**Hive**

1. **Difference between Data warehouse and database?**

A database is a structured collection of data organized for efficient storage and retrieval, often used for transactional or operational purposes. It's designed for managing day-to-day operations and supports real-time data processing.

A data warehouse is a centralized repository that stores and integrates data from various sources for the purpose of analysis and reporting. It's optimized for querying and analysing historical and aggregated data to support business intelligence and decision-making.

1. **Difference between Data warehouse and data mart?**

A data warehouse is a centralized repository that stores data from various sources for analytical reporting and data analysis. It typically contains historical data and supports enterprise-wide decision-making.

A data mart is a subset of a data warehouse, focused on specific business areas or departments. It contains a more limited set of data tailored to the needs of a particular group, making it easier for them to access and analyse data relevant to their functions. Data marts are often derived from a data warehouse.

1. **Why is hive metadata stored in SQL?**

Hive metadata is stored in SQL to provide a structured and efficient way to manage and query large-scale data stored in the Hadoop ecosystem. Storing metadata in SQL allows users to interact with Hive using familiar SQL-like queries and tools, making it easier to manage, access, and analyse data in a distributed and scalable environment like Hadoop. It also facilitates integration with other SQL-based systems and tools, simplifying data management and access for users.

1. **Which SQL is the default database for hive?**

Apache Hive, a data warehousing and SQL-like query language tool for Hadoop, originally used Apache Hadoop Distributed File System (HDFS) as its default storage and the **Apache Derby** database as its default metadata store. However, in more recent versions of Hive (starting from Hive 3.0.0), the default metadata store has been changed to Apache HBase.

So, the default database for Hive, in terms of its metadata store, is Apache HBase as of Hive 3.0.0 and later versions. It's important to note that you can configure Hive to use other databases for metadata storage if needed.

1. **What is managed table?**

A managed table is a type of table that is managed by Hive itself. When you create a managed table in Hive, Hive is responsible for managing both the table's metadata and its data. This means that Hive handles the storage of the table's data in its default warehouse directory (usually in HDFS) and maintains the table's schema and other metadata information in the Hive Meta store.

Key points about managed tables in Hive.

**Data Storage:** Hive manages the data storage for managed tables, and the data is stored in a location controlled by Hive.

**Metadata:** Hive maintains metadata about the table, such as its schema (columns and data types), partitioning information, and other table properties, in the Hive Meta store database.

**Data Management:** Hive provides features for data management, including data loading, insertion, and deletion.

**Data Durability:** Data in managed tables is durable and is protected from accidental data loss.

**Backup and Recovery:** Hive typically includes support for backup and recovery of managed tables, making it easier to recover data in case of failures.

Managed tables are suitable for use cases where you want Hive to have full control over the data and metadata, and you are not concerned with managing the data files or locations manually. However, it's important to note that managed tables may not be as flexible as external tables, which allow you to have more control over the data files and their locations.

In contrast, there are also external tables in Hive, where you have more control over the data and the data storage locations, but you need to manage the data and metadata yourself.

1. **What is external table?**

An external table in Hive is a type of table that allows you to define a schema for data without moving or copying the data into Hive's default storage location (typically HDFS). Instead, an external table references data that exists outside of Hive, often in a location managed by the user or another system. Key points about external tables in Hive:

**Data Location:** External tables do not manage the storage of the data. The data files are stored in an external location that is not controlled by Hive. This can be a directory in HDFS, a remote file system, or any other location accessible to Hive.

**Metadata Management:** Hive manages the metadata for external tables, including the schema (columns and data types) and other table properties. However, it does not control the data files themselves.

**Data Independence:** External tables provide a level of data independence, as changes to the external data do not affect the table's definition or metadata.

**Data Loading:** When you query an external table, Hive reads the data from the external location. You can load or update data in the external location independently, and those changes will be reflected in the external table when queried.

**Storage Format:** External tables can work with various data formats, such as text, Parquet, ORC, or Avro, depending on how you define the table.

External tables are useful when you have data stored outside of Hive, and you want to query and analyse it using Hive's SQL-like query language without duplicating the data. This approach is beneficial for scenarios where the data is managed by other systems or when you need to share data across different tools and platforms. While external tables offer flexibility and data independence, they require careful management of the data files and locations, and you should ensure that the data remains accessible and properly formatted for Hive to work with.

1. **When do we use external table?**

External tables in Hive are typically used in situations where you want to analyse and query data that is stored outside of Hive's default storage location (e.g., HDFS) without moving or duplicating the data. Here's a use case to illustrate when you might use an external table:

**Use Case: Log Data Analysis**

Suppose you work for a company that collects log data from various web applications, and this log data is stored in a centralized storage system or in a different file system, not managed by Hive. The log data is generated by different applications and arrives in various formats, such as JSON, CSV, or plain text.

In this scenario, you can use an external table in Hive to analyse the log data without physically moving or copying it into Hive's storage. Here's how you can do it:

**Create an External Table:** You create an external table in Hive that defines the schema for the log data, including the structure of the log entries, column names, and data types. You specify the location of the external data, which can be a directory in HDFS, a remote file system, or any other accessible location.

**Query the Log Data:** With the external table defined, you can now run SQL queries on the log data using Hive's SQL-like language. You can apply filters, aggregations, and transformations to gain insights from the log entries.

**Data Updates:** Since log data is continuously generated, new log files can be added to the external location regularly. The external table automatically reflects any changes or additions in the external data, allowing you to query and analyse the most up-to-date information.

**Data Retention:** If you need to retain the log data for a certain period or maintain it in its original format, you can do so outside of Hive. The external table remains connected to this data source, making it easy to access historical log data.

**Data Sharing:** If other teams or tools also need access to the same log data, they can do so without duplicating or moving the data, thus ensuring data consistency.

In this use case, using an external table is advantageous because it allows you to analyse and gain insights from log data efficiently without incurring the overhead of copying or moving large volume.

of data. It also maintains data independence and allows for easy data sharing among different teams or systems.

1. **Diff between managed and external table?**

Managed tables and external tables in Hive differ in how they handle data storage and management. Here's a brief comparison of the key differences between managed and external tables:

**Managed Table:**

**Data Storage:** Managed tables store data in Hive's default storage location (usually HDFS), and Hive manages the data files.

**Metadata Management:** Hive manages both the table's metadata (schema, partitions, etc.) and the data.

**Data Location:** Data is stored within Hive's control, and Hive is responsible for maintaining it.

**Data Durability:** Data in managed tables is considered durable and protected from accidental data loss.

**Backup and Recovery:** Hive typically includes support for backup and recovery of managed table data.

**External Table:**

**Data Storage:** External tables reference data stored outside of Hive's default storage. Data is stored in an external location that is not controlled by Hive.

**Metadata Management:** Hive manages the table's metadata, including the schema, but it doesn't control the data files themselves.

**Data Location:** Data is stored in an external location, often managed by the user or another system.

**Data Independence:** Changes to the external data don't affect the table's definition or metadata.

**Data Loading:** When querying an external table, Hive reads the data from the external location. You can load or update data independently in the external location.

In summary, managed tables are fully controlled by Hive, including data storage and metadata management, and they are suitable when you want Hive to manage both data and metadata. On the other hand, external tables are used when you want to query and analyse data that's stored outside of Hive's default location, offering flexibility and data independence, but requiring careful management of the external data.

1. **What happens if you don’t provide location to external table?**

When you create an external table in Hive, it's essential to specify the LOCATION parameter, which tells Hive the location of the external data. If you don't provide a location for an external table, the table creation will fail, and you won't be able to use the table for querying or data manipulation. Hive requires the LOCATION parameter to know where to find the data associated with the external table.

Here's what happens if you don't provide a location for an external table:

**Table Creation Failure:** If you attempt to create an external table without specifying the LOCATION parameter, Hive will raise an error or warning, depending on the Hive version. The table won't be created, and no metadata or schema information will be associated with the table.

**No Data Access:** Without a location, Hive won't know where to find the data, making it impossible to run queries or perform any operations on the table since there's no data source associated with it.

**Data Independence Lost:** The primary advantage of external tables is their ability to work with data stored outside of Hive. Without a location, this data independence is lost, and the table becomes unusable.

To successfully create and use an external table, always specify the LOCATION parameter to point to the location of the external data you intend to query with the table. This ensures that Hive knows where to find the data when you run queries against the table.

1. **Performance optimization in hive?**

Performance optimization in Hive is crucial for ensuring that your queries run efficiently and that you can process and analyse large datasets effectively. Here are some key strategies for optimizing performance in Hive:

**Use Appropriate Data Formats:** Choose the right data storage format for your data. Columnar formats like Parquet and ORC are often more efficient for analytics queries compared to plain text or other formats.

**Partitioning:** Implement partitioning to prune unnecessary data when running queries. Hive allows you to partition data based on one or more columns, which can significantly reduce the amount of data scanned.

**Bucketing:** Bucketing is a technique to evenly distribute data into a fixed number of buckets based on a specific column's values. It can improve query performance by reducing the number of files to read.

**Optimize Joins:** Avoid performing expensive joins by using techniques like map-side joins, and make use of Bloom filters and MapReduce settings like mapreduce.reduce.input.buffer.percent to control join performance.

**Use Appropriate File Sizes:** Ensure that the size of individual data files is optimized. Smaller files can lead to overhead in query execution, so it's often best to merge smaller files into larger ones.

**Compression:** Choose appropriate compression codecs to minimize data storage size and reduce I/O. Popular codecs include Snappy, Gzip, and LZO.

**Indexing:** Hive supports indexing on certain columns, which can improve query performance. Use indexing where appropriate.

**Statistics:** Collect and use table and column statistics to help Hive's query optimizer make better decisions regarding query execution plans.

**Tez and Vectorization:** Consider using execution engines like Tez and enabling vectorized query processing. Tez can improve query execution speed and resource utilization, while vectorization improves query performance by processing data in batches.

**Hardware and Cluster Configuration**: Ensure that your cluster hardware is appropriately sized and configured for your workloads. Adjust the cluster's resource allocation and configuration parameters as needed.

**Tuning Memory Parameters:** Configure memory-related parameters like mapreduce.map.memory.mb and mapreduce.reduce.memory.mb to optimize memory usage and minimize spills to disk.

**Use Caching:** Hive supports result set caching, which can be beneficial for frequently accessed data.

**Data Pruning and Filtering:** Minimize the data scanned by applying filters and WHERE clauses in your queries to reduce the dataset processed.

**Regularly Analyse and Tune Queries:** Analyse query execution plans, monitor query performance, and use Hive's EXPLAIN statement to understand how queries are executed. Adjust query writing and configuration based on the insights gained.

**Use Vectorized UDFs:** If necessary, create and use User-Defined Functions (UDFs) in vectorized form for better performance.

Performance optimization in Hive involves a combination of data organization, query design, and cluster configuration. The specific techniques and best practices to use will depend on your use case and the characteristics of your data and cluster. Regular monitoring and iterative optimization are essential for maintaining good query performance over time.

1. **Explain partition table. Give example.**

A partitioned table in Hive is a table that is divided into subdirectories based on the values of one or more specific columns. Partitioning is a technique that helps improve query performance by allowing Hive to eliminate unnecessary data during query execution. It's especially useful when dealing with large datasets.

Here's a brief explanation and an example of a partitioned table:

**Partitioned Table Explanation:**

In a partitioned table, data is organized into subdirectories within the table's location based on the values in one or more designated columns. These columns are known as partition keys.

Partitioning is typically used when you have a column that has a high cardinality (many distinct values), and you frequently filter or query the data based on the values in that column.

Hive keeps track of the partitions and their locations in the table's metadata.

**Example of a Partitioned Table:**

Let's say you have a table storing sales data for a retail business, and you want to partition the data by the "year" and "month" columns. Here's what the directory structure of the partitioned table might look like:

A computer code with numbers and symbols

Description automatically generated

In this example, the "sales\_table" is partitioned by "year" and "month." Each unique combination of "year" and "month" values results in a separate subdirectory within the table's location. The data for each year and month is stored in the respective subdirectories.

By partitioning the table in this way, you can optimize query performance. For instance, if you want to retrieve sales data for January 2021, Hive will only scan the "year=2021/month=01" subdirectory, reducing the amount of data to process compared to scanning the entire table. When you create a partitioned table in Hive, you specify the partition keys, and Hive takes care of managing the data organization and handling partitions during query execution. This can significantly improve the speed and efficiency of queries, especially when filtering or aggregating data based on the partitioning columns.

1. **Explain bucket table. Give example.**

A bucketed table in Hive is a table that organizes data into a fixed number of buckets or files based on the values in one or more columns. Bucketing is a technique used to improve query performance by distributing data uniformly across these buckets, allowing for more efficient data retrieval and join operations.

Here's a brief explanation and an example of a bucketed table:

Bucketed Table Explanation:

In a bucketed table, data is divided into a specific number of files or buckets, and each bucket contains a portion of the data rows. The division is based on the values in one or more columns, which are known as bucketing columns.

Bucketing is typically used when you have a column with a high cardinality, and you frequently perform join operations using that column. By distributing data into buckets, Hive can optimize join operations as it knows data with the same bucketing column values will be in the same bucket.

Hive keeps track of the bucketing metadata in the table's definition.

**Example of a Bucketed Table:**

Let's say you have a table storing customer data, and you want to bucket the data based on the "customer\_id" column into 5 buckets. Here's an example of how the bucketed table might look:

A computer screen shot of a bucket

Description automatically generated

In this example, the "customer\_table" is bucketed based on the "customer\_id" column, and it's divided into 5 buckets (bucket\_00000 to bucket\_00004). The data rows are distributed evenly across these buckets based on the hash value of the "customer\_id." Rows with the same hash value for the "customer\_id" will end up in the same bucket.

When you perform join operations using the "customer\_id," Hive can efficiently perform the join by matching data within the same buckets, reducing the amount of data that needs to be scanned.

Bucketing is particularly useful when dealing with large datasets and when you often need to join tables based on the bucketing columns. It can significantly improve query performance and reduce the amount of data shuffling during join operations.

To create a bucketed table in Hive, you specify the bucketing columns and the number of buckets to use. Hive then takes care of organizing the data into the specified number of buckets and managing the bucketing metadata.

1. **Diff between partition and bucketed table**.

Partitioned tables and bucketed tables in Hive are both techniques used to optimize query performance and data organization, but they serve different purposes and have distinct characteristics. Here's a brief comparison of partitioned tables and bucketed tables:

**Partitioned Table:**

**Organization:** Data in a partitioned table is organized into subdirectories based on the values of one or more designated columns (partition keys). Each unique combination of partition key values results in a separate subdirectory.

**Purpose:** Partitioning is primarily used to reduce the amount of data scanned during query execution by pruning partitions that don't match specified filter conditions.

**Flexibility:** You can partition by high-cardinality columns, such as date, year, or category, to optimize filtering and queries based on those columns.

**Number of Files:** The number of files in each partition can vary, and the size of partitions may differ.

**Use Case:** Useful for data where you need to efficiently filter and retrieve data based on specific criteria, such as date or category.

**Bucketed Table:**

**Organization:** Data in a bucketed table is divided into a fixed number of files or buckets based on the values in one or more designated columns (bucketing columns). Rows are evenly distributed across these buckets.

**Purpose:** Bucketing is primarily used to optimize join operations, especially when joining large tables based on the bucketing column. Data with the same bucketing column values will end up in the same bucket.

**Flexibility:** Bucketing is typically applied to high-cardinality columns, and the number of buckets is predetermined, which allows for more predictable and efficient join operations.

Number of Files: Each bucket contains roughly the same number of rows, resulting in a fixed number of files.

**Use Case:** Ideal for situations where efficient join operations are essential, and the bucketing column is commonly used for joining tables.

In summary, partitioning is focused on data pruning and efficient filtering, which helps in scenarios where data needs to be retrieved based on specific criteria. Bucketing, on the other hand, is designed for optimizing join operations by distributing data uniformly across buckets. The choice between partitioned and bucketed tables depends on your specific use case and query patterns. In some cases, you may even use both techniques together to achieve optimal query performance.

1. **How is data distributed among buckets?**

Data distribution among buckets in Hive's bucketed tables is determined by the values in the bucketing columns and a hash function. Here's a brief explanation of how data is distributed among buckets:

**Choice of Bucketing Columns:** First, you specify one or more columns in your table as the bucketing columns when creating a bucketed table. These columns will be used to determine the distribution of data into buckets. The choice of bucketing columns is critical because it affects how the data is distributed.

**Hash Function:** Hive uses a hash function to calculate a hash value for each row based on the values in the bucketing columns. The hash function is designed to distribute data evenly across the specified number of buckets.

**Determining Bucket Number:** The calculated hash value for each row is used to determine which bucket the row belongs to. Hive divides the range of hash values into the desired number of buckets, and each range corresponds to a specific bucket.

**Data Distribution:** Rows with the same hash value in the bucketing columns will end up in the same bucket. This means that data with similar values in the bucketing columns will be distributed uniformly across the buckets. Rows with different hash values will be distributed into different buckets.

**Bucket Number Assignment:** Each row is assigned to the bucket number based on its hash value, ensuring that each bucket contains a roughly equal number of rows, and each row is assigned to one and only one bucket.

The primary goal of this distribution method is to achieve a balanced distribution of data across the buckets so that no single bucket becomes significantly larger or smaller than the others. Even data distribution helps optimize query performance, especially in join operations, as the data in the same bucket can be processed efficiently.

It's important to choose the appropriate number of buckets and the right bucketing columns to ensure that data is evenly distributed, and the benefits of bucketing are fully realized. Hive handles the distribution process automatically when you create a bucketed table, based on the hash values of the bucketing columns.