditions on the `region` or `order_date`, Hive will automatically prune partitions that don't match the query condition. Consider this query:

```
SELECT order_id, amount
FROM sales
WHERE region = 'US' AND order_date = '2024-01-01';
```

- **Without partition pruning**: Hive would scan all partitions (possibly across multiple regions and dates), leading to unnecessary data processing.

- **With partition pruning**: Hive will only scan the partition where `region = 'US'` and `order_date = '2024-01-01'`. All other partitions will be ignored, drastically improving performance.

**Additional Example:**

If you have a year's worth of sales data partitioned by `order_date`, a query for a specific day like:

```
SELECT COUNT(*)
FROM sales
WHERE order_date = '2023-11-01';
```

Hive would scan only the partition for `2023-11-01`, skipping the rest of the year's data.

**Best Practices:**

- Always use partition columns in `WHERE` clauses to benefit from partition pruning.
- Partition by columns that are frequently used in queries, like `date` or `region`.

## 2. Predicate Pushdown

**What is Predicate Pushdown?**

Predicate pushdown allows Hive to filter data at the storage layer (e.g., in ORC or Parquet formats) before reading it into the query engine. This reduces the volume of data read into Hive, minimizing I/O and speeding up query execution.

**Detailed Example:**

Suppose we have a table `orders` stored as ORC:

```
CREATE TABLE orders (
  order_id INT,
  customer_id INT,
  amount DOUBLE,
  status STRING
)
STORED AS ORC;
```

If you run this query:

```
SELECT order_id, amount
FROM orders
WHERE status = 'completed';
```

- **Without predicate pushdown**: Hive would read all rows from the ORC file and then apply the filter on `status` after loading the data into memory.
- **With predicate pushdown**: The filter `status = 'completed'` is pushed down to the storage layer. Only rows where `status = 'completed'` are read from disk, reducing the amount of data transferred and processed.

**Multiple Examples:**
**Numeric Filters**:

```
SELECT order_id, amount
FROM orders
WHERE amount > 1000;
```

1. With predicate pushdown, Hive will only read rows where `amount > 1000`, reducing the size of data pulled from disk.

**Date Filters**:

```
SELECT order_id
FROM orders
WHERE order_date = '2023-01-15';
```

2. Hive will only read rows with `order_date = '2023-01-15'` from the ORC files, skipping irrelevant data.

**Best Practices:**

- Use columnar storage formats like ORC or Parquet to enable predicate pushdown.
- Always specify conditions in `WHERE` clauses to take advantage of this optimization.

## 3. Dynamic Partitioning

**What is Dynamic Partitioning?**

Dynamic partitioning allows Hive to automatically determine the partition values for incoming data based on the data itself. This is particularly useful when inserting data from an unpartitioned source into a partitioned table.

**Detailed Example:**

Consider an unpartitioned table `unpartitioned_sales` and a partitioned table `sales_partitioned`:

```
CREATE TABLE sales_partitioned (
  order_id INT,
  amount DOUBLE
)
PARTITIONED BY (region STRING, order_date STRING);
```

You want to insert data into the partitioned table based on the values of `region` and `order_date` from the unpartitioned source. With dynamic partitioning, Hive will figure out which partitions to insert into without explicitly specifying them:

```
INSERT INTO TABLE sales_partitioned PARTITION (region, order_date)
SELECT order_id, amount, region, order_date
FROM unpartitioned_sales;
```

- **Without dynamic partitioning**: You would have to manually specify the partition values for each insert, which is cumbersome for large datasets.
- **With dynamic partitioning**: Hive automatically determines the partition values based on the data in the `region` and `order_date` columns.

**Multiple Examples:**
**Dynamic Partitioning with Dates**:

```
INSERT INTO TABLE sales_partitioned PARTITION (region, order_date)
SELECT order_id, amount, region, order_date
FROM daily_sales;
```

1. Hive will create partitions dynamically based on the `order_date` field.

**Dynamic Partitioning with Regions**:

```
INSERT INTO TABLE sales_partitioned PARTITION (region, order_date)
```

```
SELECT order_id, amount, region, order_date
FROM region_sales;
```

2. Here, Hive partitions the data based on `region` and `order_date` dynamically.

**Best Practices:**

- Use dynamic partitioning when loading data from unpartitioned sources.
- Ensure that partition columns are part of the SELECT clause when inserting into a partitioned table.

## 4. Compression Techniques

**What is Compression?**

Hive can use compression to reduce the size of intermediate data and the final output, which saves storage space and reduces I/O during query execution.

**Detailed Example:**

To enable compression for intermediate data (shuffle) and output files, you can use the following settings:

```
SET hive.exec.compress.intermediate=true;  -- Compress intermediate
data
SET mapreduce.map.output.compress=true;    -- Compress MapReduce
intermediate output
SET
mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.Snap
pyCodec;  -- Use Snappy compression
```

To compress final output files:

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

**Multiple Examples:**

**MapReduce with Compression**: When running a MapReduce query that generates large amounts of shuffle data, enabling compression significantly reduces the amount of data transferred between mappers and reducers:

```
SET hive.exec.compress.intermediate=true;
SET mapreduce.map.output.compress=true;
```

1.

**Output Compression**: For a query that writes large outputs to HDFS:

```
SET hive.exec.compress.output=true;
SET mapreduce.output.fileoutputformat.compress=true;
```

2. Compressed output files will be smaller, reducing disk usage and speeding up subsequent queries that read this data.

**Best Practices:**

- Use compression for large datasets, especially when dealing with heavy shuffle or large outputs.
- Snappy is often a good balance between compression ratio and speed.

## 5. Auto Reduce Parallelism

### What is Auto Reduce Parallelism?

Auto reduce parallelism dynamically adjusts the number of reducers based on the size of the data. Hive uses a configuration to estimate how much data each reducer should handle and adjusts the number of reducers accordingly.

### Detailed Example:

Let's assume you have a large dataset, and Hive is assigning too few reducers, causing them to become overloaded. You can tune this with the following settings:

```
SET hive.exec.reducers.bytes.per.reducer=67108864;  -- 64 MB per
reducer
SET hive.exec.reducers.max=1000;  -- Maximum 1000 reducers
```

This ensures Hive uses the optimal number of reducers based on the data size. For example, if your dataset is 1 GB and you set `64 MB per reducer`, Hive will automatically use approximately 16 reducers.

**Multiple Examples:**
**Optimizing Large Joins**: If you have a query with a large join:

```
SELECT o.order_id, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

1. By adjusting `hive.exec.reducers.bytes.per.reducer`, Hive will dynamically allocate more reducers for the join, improving performance.
2. **Skewed Data with Large Reducers**: When dealing with skewed data, adjusting the reducer size can help balance the workload across reducers, preventing some reducers from being overwhelmed while others finish early.

**Best Practices:**

- Start with a reasonable size for `hive.exec.reducers.bytes.per.reducer`, like 64 MB or 128 MB, and adjust based on the dataset size.
- Set a maximum number of reducers to prevent excessive parallelism, which can lead to overhead.

## 6. Avoid Cartesian Joins

**What is a Cartesian Join?**

A Cartesian join (or cross join) occurs when there is no explicit join condition between tables, leading to a full combination of all rows from both tables. This results in an explosion of the dataset size, which can severely degrade performance.

**Detailed Example:**

Consider two tables, `customers` and `orders`. If you accidentally run a query without specifying a join condition:

```
SELECT *
FROM customers, orders;
```

Hive will generate a Cartesian product of all rows in `customers` and `orders`, which could be disastrous for large datasets.

**How to Prevent Cartesian Joins:**

To avoid Cartesian joins, always include an explicit `JOIN` condition. For example:

```
SELECT c.customer_name, o.order_id
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

You can also prevent Cartesian joins by enabling strict mode in Hive:

```
SET hive.mapred.mode=nonstrict;  -- Default mode allows cartesian
joins
SET hive.mapred.mode=strict;  -- Prevents queries with no join
condition
```

**Multiple Examples:**
**Correct Join**:
[1]

```
SELECT c.customer_name, o.amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

1. This avoids a Cartesian product and joins only matching rows.

**Accidental Cartesian Join**:

```
SELECT *
FROM orders o, customers c;
```

2. This creates a Cartesian product and should be avoided, especially for large datasets.

**Best Practices:**

- Always use explicit join conditions.
- Enable strict mode to prevent accidental Cartesian joins.

---

[1]

As we move forward, in subsequent lectures, we are going to learn one of the most prominent batch data technologies i.e Spark. It necessary to know why we the industry is moving towards Spark and what Hive lag which Spark does better.

## Hive vs Spark: Why Hive is Replaced by Spark

To understand why Spark has overtaken Hive, we need to explore their historical background, architectural differences, performance aspects, and overall functionality.

## 1. Historical Context: Evolution of Hive and Spark

**Apache Hive:**

Apache Hive was developed at Facebook to handle large-scale data stored in the Hadoop Distributed File System (HDFS). Its main goal was to simplify querying large datasets using a familiar SQL-like language known as HiveQL. Before Hive, most processing on Hadoop was done using MapReduce, a low-level programming model that required custom Java code. Hive abstracted the complexity of MapReduce by allowing users to write SQL queries instead.

Hive uses MapReduce as its execution engine by default, though it later added support for Tez and Spark as execution engines. Over time, Hive became the go-to solution for batch processing on Hadoop, enabling users to run long-running analytical queries on petabytes of data.

**Apache Spark:**

Spark emerged from the University of California, Berkeley, in response to the limitations of MapReduce. It was designed to address the inefficiencies of MapReduce by introducing an in-memory distributed computing framework. Spark's primary goal was to support faster, iterative computations that are common in machine learning and data science. Unlike Hive, which was designed for batch processing, Spark was designed to support real-time and streaming data processing as well.

With its powerful APIs and libraries (Spark SQL, Spark Streaming, MLlib, GraphX), Spark rapidly became a unified engine for both batch and real-time data processing, eclipsing Hadoop's traditional MapReduce paradigm.

## 2. Hive’s MapReduce Bottleneck

At its core, Hive relies heavily on MapReduce for query execution, which presents several key limitations:

**a. Disk I/O Overhead:**

MapReduce is inherently a disk-based system, which means intermediate data between the map and reduce stages is written to disk and then read again. This causes significant I/O overhead, especially for complex queries that involve multiple stages (e.g., joins, aggregations). As a result, MapReduce is relatively slow for tasks requiring iterative processing, where the same dataset is repeatedly accessed, transformed, and written.

For instance, running a query like:

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

In Hive with MapReduce would involve multiple phases where intermediate results are written to HDFS, causing a performance bottleneck due to disk I/O.

**b. High Latency:**

MapReduce's disk-heavy architecture results in high latency, making Hive unsuitable for real-time processing or interactive querying. Even for simple analytical queries, users might experience long waiting times, especially for large datasets. For data pipelines that require immediate insights or quick turnaround, this is a significant drawback.

**c. Multiple Job Initiation Overhead:**

MapReduce initiates a new job for each stage of the query (e.g., a join followed by an aggregation), which involves launching a new JVM (Java Virtual Machine) for each job. The overhead associated with starting and stopping JVMs adds to the overall execution time, further reducing efficiency.

## 3. Spark’s In-Memory Computing Advantage

In contrast, Spark was designed from the ground up to address the inefficiencies of MapReduce. The biggest differentiator between Hive and Spark is **in-memory computation**.

**a. In-Memory Processing:**

Spark processes data in memory wherever possible, significantly reducing the need for disk I/O. Once data is loaded into memory, Spark can perform multiple transformations without having to write intermediate results to disk, which leads to dramatic speed improvements.

For example, iterative algorithms (such as those used in machine learning or graph processing) can be executed orders of magnitude faster in Spark than in Hive because the same data is reused multiple times within memory.

A query like:

sql
Copy code

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

In Spark would read the data into memory, perform all necessary transformations, and only write the final result back to disk. This approach drastically reduces execution time, especially for iterative tasks.

**b. Directed Acyclic Graph (DAG) Execution:**

While Hive uses the traditional MapReduce model, which involves distinct map and reduce phases, Spark employs a Directed Acyclic Graph (DAG) for execution. A DAG allows for more flexible, optimized execution plans where multiple stages of a query can be processed in parallel or pipelined. This reduces the need for multiple shuffle operations and disk writes.

In Spark, tasks are scheduled and optimized based on dependencies within the DAG, leading to faster and more efficient query execution.

**c. Unified Engine for Batch and Real-Time Processing:**

Hive was built for batch processing and later adapted for real-time streaming with the introduction of execution engines like Tez and Spark. However, these were add-ons to an inherently batch-oriented system. On the other hand, Spark was designed from the beginning to handle both batch and real-time data.

With **Structured Streaming**, Spark provides a high-level API for real-time data processing, enabling users to process streams of data using the same DataFrame

and SQL APIs as batch queries. This makes Spark far more versatile than Hive in modern data engineering pipelines, where both real-time and batch processing are needed.

## 4. Performance Comparisons

The performance differences between Hive and Spark can be stark, especially for large datasets and complex queries:

**a. Query Speed:**

Spark can outperform Hive by 10x to 100x for certain queries, particularly those involving iterative algorithms or real-time data processing. This is largely due to Spark's in-memory capabilities and efficient DAG-based execution model.

For example, machine learning workflows that involve iterative algorithms like k-means clustering or linear regression can be unbearably slow in Hive due to its reliance on MapReduce, which continuously writes data to disk between iterations. In contrast, Spark performs these operations in memory, allowing them to complete much faster.

**b. Resource Utilization:**

Spark is more efficient in terms of resource utilization because it avoids the repetitive disk I/O operations required by MapReduce. This results in lower memory and CPU usage for the same workload. In resource-constrained environments, Spark's ability to complete jobs faster while using fewer resources is a critical advantage.

**c. Concurrency:**

Spark handles concurrency better than Hive. Since Hive jobs rely on MapReduce, which has a high overhead for launching and managing jobs, it struggles with concurrency, especially in a multi-user environment. Spark, with its faster job execution and better resource management, supports a higher degree of parallelism, making it more suitable for shared cluster environments.

## 5. Advanced APIs and Ecosystem Integration

While Hive is primarily a SQL query engine, Spark offers a rich ecosystem of APIs for various data processing needs:

**a. Spark SQL:**

Spark SQL allows users to run SQL queries on Spark's distributed data, similar to HiveQL, but with the added benefit of in-memory execution. Spark SQL is also integrated with Spark's other libraries, making it easy to combine SQL queries with machine learning pipelines or graph analytics.

**b. MLlib for Machine Learning:**

Spark includes **MLlib**, a machine learning library that supports algorithms like classification, clustering, and recommendation. Hive lacks built-in support for machine learning, making Spark a more versatile tool for data scientists and engineers working with big data.

**c. GraphX for Graph Processing:**

Spark provides **GraphX** for graph processing, which Hive doesn't support. This allows Spark to perform complex graph computations like PageRank or shortest-path algorithms, making it more suitable for social network analysis, recommendation engines, and similar applications.

## 6. Conclusion: Why Spark Replaces Hive

The shift from Hive to Spark is driven by performance, flexibility, and scalability. Hive, while once a powerful tool for querying massive datasets, is limited by its reliance on MapReduce, which leads to high latency, disk I/O bottlenecks, and slow query execution.

Spark addresses these issues by offering in-memory processing, a DAG execution model, and a unified engine for batch, real-time, and iterative computations. Its ability to handle both SQL queries and advanced analytics (machine learning, graph processing, etc.) makes it a more versatile tool in modern data engineering.

In summary, Spark's superior performance, advanced ecosystem, and real-time capabilities make it the preferred choice over Hive for big data processing in today's fast-paced, data-driven world.

In Summary

| Limitation | Description | Strategy to Overcome |
|---|---|---|

| | | |
|---|---|---|
| **Partition Pruning** | Reads only relevant partitions based on query conditions, avoiding full dataset scans. | Use partition columns in `WHERE` clauses; partition by frequently queried columns like date or region. |
| **Predicate Pushdown** | Filters data at storage layer (e.g., ORC, Parquet) before reading, reducing I/O and speeding up execution. | Use columnar storage formats; specify filter conditions in `WHERE` clauses. |
| **Dynamic Partitioning** | Automatically determines partition values during data insertion, useful for loading data into partitioned tables. | Enable dynamic partitioning; include partition columns in the `SELECT` clause for partitioned inserts. |
| **Compression Techniques** | Reduces intermediate and output data size to save storage and minimize I/O. | Use compression settings like Snappy for shuffle and output data compression in MapReduce queries. |
| **Auto Reduce Parallelism** | Dynamically adjusts number of reducers based on data size, preventing overloaded reducers and balancing the workload. | Set appropriate values for `hive.exec.reducers.bytes.per.reducer` and a maximum reducer count to balance performance. |
| **Avoid Cartesian Joins** | Prevents unintended cross joins, which can create a massive data explosion and slow performance. | Use explicit join conditions; enable strict mode to avoid accidental Cartesian joins. |

| | | |
|---|---|---|
| **MapReduce Disk I/O Overhead** | MapReduce-based execution writes intermediate data to disk, causing significant I/O and slowing down complex queries. | Use Tez or Spark execution engines for Hive; or migrate to Spark for in-memory processing. |
| **High Latency and Job Overheads** | High startup latency and multiple job initiations add to execution time, especially in large, iterative queries. | Leverage Spark's DAG-based execution model to minimize multiple stages and reduce startup time. |
| **Limited Real-Time and Advanced Processing** | Hive is batch-oriented and lacks support for real-time, iterative computations common in data science and machine learning. | Use Spark for in-memory and streaming capabilities, supporting both batch and real-time processing needs. |