ASSIGNMENT 10.1

TASK1

1.What is NoSQL data base?

Ans.NoSQL database refers to Not Only SQL databases.It is an approach to databases that represents a shift away from traditional databse management systems (RDMS).

Relational databases rely on tables, columns, rows, or schemas to organize and retrieve data. In contrast, NoSQL databases do not rely on these structures and use more flexible data models.

As RDBMS have increasingly failed to meet the performance, scalability, and flexibility needs that next-generation, data-intensive applications require, NoSQL databases have been adopted by mainstream enterprises. NoSQL is particularly useful for storing unstructured data, which is growing far more rapidly than structured data and does not fit the relational schemas of RDBMS.

Common types of unstructured data include: user and session data; chat, messaging, and log data; time series data such as IoT and device data; and large objects such as video and images.

BENEFITS OF NOSQL

NoSQL databases offer enterprises important advantages over traditional RDBMS, including:

- **Scalability:** NoSQL databases use a horizontal scale-out methodology that makes it easy to add or reduce capacity quickly and non-disruptively with commodity hardware. This eliminates the tremendous cost and complexity of manual sharding that is necessary when attempting to scale RDBMS.
- **Performance:** By simply adding commodity resources, enterprises can increase performance with NoSQL databases. This enables organizations to continue to deliver reliably fast user experiences with a predictable return on investment for adding resources—again, without the overhead associated with manual sharding.
- **High Availability:** NoSQL databases are generally designed to ensure high availability and avoid the complexity that comes with a typical RDBMS architecture that relies on primary and secondary nodes. Some “distributed” NoSQL databases use a masterless architecture that automatically distributes data equally among multiple resources so that the application remains available for both read and write operations even when one node fails.
- **Global Availability:** By automatically replicating data across multiple servers, data centers, or cloud resources, distributed NoSQL databases can minimize latency and ensure a consistent application experience wherever users are located. An added benefit is a significantly reduced database management burden from manual RDBMS configuration, freeing operations teams to focus on other business priorities.

- **Flexible Data Modeling:** NoSQL offers the ability to implement flexible and fluid data models. Application developers can leverage the data types and query options that are the most natural fit to the specific application use case rather than those that fit the database schema. The result is a simpler interaction between the application and the database and faster, more agile development.

2.How does data get stored in NoSQl database?

Ans. HBase stores data in a form of a distributed sorted multidimensional persistence maps called Tables.

- **Key-value data stores:** It emphasize simplicity and are very useful in accelerating an application to support high-speed read and write processing of non-transactional data. Stored values can be any type of binary object (text, video, JSON document, etc.) and are accessed via a key. The application has complete control over what is stored in the value, making this the most flexible NoSQL model. Data is partitioned *and replicated* across a cluster to get scalability and availability. For this reason, key value stores often do not support transactions. However, they are highly effective at scaling applications that deal with high-velocity, non-transactional data.
- **Document stores:** typically store self-describing JSON, XML, and BSON documents. They are similar to key-value stores, but in this case, a value is a single document that stores all data related to a specific key. Popular fields in the document can be indexed to provide fast retrieval without knowing the key. Each document can have the same or a different structure.
- **Wide-column stores:** Wide-column NoSQL databases store data in tables with rows and columns similar to RDBMS, but names and formats of columns can vary from row to row across the table. Wide-column databases group columns of related data together. A query can retrieve related data in a single operation because only the columns associated with the query are retrieved. In an RDBMS, the data would be in different rows stored in different places on disk, requiring multiple disk operations for retrieval.
- **Graph stores:** A graph database uses graph structures to store, map, and query relationships. They provide index-free adjacency, so that adjacent elements are linked together without using an index.

In the in-memory databases like Redis/CouchBase/Tarantool/Aerospike everything is stored in RAM in balanced trees like RB-Tree or in hash tables.

All the writes are applied on both RAM and disk, but on disk it goes in an append-only way.

A file append can be done as fast as 100Mbytes per second on a normal magnetic disk. If a record size is, say, 1K, then the data will be written at 100krps.

In the disk NOSQL databases and db-engines like Cassandra/HBase/RocksDB/LevelDB/Sophia the main idea is that you have a snapshot file and a write ahead log (WAL) file.

Snapshot contains already prepared data in a form of B-Tree with upper levels of that tree being permanently in RAM, that can be accesses for reading by doing only one disk seek.

A WAL contains all the new changes on top of a current snapshot. Asnapshot file is being totally rebuilt on a regular basis using currentsnapshot and a WAL.

All the writes are done nearly as fast as with inmemorydatabases. "Nearly" because disk is partially busy by doing regular snapshot converting that was described earlier. Reads are significantly slower than that are in in-memory databases, because they take at least one disk seek, but good news is that they can be cached in optimized in-memory structures like RB-Trees/hash tables.

3.What is a column family in HBase?

Ans. Columns in HBase are grouped into column families. All column members of a column family have the same prefix.

For example, the columns courses:history and courses:math are both members of the courses column family.

The colon character (:) delimits the column family from the .

The column family prefix must be composed of printable characters. The qualifying tail, the column family qualifier, can be made of any arbitrary bytes.

Column families must be declared up front at schema definition time whereas columns do not need to be defined at schema time.

Physically, all column family members are stored together on the filesystem. Because tunings and storage specifications are done at the column family level,it is advised that all column family members have the same general access pattern and size characteristics.

4.How many maximum number of columns can be added to HBase table?

Ans. HBase currently does not do well with anything above two or three column families so we should keep the number of column families in your schema low.

Currently, flushing and compactions are done on a per Region basis so if one column family is carrying the bulk of the data bringing on flushes, the adjacent families will also be flushed though the amount of data they carry is small.

We can introduce a second and third column family in the case where data access is usually column scoped; i.e. you query one column family or the other but usually not both at the one time.

5.Why columns are not defined at the time of table creation in HBase?

Ans. Column families must be declared up front at schema definition time whereas columns do not need to be defined at schema time because it can be conjured on the fly while the table is up an running.

6.How does data get managed in HBase?

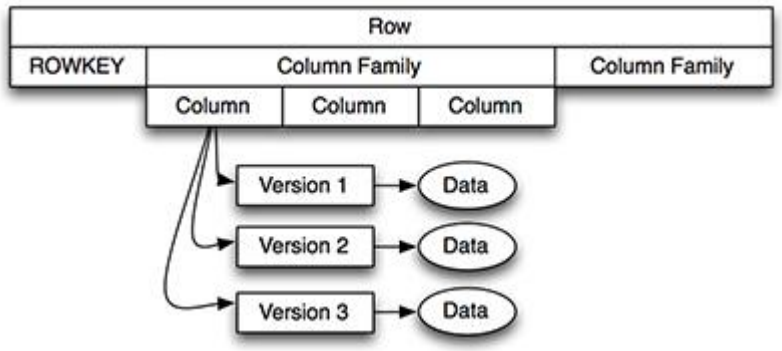
Ans. HBase is not a relational database and requires a different approach to managing data. HBase actually defines a four-dimensional data model to manage data:

- Row Key: Each row has a unique row key; the row key does not have a data type and is treated internally as a byte array.
- Column Family: Data inside a row is organized into column families; each row has the same set of column families, but across rows, the same column families do not need the same column qualifiers. Under-the-hood, HBase stores column families in their own data files, so they need to be defined upfront, and changes to column families are difficult to make.

Column Qualifier: Column families define actual columns, which are called column qualifiers. You can think of column qualifiers as the columns themselves.A unique combination of row key, column family and column qualifier forms a cell. Data contained in a cell is referred to as cell value. There is no concept of data type when referring to cell values and they are stored as bytearrays. Indexing and sorting only happens on the row key.

- Version: Each column can have a configurable number of versions, and you can access the data for a specific version of a column qualifier.

  Above four coordinates define each cell shown below in the figure



HBase Four-Dimensional Data Model

To access an individual piece of data, we need to know its row key, column family, column qualifier, and version.Tables in Hbase have several properties that need to be understood for one to come up with an effective data model.

7.What happens internally when new data gets inserted into HBase table?

Ans. When we inserted the new data into Hbase table,it will replace the existing value present in the table.We can use **put** command to update the existing cell value.

Task 2

1. Create an HBase table named 'clicks' with a column family 'hits' such that it should be able to store last 5 values of qualifiers inside 'hits' column family.

2. Add few records in the table and update some of them. Use IP Address as row-key. Scan the table to view if all the previous versions are getting displayed.

```
hbase(main):011:0>
hbase(main):012:0* put 'clicks', '127.0.0.1', 'hits:quantity', '3'
0 row(s) in 0.1490 seconds

hbase(main):013:0>
hbase(main):014:0* put 'clicks', '127.0.0.1', 'hits:date', '2017-04-20'
0 row(s) in 0.0060 seconds

hbase(main):015:0>
hbase(main):016:0* put 'clicks', '127.0.0.1', 'hits:url', 'https://github.com'
0 row(s) in 0.0040 seconds

hbase(main):017:0>
hbase(main):018:0* put 'clicks', '127.0.0.1', 'hits:rating', '5'
0 row(s) in 0.0110 seconds

hbase(main):019:0> put 'clicks', '127.0.0.1', 'hits:item', 'image'
0 row(s) in 0.0130 seconds
```

acadgild@localhost:~

```
hbase(main):020:0> scan 'clicks'
ROW                                   COLUMN+CELL
 127.0.0.1                            column=hits:date, timestamp=1519452966589, value=2017-04-20
 127.0.0.1                            column=hits:item, timestamp=1519453022886, value=image
 127.0.0.1                            column=hits:quantity, timestamp=1519452966467, value=3
 127.0.0.1                            column=hits:rating, timestamp=1519452991894, value=5
 127.0.0.1                            column=hits:url, timestamp=1519452966635, value=https://github.com
1 row(s) in 0.0280 seconds

hbase(main):021:0>
```

Updating the values in table clicks, scan to check the updated values and describe command will show the version.

acadgild@localhost:~

```
hbase(main):026:0> alter 'clicks', {NAME => 'hits', VERSIONS => 2}
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.9910 seconds

hbase(main):027:0>
hbase(main):028:0* put 'clicks', '127.0.0.1', 'hits:url', 'https://facebook.com'
0 row(s) in 0.0050 seconds

hbase(main):029:0>
hbase(main):030:0* put 'clicks', '127.0.0.1', 'hits:rating', '10'
0 row(s) in 0.0100 seconds

hbase(main):031:0>
```

acadgild@localhost:~

```
hbase(main):031:0> scan 'clicks'
ROW                                   COLUMN+CELL
 127.0.0.1                            column=hits:date, timestamp=1519452966589, value=2017-04-20
 127.0.0.1                            column=hits:item, timestamp=1519453022886, value=image
 127.0.0.1                            column=hits:quantity, timestamp=1519452966467, value=3
 127.0.0.1                            column=hits:rating, timestamp=1519454200406, value=10
 127.0.0.1                            column=hits:url, timestamp=1519454198288, value=https://facebook.com
1 row(s) in 0.0400 seconds
```

Describe will show the version details.