Problem Statement

1. What is NoSQL data base?

NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:

Developers are working with applications that create massive volumes of new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data.

Long gone is the twelve-to-eighteen month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.

Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.

Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available

1. How does data get stored in NoSQl database?

**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 on-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. A snapshot file is being totally rebuilt on a regular basis using current snapshot and a WAL. All the writes are done nearly as fast as with in-memory databases. "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.

1. What is a column family in HBase?

Columns in Apache HBase are grouped into column families. All column members of a column family have the same prefix. For example, the columns courses:history andcourses: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 ofprintable 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 but can be conjured on the fly while the table is up an running.

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.

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

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 though the amount of data they carry is small. When many column families the flushing and compaction interaction can make for a bunch of needless i/o loading (To be addressed by changing flushing and compaction to work on a per column family basis). For more information on compactions,

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.

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

Column families are specified when a table is created. They should be carefully designed before a table is created since it would be either impossible or difficult to change them later.

Column families’ names are strings that are composed of characters that are safe to use in file system paths.

All columns in a column family are stored and sorted together in the same HFile.

Column families group columns together physically and logically and they are usually used for a performance reason. A column family has a set of parameters that specify its storage (e.g., caching, compression, etc.). All tuning and storage specifications are done at the column family level. It is important that all column family members have the same or similar access pattern and sizes.

Some shortcomings in the current HBase implementation do not properly support large number of column families in a single table. That number should be in low tens. Most of the time up to three column families should work fine without any significant performance drawback. Ideally you should go with a single column family. The column family names should be as small as possible, preferably one character.

A column family can have an arbitrary number of columns denoted by a column qualifier which is like a column’s label. For example:

{

"row1": {"1": {"color": "green",

"size": 25},

"2": {"weight": 52,

"size": 18}

},

"row2": {"1": {"color": "blue"},

"2": {"height": 192,

"size": 43}

}

}

As you can see in the example above, the same column family (e.g., “1”) in two rows can have different columns. In row “row1”, it has columns “color” and “size”, while in row “row2”, it has only “color” column. It can also have a column that is none of the above. Since rows can have different columns in column families there is no a single way to query for a list of all columns in all column families. This means that you have to do a full table scan.

There is no specific limit on the number of columns in a column family. Actually you can have millions of columns in the single column family.

1. How does data get managed in HBase?

Key/value pairs in HBASE maps are kept in an alphabetical order. The amount of data you can store in HBase can be huge and the data you are retrieving via your queries should be near each other.

For example, if you run a query on an HBase table that returns thousands of rows which are distributed across many machines, the latency affected by your network can be significant. This data distribution is determined by a row key of the HBase table. Because of that the row key design is one of the most important aspects of the HBase data modeling (schema design). If a row key is not properly designed it can create hot spotting where a large amount of client traffic is directed at one or few nodes of a cluster.

The row key should be defined in a way that allows related rows to be stored near each other. These related rows will be retrieved by queries and as long as they are stored near each other you should experience good performance. Otherwise the performance of your system will be impacted.

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

HBase’s column-oriented architecture allows for huge, wide, sparse tables.

HBase is strongly consistent on a row-level since a single region always manage an entire row.

Multiversioning can help us to avoid edit conflicts caused by concurrent data access processes and also retain data for whatever time it is needed as long as enough storage is provided.

When data schemas are properly designed, HBase provides excellent random read performance and near-optimal write operations in terms of I/O channel saturation.

HBase makes an efficient use of storage by supporting pluggable compression algorithms.

HBase extends the Bigtable model, which only considers a single index. In addition, it provides push-down predicates, that is, filters, reducing data transferred over the network.

Designing the schema in a way to completely avoid explicit locking, combined with row-level atomicity, gives you the ability to scale your system without any notable effect on read or write performance.

There are several different ways to write data into HBase. Some of them are listed below.

A Put operation writes data into HBase.

A Delete operation deletes data from HBase. What actually happens during a Delete depends upon several factors.

A CheckAndPut operation performs a Scan before attempting the Put, and only does the Put if a value matches what is expected, and provides row-level atomicity.

A CheckAndDelete operation performs a Scan before attempting the Delete, and only does the Delete if a value matches what is expected.

An Increment operation increments values of one or more columns within a single row, and provides row-level atomicity.