Writing Hive queries effectively - Make Hive queries 2x faster

While working as a Data Engineer, you will encounter a lot of issues regarding slow Hive performance, and it's necessary for you to be ware of these issues and how to solve them. I have collated the most common issues that have been encountered and are bound to happen in the Data engineering world. Let's go over them one by one.

Inefficient Joins Due to Large Data Skew

Data skew refers to an imbalanced distribution of data across partitions or nodes in a distributed system. In Hive, when performing a join operation, the system divides data into smaller chunks, typically called partitions, and assigns them to different nodes or reducers to be processed in parallel. If the data is not evenly distributed, some reducers end up with much more data than others, creating a **bottleneck** that slows down the overall query execution.

This issue commonly arises when joining tables with **skewed keys**, where certain values occur significantly more frequently than others. For example, imagine you have a table of customer orders, and 90% of your data is associated with a single customer. If a join is performed on the customer ID, one reducer will handle the majority of the data, while the rest remain underutilized, leading to inefficient parallelism.

Example Scenario of Data Skew

Consider two tables:

- `orders` table with millions of records, each associated with a customer ID.
- `customers` table with information about the customers, with a join on the `customer_id`.

If one customer (e.g., customer ID = 12345) makes an extraordinarily large number of orders compared to others, a join operation that includes `customer_id` will assign the majority of this customer's records to a single reducer. Other reducers will receive much less data, leading to an uneven distribution of work.

```
SELECT o.order_id, c.customer_name
```

```
FROM orders o
JOIN customers c
ON o.customer_id = c.customer_id;
```

In the above query, if `customer_id = 12345` has, say, 90% of the total orders, one reducer will be overloaded, delaying the entire query completion as other reducers will finish their tasks early.

Symptoms of Data Skew in Joins

- **Long query execution times**: A few reducers take significantly longer to finish compared to others.
- **Inefficient resource usage**: Some reducers may be idle while others are overwhelmed with data.
- **Memory errors**: Reducers handling a large chunk of skewed data might face out-of-memory (OOM) errors.

How to Detect Data Skew

1. **Query Execution Plans**: You can inspect the execution plan of your query to see how data is partitioned across reducers. If one or a few reducers handle a disproportionately large amount of data, it's a sign of data skew.
2. **Monitoring Logs**: Logs from Hadoop or YARN (if you're using it as the execution engine) will show skewed reducer loads.
3. **Skewed Key Analysis**: Analyze your data distribution before the join operation to identify highly frequent key values that may lead to skew.

How to Fix Data Skew in Hive Joins

1. Skew Join Optimization

Hive provides a configuration to handle data skew automatically when performing joins. This is known as **Skew Join Optimization**. When enabled, Hive detects skewed keys and splits them into smaller partitions, distributing the processing more evenly across reducers.

How it works: If a particular key (e.g., customer ID) is responsible for a large number of records, Hive will break that key's records into smaller chunks and assign them to multiple reducers. This avoids overloading a single reducer.

Steps to enable Skew Join Optimization:

```
SET hive.optimize.skewjoin=true;
```

This configuration instructs Hive to monitor the data distribution during the map phase. If skew is detected, the join is broken down into two phases:

- Phase 1: The skewed key is handled separately.
- Phase 2: The remaining non-skewed data is processed in the regular manner.

2. **Sampling Skewed Data**

Another approach is to **sample the data** to identify skewed keys. By running a preliminary query to find out which keys have disproportionately large numbers of records, you can either manually handle those keys (e.g., by running special queries for them) or adjust the query execution strategy.

Example:

```
SELECT customer_id, COUNT(*)

FROM orders

GROUP BY customer_id

ORDER BY COUNT(*) DESC

LIMIT 10;
```

This query identifies the top 10 customers responsible for the most orders. Once identified, you can treat these keys differently by distributing their data more evenly.

Small Files Problem in HDFS

What is the Small Files Problem?

In Hadoop's Distributed File System (HDFS), the **small files problem** refers to the inefficiency that arises when too many small files (files smaller than HDFS block size, typically 128MB or 256MB) are stored and processed in the system. While HDFS is designed for handling large-scale data, it struggles with the management of numerous small files because of its architecture, which optimizes for large, sequential data blocks rather than many tiny files.

Why Are Small Files a Problem in HDFS?

1. **NameNode Memory Overload:** HDFS stores metadata about the file system in the **NameNode**—the master node that keeps track of the file hierarchy and where blocks are located. Each file in HDFS (whether large or small) is represented by an entry in the NameNode's memory. If there are millions of small files, the NameNode's memory can be overwhelmed by the sheer number of file metadata entries, which can cause performance degradation or even out-of-memory (OOM) errors.
2. **Increased I/O Overhead:** HDFS is designed to work efficiently with large files that are split into multiple blocks (each block typically 128MB or larger). However, when small files are stored, the I/O operations required to retrieve and process these files become inefficient. Since each file (no matter its size) is handled as a separate block, accessing and processing millions of small files leads to excessive disk seeks and slower data processing.
3. **Inefficient MapReduce Jobs:** MapReduce, the traditional execution engine for Hadoop, assigns one input split to each mapper. Small files cause inefficiency because each mapper processes a small amount of data, leading to **too many map tasks**, each doing minimal work. This results in more overhead in terms of starting, managing, and coordinating mappers rather than actual data processing.
4. **Network Congestion:** The small files problem also creates **network congestion**, as retrieving multiple small files requires many separate read operations across the network, instead of fetching larger, contiguous blocks of data. This leads to increased network traffic and delays in processing.

Symptoms of the Small Files Problem

- **Longer query execution times** due to inefficient file access patterns.
- **Overloaded NameNode memory** when working with a large number of files, leading to memory issues or slower performance.
- **Increased task overhead** in MapReduce or other distributed execution engines, where many mappers perform very little work on each small file.
- **High disk I/O and network utilization**, as the system constantly switches between files, increasing seek times and network requests.

Common Causes of Small Files

1. **Data Ingestion from External Sources:** Data being ingested from multiple sources like sensors, logs, or external systems may come in as small files. For example, log files might be generated every minute, leading to a large number of small files over time.
2. **Improper File Splitting:** During data extraction or transformation, files may be split into smaller chunks without merging or aggregation. This can happen if data is not batched properly or if the storage format is inefficiently chosen.
3. **Frequent Data Dumping:** Systems that generate frequent incremental data dumps (e.g., every second or minute) create many small files instead of larger, consolidated ones.

Solutions to the Small Files Problem

1. Combine Small Files into Larger Files

The most effective solution is to combine multiple small files into larger files. This reduces the number of files HDFS needs to manage and allows more efficient processing.

Using Hive's File Merging Capabilities Hive provides built-in mechanisms to combine small files during query execution. You can enable these configurations to merge small files at different stages of execution, like during map-reduce or at the output phase.

Settings for File Merging:

```
SET hive.merge.mapfiles=true;          -- Merge small files at the map
stage
SET hive.merge.mapredfiles=true;       -- Merge small files at the
reduce stage
SET hive.merge.size.per.task=256000000; -- Set merged file size (e.g.,
256MB)
SET hive.merge.smallfiles.avgsize=128000000; -- Set average file size
threshold
```

In these settings:

- `hive.merge.mapfiles=true`: This setting ensures that small files generated at the map phase are merged.
- `hive.merge.mapredfiles=true`: This merges small files at the reduce phase.

- `hive.merge.size.per.task` allows you to control the size of the merged files, and the `hive.merge.smallfiles.avgsize` threshold triggers the merge when the average file size is smaller than the defined limit.

2. Use Efficient File Formats (ORC/Parquet)

Storing data in **columnar formats** like **ORC (Optimized Row Columnar)** or **Parquet** significantly reduces the number of small files because these formats are highly compressed and optimized for batch processing. Unlike text formats (CSV, JSON), columnar formats store data more compactly, making it easier to combine files and reducing the number of files created during data processing.

How to Create ORC/Parquet Tables:

```
CREATE TABLE my_table (

  id INT,

  name STRING,

  sales DOUBLE

)

STORED AS ORC;
```

ORC and Parquet tables are also better optimized for query performance, with features like **predicate pushdown**, **compression**, and **splitting** that help in efficiently handling large datasets.

Broadcast Join

In a distributed computing environment like Hadoop or Spark, a **join** is an operation that combines two datasets based on a common key. In Hive or Spark, join operations can be resource-intensive because they require shuffling data across nodes to ensure that matching keys are brought together for processing.

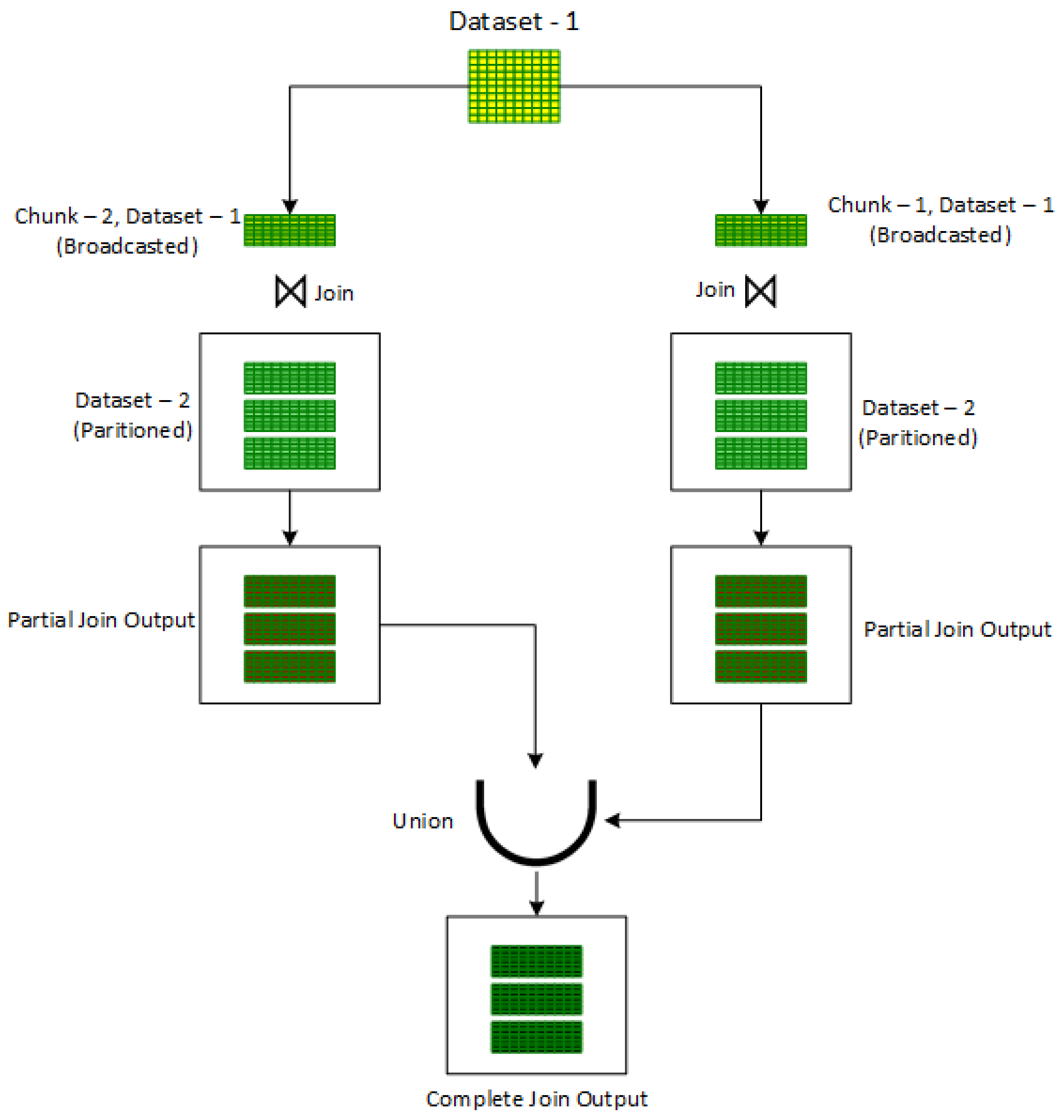
In the context of Hive, Spark, or Hadoop, there are two primary types of joins:

1. **Shuffle Join**: Both datasets are partitioned across multiple nodes, and data with matching keys is brought together by shuffling data between nodes.
2. **Broadcast Join**: A more efficient alternative for cases where one of the datasets is small enough to fit into memory, enabling the small dataset to be broadcasted to all nodes.

What is a Broadcast Join?

A **broadcast join** (also called a **map-side join** in Hive) is a join strategy where a smaller dataset is distributed (or "broadcast") to all worker nodes, avoiding the costly shuffle operation that occurs in traditional joins. Instead of moving large chunks of data between nodes (as in a shuffle join), the smaller dataset is copied to all nodes in the cluster, where it is joined with partitions of the larger dataset locally. This allows the join to happen entirely on the mapper side (hence the term "map-side join"), bypassing the reduce phase.

This approach is highly efficient when the smaller dataset can fit in memory because it reduces the need for network I/O and eliminates the shuffling of large amounts of data.



When to Use a Broadcast Join

Broadcast joins are most effective when:

1. **One of the tables is much smaller** than the other. The smaller table can fit into memory, and its data can be sent to all worker nodes.
2. **Reducing network traffic** is a priority. By broadcasting the smaller table, there is no need to shuffle large amounts of data between nodes.
3. **Map-side join optimizations** are available and can be leveraged.

In systems like Hive, Spark, or even Presto, you can optimize query performance by broadcasting smaller tables in joins when applicable.

How a Broadcast Join Works

The key idea is to replicate the smaller table (or dataset) across all nodes. Each node then performs a local join with its partition of the larger dataset, without any need for data movement between nodes.

Here's a high-level breakdown of how a broadcast join works:

1. **Small Dataset Identification**: The system identifies the smaller dataset in the join. This dataset must be small enough to fit in the memory of the worker nodes.
2. **Broadcasting**: The smaller dataset is sent (or broadcast) to all worker nodes across the cluster.
3. **Local Join Execution**: Each worker node has a partition of the larger dataset. With the smaller dataset already in memory, the node performs the join locally between its partition and the broadcasted dataset.
4. **Result Gathering**: The results from each worker node are collected, with no further shuffling or communication required.

This process avoids the costly data movement and shuffling required by a traditional reduce-side join, where both datasets are partitioned, and matching keys are shuffled across nodes.

Example in Hive

Let's say we have two tables:

- `customers` (small table): Contains customer information with a few thousand records.
- `orders` (large table): Contains millions of orders, each associated with a customer ID.

Typically, joining these tables would require data from both to be shuffled between nodes:

```
SELECT o.order_id, c.customer_name
```

```
FROM orders o

JOIN customers c

ON o.customer_id = c.customer_id;
```

If `customers` is small, Hive can broadcast it to all nodes to perform a broadcast join. By enabling **map-side joins**, Hive will automatically broadcast the smaller table.

Enabling a Broadcast Join in Hive:

```
SET hive.auto.convert.join=true;
```

With this setting, Hive detects that `customers` is smaller than a certain threshold and will broadcast it, performing a map-side join rather than a reduce-side shuffle join.

Configurations for Broadcast Join in Hive:

`hive.auto.convert.join.noconditionaltask.size`: This configuration controls the threshold for the size of the smaller table (in bytes) for Hive to automatically broadcast it. If the smaller table is below this size, Hive will use a broadcast join.
Example:
```
SET hive.auto.convert.join.noconditionaltask.size=10000000;  -- 10MB
```

- In this case, if `customers` is smaller than 10MB, Hive will broadcast it for the join.

Overloaded Reducers Causing Bottlenecks in Hive

What Happens When Reducers are Overloaded?

When a reducer is overloaded, it has to handle an excessive amount of data. As a result, the following issues occur:

1. **Memory Overflows (OOM Errors)**: An overloaded reducer may consume more memory than it has been allocated, leading to **Out of Memory (OOM)** errors. These errors can cause the entire query or job to fail.
2. **Prolonged Execution Times**: If a few reducers receive more data than others, they take much longer to complete their tasks, creating a bottleneck. Since

the job can't finish until all reducers complete their tasks, even reducers that finish early remain idle while waiting for the overloaded reducers.

3. **Imbalanced Resource Utilization**: When some reducers are overloaded while others remain underutilized, the parallel processing capability of the system is not used efficiently. This can result in wasted resources, including CPU, memory, and disk I/O.
4. **High Disk I/O and Network Traffic**: An overloaded reducer may require large amounts of temporary storage (spilling to disk), leading to high disk I/O. Additionally, large amounts of data may need to be shuffled across the network to reach the reducers, leading to network congestion.

How to Detect Overloaded Reducers?

1. **Logs and Metrics**: Check Hive and Hadoop logs for signs of **data skew**, such as OOM errors or unusually long reduce task times. Monitoring systems like the **YARN Resource Manager UI**, **Hadoop JobTracker**, or tools like **Ganglia** or **Prometheus** can provide insights into how the workload is distributed across reducers. If some reducers are taking significantly longer than others, it's a sign of data skew or overloaded reducers.
2. **Execution Plans**: Review the execution plan for the query (available through `EXPLAIN`) to identify whether there's an uneven distribution of work across reducers. If the number of reducers is too small for the amount of data, this may indicate a configuration issue.
3. **Look at the Shuffle Data Size**: If a query involves a large amount of shuffle data (data transferred between the mappers and reducers), it can overload reducers. Monitoring the shuffle size for each reducer can help identify the problem.

Solutions to Overloaded Reducers

1. Adjust the Number of Reducers

The number of reducers Hive uses is determined by the amount of data being processed and the `hive.exec.reducers.bytes.per.reducer` setting. This setting specifies how many bytes of data should be processed by each reducer. If this value is too large, there will be fewer reducers, leading to some reducers being overloaded with too much data.

How to adjust the reducer size:

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

This configuration ensures that each reducer will handle approximately 64MB of data. If the data size is large, more reducers will be created, balancing the load more effectively.

You can also explicitly set the number of reducers using the following configuration:

```
SET mapreduce.job.reduces=100;  -- Set 100 reducers
```

However, setting the number of reducers manually should be done with caution. Too few reducers can cause overloading, while too many can lead to overhead without performance benefits.

2. Enable Data Skew Handling in Joins

When data skew is the cause of overloaded reducers, you can enable **skew join optimization** in Hive. This setting automatically detects when certain keys are skewed (i.e., they account for an uneven amount of data) and processes those keys separately to avoid overloading reducers.

How to enable skew join optimization:

```
SET hive.optimize.skewjoin=true;
```

When this is enabled, Hive will split the processing of skewed keys into smaller tasks, distributing them across multiple reducers to ensure that no single reducer is overwhelmed.

3. Use Partitioning and Bucketing

Partitioning and bucketing are techniques used to ensure that data is distributed evenly across reducers, reducing the chances of overloading.

Partitioning: Breaks down large datasets into smaller, more manageable pieces based on certain columns. This can help minimize the amount of data each reducer processes by only processing relevant partitions.

Example: If you frequently query by date, partition your table by the `date` column.

```
CREATE TABLE partitioned_orders (

    order_id INT,

    customer_id INT,

    amount DOUBLE

)

PARTITIONED BY (order_date STRING)

STORED AS PARQUET;
```

Partitioning ensures that only relevant portions of the table are processed by each reducer, thus reducing load.

Bucketing: Distributes data into a fixed number of "buckets" based on a specific column, ensuring that reducers handle approximately the same amount of data.

Example of bucketing:

```
CREATE TABLE bucketed_orders (

    order_id INT,

    customer_id INT,

    amount DOUBLE

)

CLUSTERED BY (customer_id) INTO 32 BUCKETS

STORED AS ORC;
```

Bucketing ensures that data is split more evenly, reducing the chances that a reducer will be overloaded.

4. Increase Memory for Reducers

If the reducers are overloaded because of insufficient memory allocation, you can increase the memory available for each reducer task. This helps the reducers handle larger chunks of data without spilling to disk or running out of memory.

For MapReduce:

```
SET mapreduce.reduce.memory.mb=8192;      -- 8GB memory for reducers

SET mapreduce.reduce.java.opts=-Xmx6144m; -- 6GB heap space for
reducer JVM
```

For Tez:

```
SET tez.task.resource.memory.mb=8192;      -- 8GB memory for Tez tasks

SET tez.task.java.opts=-Xmx6144m;          -- 6GB heap space for Tez
task JVM
```

Increasing the memory reduces the likelihood of reducers failing due to memory overflows and helps improve the performance of memory-intensive operations.