**ASS-32.1**

**Explain in brief**

**● What is the difference between memstore and hfile in HBase?**

|  |  |
| --- | --- |
| **Memstore** | **HFILE** |
| The MemStore is a write buffer where HBase accumulates data in memory before a permanent write. | The HFile is the underlying storage format for HBase. |
| Memstore is kept in RegionServer (RS) main memory. | HFiles are written to HDFS. |
| When write request is processed, data is first written into the Memstore. | When certain thresholds are met Memstore data gets flushed into HFile. |

**● Describe compactions in HBase.**

**Compaction:**

Apache HBase is a distributed data store based upon a log-structured merge tree, so optimal read performance would come from having only one file per store (Column Family). However, that ideal isn’t possible during periods of heavy incoming writes. Instead, HBase will try to combine HFiles to reduce the maximum number of disk seeks needed for a read. This process is called compaction.

Compaction, the process by which HBase cleans up after itself, comes in two flavors: major and minor

**Minor compactions:**

1. Minor compactions combine a configurable number of smaller HFiles into one larger HFile.

2. You can tune the number of HFiles to compact and the frequency of a minor compaction.

3. Minor compactions are important because without them, reading a particular row can require many disk reads and cause slow overall performance.

4. Until the HBase system performs a minor compaction, reading from Row 00001 would require three disk reads to retrieve the relevant HFile content.

5. Minor compactions seek to minimize system overhead while keeping the number of HFiles under control.

6. HBase designers took special care to give the HBase administrator as much tuning control as possible to make any system impact “minor.”

**Major compaction:**

1. As its name implies, a major compaction is different from the perspective of a system impact. However, the compaction is quite important to the overall functionality of the HBase system.

2. A major compaction seeks to combine all HFiles into one large HFile.

3. In addition, a major compaction does the cleanup work after a user deletes a record. When a user issues a Delete call, the HBase system places a marker in the key-value pair so that it can be permanently removed during the next major compaction.

4. Additionally, because major compactions combine all HFiles into one large HFile, the time is right for the system to review the versions of the data and compare them against the time to live (TTL) property. Values older than the TTL are purged.

5. Users who are trying to add, retrieve, or manipulate data in the system during a major compaction, they may see poor system response time.

6. In addition, the HBase cluster may have to split regions at the same time that a major compaction is taking place and balance the regions across all RegionServers. This scenario would result in a significant amount of network traffic between RegionServers.

7. For these reasons, your HBase administrator needs to have a major compaction strategy for your deployment.

**● List and explain the logical entities in HBase.**

**HBase Data Model**

The Data Model in HBase is designed to accommodate semi-structured data that could vary in field size, data type and columns. Additionally, the layout of the data model makes it easier to partition the data and distribute it across the cluster. The Data Model in HBase is made of different logical components such as Tables, Rows, Column Families, Columns, Cells and Versions.

**Tables** – The HBase Tables are more like logical collection of rows stored in separate partitions called Regions. Every Region is then served by exactly one Region Server.

**Rows** – A row is one instance of data in a table and is identified by a rowkey. Rowkeys are unique in a Table and are always treated as a byte[].

**Column Families** – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it’s important that proper care be taken when designing Column Families in table. The table above shows Customer and Sales Column Families. The Customer Column Family is made up 2 columns – Name and City, whereas the Sales Column Families is made up to 2 columns – Product and Amount.

**Columns** – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily:columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied number of Columns.

**Cell** – A Cell stores data and is essentially a unique combination of rowkey, Column Family and the Column (Column Qualifier). The data stored in a Cell is called its value and the data type is always treated as byte[].

**Version** – The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

**● What will happen if we do not create a row key while inserting the data?**

1. HBase data stores consist of one or more tables, which are indexed by row keys.

2. Data is stored in rows with columns, and rows can have multiple versions. By default, data versioning for rows is implemented with time stamps.

3. Row keys are implemented as byte arrays, and are sorted in byte-lexicographical order, which simply means that the row keys are sorted, byte by byte, from left to right.

4. The reason for this special attention is that proper row key design is crucial to achieving good performance in HBase — not doing so means you won’t realize the full value of your HBase cluster.

5. Sorted row keys can help you access your data faster.

**● How can filters be applied in HBase and what are the benefits?**

1. HBase can query data very quickly on demand but specific use cases may require to only return a subset of the scan results.

2. Instead of scanning the entire dataset only to return a subset to the client, we can use Filters to get the data closer to what we need in less amount of time.

3. Thus, HBase has a set of predefined Filters as well as custom filters that we can use to scan and get filtered results from the HBase database.

4. There are two prominent ways to read data from HBase.

-Get is simply a Scan limited by the API to one row.

-Scan fetches zero or more rows of a table. By default, a Scan reads the entire table from start to end. We can limit our Scan results in several different ways, which affect the Scan’s load in terms of IO, network, or both, as well as processing load on the client side.

5. When reading data from HBase using Get or Scan operations, we can use custom filters to return a subset of results to the client. It does reduce network bandwidth and reduces the amount of data the client needs to process.

6. Filters are generally used when using the Java API and take zero or more arguments, in parentheses. Where the argument is a string, it is surrounded by single quotes (‘string’).

**● What are the data model operations in hBase?**

**Data Model Operations**

The four primary data model operations are Get, Put, Scan, and Delete. Operations are applied via HTable instances.

**1. Get**

Get returns attributes for a specified row. Gets are executed via HTable.get.

**2. Put**

Put either adds new rows to a table (if the key is new) or can update existing rows (if the key already exists). Puts are executed via HTable.put (writeBuffer) or HTable.batch (non-writeBuffer).

**3. Scans**

Scan allow iteration over multiple rows for specified attributes.

**4. Delete**

Delete removes a row from a table. Deletes are executed via HTable.delete.

HBase does not modify data in place, and so deletes are handled by creating new markers called tombstones. These tombstones, along with the dead values, are cleaned up on major compactions.

**● How can MapReduce be used with HBase?**

1. HBase provides a TableInputFormat, to which you provided a table scan, that splits the rows resulting from the table scan into the regions in which those rows reside.
2. The map process is passed an ImmutableBytesWritable that contains the row key for a row and a Result that contains the columns for that row.
3. The map process outputs its key/value pair based on its business logic in whatever form makes sense to your application.
4. The reduce process builds its results but emits the row key as an ImmutableBytesWritableand a Put command to store the results back to HBase.
5. Finally, the results are stored in HBase by the HBase MapReduce infrastructure. (You do not need to execute the Put commands.)

**What is regionserver?**

1. RegionServers are the software processes (often called daemons) you activate to store and retrieve data in HBase (Hadoop Database).

2. In production environments, each RegionServer is deployed on its own dedicated compute node.

3. When you start using HBase, you create a table and then begin storing and retrieving your data.

4. However, at some point — and perhaps quite quickly in big data use cases — the table grows beyond a configurable limit. At this point, the HBase system automatically splits the table and distributes the load to another RegionServer.

5. In this process, often referred to as auto-sharding, HBase automatically scales as you add data to the system — a huge benefit compared to most database management systems, which require manual intervention to scale the overall system beyond a single server.

6. With HBase, as long as you have in the rack another spare server that’s configured, scaling is automatic.