Table Storage Optimizations

In Apache Hive, efficient data management and query performance are crucial for handling large-scale data in a distributed system. Hive provides several techniques to optimize table creation and querying, including partitioning, bucketing, and other optimizations. Understanding these techniques will help you manage and query your data more effectively.

Hive Sessions

I know this is a bit off the track, but to understand upcoming concepts, we must understand what a Hive session is. A session in Hive refers to the duration of a user's interaction with the Hive server, starting when the user connects to the Hive service and ending when the user disconnects or the session is explicitly terminated. Each session is a context within which the user can execute queries and commands and interact with the Hive database. This may not make much sense to you, but it will as move forward with further concepts.

Hive Table Partitioning

Partitioning is a method of dividing a large table into smaller, more manageable pieces based on the values of one or more columns. This helps improve query performance by reducing the amount of data that needs to be scanned during query execution.

How Partitioning Works

- **Partition Columns:** Data is divided based on the values of partition columns. Each unique value of the partition column creates a separate partition.
- **Partition Directory Structure:** In HDFS, partitions are organized into subdirectories. For example, a partitioned table on year and month will have a directory structure like /year=2024/month=08/.

Creating a Partitioned Table

```
CREATE TABLE sales_data (
  product_id INT,
  sales_amount FLOAT
)
PARTITIONED BY (year INT, month INT);
```

Adding Partitions:

You can manually add partitions to a table using the ALTER TABLE command:

ALTER TABLE sales_data ADD PARTITION (year=2024, month=08) LOCATION '/user/hive/warehouse/sales_data/year=2024/month=08/';

Automatic Partition Management:

Hive can automatically manage partitions if data is loaded using the LOAD DATA command and the partition columns are specified:

LOAD DATA INPATH '/path/to/data' INTO TABLE sales_data PARTITION (year=2024, month=08);

Querying Partitioned Tables

When querying partitioned tables, Hive can skip non-relevant partitions, improving query performance:

*SELECT * FROM sales_data WHERE year=2024 AND month=08;*

Where is all this information stored i.e partitions, its location, table metadata like column, etc?

Correct, its Hive MetaStore

Hive Table Bucketing

Bucketing in Hive is a technique used to optimize query performance by dividing a dataset into more manageable parts, called buckets. Each bucket contains a subset of the data, and this organization helps improve query efficiency, especially when dealing with large datasets. Bucketing is particularly useful when data is frequently queried based on a specific column (called the bucketing column), and you need faster access to rows.

How Bucketing Works

- **Bucket Column:** Data is hashed based on the bucket column, and the resulting hash value determines which bucket the data belongs to.

- **Number of Buckets:** The number of buckets is predefined and specified when creating the table.

Let's consider a table named `CustomerOrders` that stores millions of records, including customer IDs, order details, and transaction amounts. A typical query on this table might involve filtering data by `CustomerID`, such as:

```
SELECT * FROM CustomerOrders WHERE CustomerID = 12345;
```

If the table has millions of rows, this query might take a long time because Hive needs to scan all the data files to find rows with `CustomerID = 12345`. In this case, using **bucketing** can improve query performance.

How Bucketing Works

Bucketing helps in partitioning the data further at the file level by storing rows with the same bucketing column values into specific buckets. When Hive knows that data is already pre-organized into these buckets, it can skip scanning unnecessary data files and focus on the relevant bucket(s).

Creating a Bucketed Table

```
CREATE TABLE IF NOT EXISTS
CustomerOrders ( OrderID INT, CustomerID INT, OrderDate STRING,
OrderAmount DECIMAL(10, 2), Status STRING ) CLUSTERED BY (CustomerID)
INTO 10 BUCKETS ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS
TEXTFILE;
```

The `CLUSTERED BY (CustomerID)` clause defines that the `CustomerID` is the bucketing column.
The `INTO 10 BUCKETS` specifies that the table will be divided into 10 buckets based on the `CustomerID`.

Loading Data into Buckets:

When loading data, Hive will distribute rows into buckets based on the hash of the bucket column:

LOAD DATA INPATH '/path/to/customer_data INTO TABLE employee_data;

Advantages of Bucketing:

- **Optimized Query Performance**: Queries filtering by `CustomerID` will be faster since Hive will only scan the bucket(s) that contain the relevant data.
- **More Manageable Data Files**: Instead of storing all data in a single large file per partition, bucketing divides the data into smaller, more manageable files within each partition.
- **Efficient Joins**: When you bucket two tables on the same column (e.g., `CustomerID`), Hive can optimize joins between them because it knows that corresponding bucket values will be in the same bucket. This is known as a "bucket join," and it prevents unnecessary shuffling of data across nodes in the clusters.

Querying Bucketed Tables

```
SELECT * FROM CustomerOrders WHERE CustomerID = 1001;
```

Since Hive knows that `CustomerID = 1001` corresponds to a specific bucket, it will only scan that bucket, improving performance.

Bucketed Table Join: If you have another table, say `CustomerDetails`, also bucketed by `CustomerID`, you can perform efficient joins between `CustomerOrders` and `CustomerDetails`. For example:

```
SELECT o.OrderID, d.CustomerName

FROM CustomerOrders o

JOIN CustomerDetails d

ON o.CustomerID = d.CustomerID;
```

Since both tables are bucketed by `CustomerID`, Hive will perform the join bucket-by-bucket, avoiding a full table scan or costly shuffling across nodes.

SET Commands

Set commands can be treated as a special statement that tells Hive to tune up some settings. Now these settings are applied for a Hive Session, instead of a query. I hope you all remember what Hive Session is right. So once these settings are applied, it will be applied to all the queries executed thereafter, unless you change the settings or kill the session. Killing the session means you disconnect from HiveServer.

Setting a Configuration Property
SET <property_name>=<value>;
Retrieving the Value of a Specific Property
SET <property_name>;

Use Cases for SET in Hive:

1. **Optimizing Query Performance:**
    - Tuning memory settings, enabling/disabling vectorized execution, and configuring parallelism.
2. **Controlling Query Execution:**
    - Adjusting dynamic partitioning, setting the execution engine, or managing query caching.
3. **Customizing Logging and Debugging:**
    - Setting log levels, enabling debug mode, or configuring error handling.
4. **Defining and Using Variables:**
    - Creating reusable variables for use in multiple queries or scripts.

Now let's take a look at one of these special settings to understand how that setting changes the query execution behavior.

SET hive.exec.dynamic.partition=true; SET hive.exec.dynamic.partition.mode=nonstrict;

1. **SET hive.exec.dynamic.partition=true;**
    a. Purpose: This command enables dynamic partitioning in Hive. Dynamic partitioning allows partitions to be created on the fly during an INSERT operation based on the values in the data being inserted.
    b. When to Use: You use this setting when you want Hive to automatically determine and create partitions for your data as it is being loaded. This is particularly useful when the partition values are not known beforehand or when loading data into a table with many possible partition values.
2. **SET hive.exec.dynamic.partition.mode=nonstrict;**
    a. Purpose: This command sets the dynamic partitioning mode to nonstrict. In nonstrict mode, Hive allows the creation of dynamic partitions even when some partition columns are not explicitly specified in the INSERT statement.
    b. When to Use: You use this setting when you want Hive to dynamically create partitions without requiring that all partition columns be

specified. This allows greater flexibility when loading data into partitioned tables.

c. Modes Explained:
    i. Strict Mode (default): Hive requires at least one static partition column to be specified in the INSERT statement. This is to prevent the accidental creation of a large number of partitions, which could overwhelm the system.
    ii. Nonstrict Mode: Hive does not require any static partition columns to be specified. This mode is useful when you want Hive to determine all partition values dynamically.

Example:

SET hive.exec.dynamic.partition=true;

SET hive.exec.dynamic.partition.mode=nonstrict;

INSERT INTO TABLE sales_partitioned

PARTITION (year, month)

SELECT product_id, product_name, sale_amount, sale_date, year(sale_date) as year, month(sale_date) as month FROM sales;

In this example:

- Both the year and month partitions are dynamically created based on the data in the sale_date column.
- Since the mode is nonstrict, you are not required to specify any static partition value, allowing Hive to fully manage the partition creation.

We are going to look at more settings like this in upcoming lectures.

File Formats

There are multiple file formats supported in Hive

1. TextFile (Plain Text)

- **Description:** The default storage format in Hive. It stores data as plain text files where each line corresponds to a row in the table, and fields are separated by a delimiter (usually a comma or tab).
- **Pros:**
    - Human-readable, making it easy to inspect and debug.
    - Compatible with most tools and systems.
    - Simple to implement.
- **Cons:**
    - Inefficient storage and slower processing due to lack of compression.
    - Requires more storage space compared to binary formats.
- **Use Case:** Simple datasets, prototyping, or when interoperability with other systems is a priority.

Example:

```
CREATE TABLE sales_txt (

  product_id INT,

  product_name STRING,

  sale_date STRING,

  sale_amount FLOAT

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

STORED AS TEXTFILE;
```

2. SequenceFile

- **Description:** A binary file format that stores data in key-value pairs. It's a flat file consisting of binary key/value pairs and is splittable, making it efficient for MapReduce processing.
- **Pros:**
    - Splittable, allowing for parallel processing.
    - Compressible, leading to efficient storage.
    - Suitable for large datasets.
- **Cons:**
    - Not human-readable.

    - Requires more complex handling for keys and values.
- **Use Case:** Large datasets where efficient storage and processing are needed, particularly in MapReduce jobs.

Example:

```
CREATE TABLE sales_seq (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS SEQUENCEFILE;
```

3. ORC (Optimized Row Columnar)

- **Description:** A highly optimized columnar storage format designed specifically for Hive. ORC files offer efficient storage, fast processing, and strong compression capabilities.
- **Pros:**
    - High compression ratio, reducing storage needs.
    - Optimized for read performance, especially for complex queries.
    - Supports ACID operations and advanced features like predicate pushdown.
- **Cons:**
    - Slightly slower write performance due to overhead of optimization.
    - Not as widely compatible outside of Hadoop ecosystem.
- **Use Case:** Large datasets where query performance and storage efficiency are critical, such as in data warehousing environments.

Example:

```
CREATE TABLE sales_orc (
  product_id INT,
```

```
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS ORC;
```

4. Parquet

- **Description:** A columnar storage format that is language-agnostic and optimized for performance and efficiency, particularly for analytical queries. It is similar to ORC but with broader compatibility outside of Hive.
- **Pros:**
  - High compression and efficient storage.
  - Well-suited for queries that read only a few columns.
  - Broad support in the Hadoop ecosystem and beyond (e.g., Spark, Impala).
- **Cons:**
  - May have slower write times due to columnar format processing.
  - Less optimized for full row reads compared to row-based formats.
- **Use Case:** Analytical workloads, especially in environments that use multiple tools like Spark, Hive, and Impala.

Example:

```
CREATE TABLE sales_parquet (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS PARQUET;
```

5. Avro

- **Description:** A row-based storage format that is highly efficient and supports schema evolution. Avro files include a schema definition, making it easy to work with evolving data structures.
- **Pros:**
    - Supports schema evolution (changing schemas without breaking existing data).
    - Compact and efficient for both storage and processing.
    - Integrates well with various data serialization systems.
- **Cons:**
    - Requires handling of schema evolution complexities.
    - Less efficient for columnar queries compared to ORC or Parquet.
- **Use Case:** Datasets where schema may evolve over time or where interoperability with other systems (e.g., Kafka, Flume) is important.

Example:

```
CREATE TABLE sales_avro (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS AVRO;
```

6. JSON

- **Description:** Stores data as JSON (JavaScript Object Notation) text files. JSON is widely used for data exchange and is human-readable.
- **Pros:**
    - Human-readable and easy to work with.
    - Flexible and supports nested structures.
    - Compatible with many systems and programming languages.
- **Cons:**

    - Inefficient storage and slower processing compared to binary formats.
    - No built-in schema, which can lead to inconsistencies.
- **Use Case:** Datasets where human readability, interoperability, or nested structures are important, such as logs or web data.

Example:

```
CREATE TABLE sales_json (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE;
```

7. CSV (Comma-Separated Values)

- **Description:** A type of text file where each line represents a row, and fields are separated by commas. It's a simple and widely-used format.
- **Pros:**
    - Simple and human-readable.
    - Widely supported by many tools and systems.
    - Easy to generate and parse.
- **Cons:**
    - No support for complex data types or nested structures.
    - Larger file size and slower processing compared to binary formats.
- **Use Case:** Simple datasets, data exchange between different systems, or scenarios where human readability is important.

Example:

```
CREATE TABLE sales_csv (
```

```
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
```

- STORED AS TEXTFILE;