Session 10: HBASE BASICS Assignments 1.

Task 1: Answer in your own words with examples.

1. **What is NoSQL database?**

Ans: NoSQL is an approach to databases that represents a shift away from traditional relational database management (RDBMS). To define NoSQL, it is helpful to start by describing SQL, which is a query language used by RDBMS. 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. NoSQL can mean "Not SQL" or "not only SQL". 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.

Types of NoSQL Databases

1. **Key-value data stores:** Key-value NoSQL databases 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.)
2. **Document stores:** Document databases 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.
3. **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.
4. **Graph stores:** A graph database uses graph structure to store, map, and query relationships.

BENEFITS OF NOSQL

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

1. **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.
2. **Performance:** By simply adding commodity resources, enterprises can increase performance with NoSQL databases.
3. **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.
4. **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 are located.
5. **Flexible data modeling:** NoSQL offers the ability to implement flexibile 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.

2. **How does data get stored in NoSQL database?**

Ans: There are various NoSQL Databases. Each one uses a different method to store data. Some might use column store, some document, some graph etc., Each database has its own unique characteristics.

- **Document databases** Pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
- **Graph Stores** are used to store information about networks of data, such as social connections. Graph stores include Neo4J and Giraph.
- **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its values. Examples of key-value stores are Risk and Berkeley DB. Some key-value stores, such as Redis, allow each value to have a type, such as 'integer', which adds functionality.

- **Wide-column stores** Such as Cassandra and HBase are optimized for queries over large datasets and store columns of data together, instead of rows.

3. What is a column family in HBase?

Ans: Columns in Apache HBase are grouped into *column families.* All column members of a column family have the same prefix. For example, the column *course:history* and *courses:math* are both members of the *courses* column family. The column 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 defined at schema time but can be conjured on the fly while the table is up an running.

Physically, all column family members are stored together on the file system. 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 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 even though the amount of data they carry is small. When many column families exist the flushing and compaction interaction can make for a bunch of needles i/o (To be addressed by changing flushing and compaction to work on a per column family basis).

Try to make do with one column family if you can in your schemas. Only 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.

Cardinality of columnFamilies

Where multiple ColumnFamilies exist in a single table, be aware of the cardinality (i.e., number of rows). If ColumnFamilyA has 1 million rows and ColumnFamilyB has 1 billion rows, ColumnFamilyA's data will likely be spread across many, many regios (and RegionServers). This makes mass scan for ColumnFamily A less efficient.

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

Ans: Column families are part of the schema of the table. You can add them at runtime with an online schema change. But you wouldn't add them dynamically the way that you can dynamically create new "columns" in an HBase table, if that's what you had in mind.

The reason column families are part of the schema and would require a schema change is that they profoundly impact the way the data is stored , both on disk on disk in memory. Each column family has its own set of HFiles, and its own set of data structures in memory of the RegionServer. It would be pretty expensive to dynamically create or start using new column families.

Column families are only needed when you need to configure differently various parts of a table (for instance you want some columns to have a TTL and others to not expire), or when you want to control the locality of accesses (things accessed together should better be in the same column family if you want good performance, as the cost of operations grows linearly with the number of column families). So, again, because of those specialized reasons, it doesn't make sense to dynamically add new column families at runtime the way you would add regular "columns" within a family.

6. How does data get managed in HBase?

Ans: NoSQL databases are designed for scalability where unstructured data is spread across multiple nodes. When data volumes increase you just need to add another node to accommodate the growth. The lack of structure in NoSQL databases relaxes stringent requirements of consistency enforced in relational databases to improve speed and agility. HBase, MongoDB and Cassandra are the major options that provide NoSQL capabilities. The options differ in the features they provide, so the decision on which to use is informed by the workload that will be handled. The main difference between HBase and Cassandra databases is the consistency model they implement. Cassandra implements eventual consistency which guarantees writes are available. This provides excellent write scaling but suffers a penalty

when reading because for consistency in reads you have to read from many copies of data. On the other hand HBase provides a strong consistency model that excels at scaling reads but does not scale on writes as well as Cassandra does.

Hbase is natively supported on Hadoop and it is the subject of this tutorial. The main characteristics that make Hbase an excellent data management platform are fault tolerance, speed and usability. Fault tolerance is provided by automatic fail-over, automatically sharded and load balanced tables, strong consistency in row level operations and replication. Speed is provided by almost real time lookups, in memory caching and server side processing. Usability is provided by a flexible data model that allows many uses, a simple Java API and ability to export metrics.

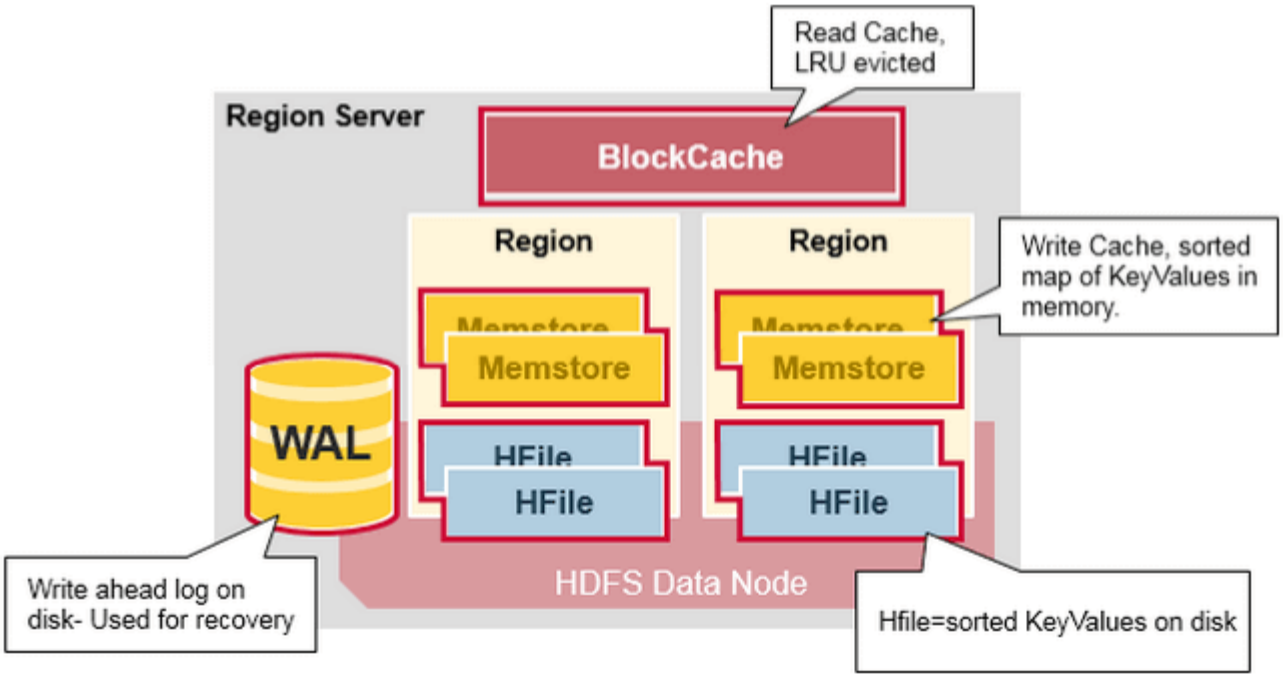
Hbase can run standalone on the local file system but this set up does not guarantee durability. Edits will be lost when daemons are not cleanly started and stopped. Such a set up is not suitable in a production environment but it provides a way of exploring how the database functions. Alternatively Hbase can be installed on a single or multi node cluster and use HDFS.

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

Ans: Once client finds region in HBase to write, the client sends the request to the region server which has that region for changes to be accepted. The region server cannot write the changes to a **HFile** immediately because the data in a HFile must be sorted by the row key. This allows searching for random rows efficiently when reading the data. Data cannot be randomly inserted into the HFile.

Instead, the change must be written to a new file. If each update were written to a file, many small files would be created. Such a solution would not be scalable nor efficient to merge or read at a later time. Therefore, changes are not immediately written to a new HFile.

Instead, each changes is stored in a place in memory called the ***memstore***, which cheaply and efficiently supports random writes. Data in the memstore is sorted in the same manner as data in a HFile. When the memstore accumulates enough data, the entire sorted set is written to a new HFile in HDFS.

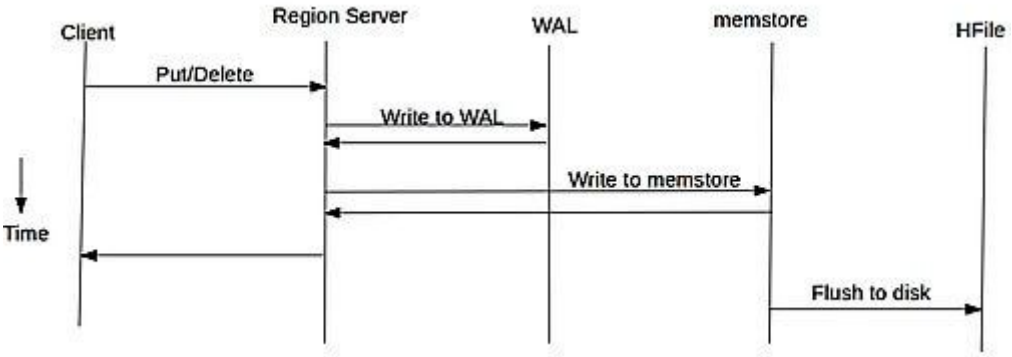


Although writing data to the memstore is efficient, it also introduces an element of risk: Information stored in memstore is stored in volatile memory, so if the sytem fails, all memstore information is lost. To help mitigate the risk, HBase saves updates in a ***write-ahead-log(WAL)*** before writing the information to memstore. In this way, if a region server fails, information that was stored in that server's memstore can be recovered from its WAL.

WAL files contain a list of edits, with one edit representing a single put or delete. The edit includes information about the change and the region to which the change applies. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk.

As WALs grow, they are eventually closed and a new, active WAL file is created to accept additional edits. This is called ***rolling the WAL file***. Once a WAL file is rolled, no additional changes are made to the old file. By default, WAL file is rolled when its size is about 95% of the HDFS block size.

A region server serves many regions, but does not have a WAL file for each region. Instead, one active WAL file is shared among all regions served by the region server. Because WAL files are rolled periodically, one region server may have many WAL files. Note that there is only one active WAL per region server at a given time.



Now you know what happens internally during HBase write operation. Let us summarize the write process:

- When the client gives a command to write, instruction is directed to **Write Ahead Log.**
- Once the log entry is done, the data to be written is forwarded to ***MemStore*** which is actually the RAM of the data node.
- Data in the memstore is sorted in the same manner as data in a HFile.
- When the memstore accumulate enough data, the entire sorted set is written to a new **HFile** in HDFS.
- Once writing data is completed, **ACK** (Acknowledgement) is sent to the client as a confirmation of task completed.

Task 2 – 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.

1. Write HBASE query for creating table '***clicks***' and column family as '***hits***' as follws:
   hbase> create 'clicks', 'hits'

```
acadgild@localhost:~
File  Edit  View  Search  Terminal  Help
hbase(main):127:0> create 'clicks','hits'
0 row(s) in 1.2310 seconds

=> Hbase::Table - clicks
hbase(main):128:0> scan 'clicks'
ROW                    COLUMN+CELL
0 row(s) in 0.0110 seconds
```

2. Use ***describe 'clicks'*** to display the structure of the table as follos:
   hbase> describe 'clicks'

```
acadgild@localhost:~
File  Edit  View  Search  Terminal  Help
hbase(main):129:0> describe  'clicks'
Table clicks is ENABLED
clicks
COLUMN FAMILIES DESCRIPTION
{NAME => 'hits', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KE
EP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', CO
MPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65
536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0400 seconds
```

3. Observe that the VERSIONS => '1' in the describe. Now we will set the versions.
4. Create values for 'hits' column family as follows:
   hbase> alter 'clicks', {NAME => 'hits', VERSIONS => 5}

```
acadgild@localhost:~
File  Edit  View  Search  Terminal  Help
hbase(main):130:0> alter 'clicks', {NAME => 'hits', VERSIONS => 5}
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.9860 seconds
```

5. Now ***describe 'clicks'*** to display the structure of the table as follows:
   hbase> describe 'clicks'

```
hbase(main):131:0> describe  'clicks'
Table clicks is ENABLED
clicks
COLUMN FAMILIES DESCRIPTION
{NAME => 'hits', BLOOMFILTER => 'ROW', VERSIONS => '5', IN_MEMORY => 'false', KE
EP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', CO
MPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65
536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0210 seconds
```

6. Now describe the VERSIONS => '5' set as 5.

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.

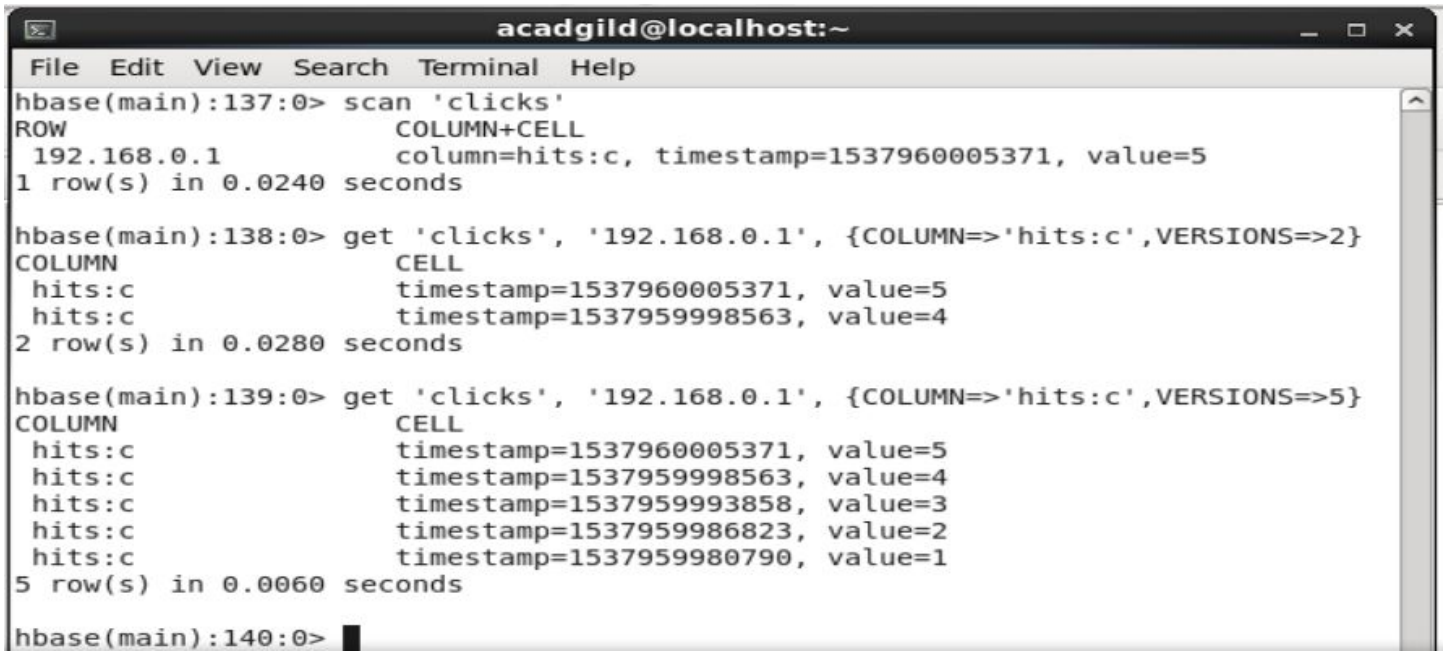
1. Now add data into the table as follows:

```
acadgild@localhost:~
File  Edit  View  Search  Terminal  Help
hbase(main):132:0> put 'clicks', '192.168.0.1', 'hits:c', '1'
0 row(s) in 0.0120 seconds

hbase(main):133:0> put 'clicks', '192.168.0.1', 'hits:c', '2'
0 row(s) in 0.0290 seconds

hbase(main):134:0> put 'clicks', '192.168.0.1', 'hits:c', '3'
0 row(s) in 0.0470 seconds

hbase(main):135:0> put 'clicks', '192.168.0.1', 'hits:c', '4'
0 row(s) in 0.0290 seconds

hbase(main):136:0> put 'clicks', '192.168.0.1', 'hits:c', '5'
0 row(s) in 0.0400 seconds

hbase(main):137:0>
```

2. Now scan the table to print the value of the column '***hits:c***' as follows:
   hive> scan 'clicks'

```
acadgild@localhost:~
File  Edit  View  Search  Terminal  Help
hbase(main):137:0> scan 'clicks'
ROW                   COLUMN+CELL
 192.168.0.1          column=hits:c, timestamp=1537960005371, value=5
1 row(s) in 0.0240 seconds
```

3. Now get the 'clicks' table to version 2 and versions 5 and lets see the value of the column '***hits:c'*** as follows:



This shows the value of '***hits:c'*** column for different versions.