Session 9

Assignment 3

|  |  |
| --- | --- |
| **Prepared For:** | AcadGild |
|  |  |
| **Document Approval:** | **AcadGild** |
|  |  |
|  |  |
|  |  |
|  |  |
| **Project Title:** | Session 9 – Assignment 3 |
|  |  |
| **Prepared By:** | Duncan Burgess |
|  |  |
|  | dburgess@duncb.com |
|  |  |
| **Primary Engineer:** | Duncan Burgess |
|  |  |
| **Document Reference:** | **Session 9 – Assignment 3** |
|  |  |
| **Start Date:** | 25/09/2017 |
|  |  |
|  |  |

# 

# Contents

[Contents 2](#_Toc494105319)

[Change History 3](#_Toc494105320)

[1. Problem Statement 4](#_Toc494105321)

[1.1. NoSQL Databases 5](#_Toc494105322)

[1.2. Types of NoSQL Databases 5](#_Toc494105323)

[1.3. CAP Theorem 6](#_Toc494105324)

[1.4. HBase Architecture 8](#_Toc494105325)

[1.5. HBase vs RDBMS 14](#_Toc494105326)

# Change History

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Document Revision** | **Date** | **Authored By** | **Authorised By** | **Sections Affected** | **Reason for Change** |
| Rev 01 | 25/09/2017 | Duncan Burgess |  | All | Initial release. |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# Problem Statement

Explain the below concepts with an example in brief.

* NoSQL Databases
* Types of NoSQL Databases
* CAP Theorem
* HBase Architecture
* HBase vs RDBMS

## NoSQL Databases

A NoSQL database environment is, simply put, a non-relational and largely distributed database system that enables rapid, ad-hoc organization and analysis of extremely high-volume, disparate data types. NoSQL databases are sometimes referred to as cloud databases, non-relational databases, Big Data databases and a myriad of other terms and were developed in response to the sheer volume of data being generated, stored and analysed by modern users (user-generated data) and their applications (machine-generated data).

In general, NoSQL databases have become the first alternative to relational databases, with scalability, availability, and fault tolerance being key deciding factors. They go well beyond the more widely understood legacy, relational databases (such as Oracle, SQL Server and DB2 databases) in satisfying the needs of today’s modern business applications. A very flexible and [**schema-less data model**](http://planetcassandra.org/blog/post/schema-vs-schema-less), horizontal scalability, distributed architectures, and the use of languages and interfaces that are “not only” SQL typically characterise this technology.

From a business standpoint, considering a NoSQL or ‘Big Data’ environment has been shown to provide a clear competitive advantage in numerous industries. In the ‘age of data’, this is compelling information as a great saying about the importance of data is summed up with the following “if your data isn’t growing then neither is your business”.

## Types of NoSQL Databases

There are four general types of NoSQL databases, each with their own specific attributes:

* Graph database – Based on [graph theory](https://web.archive.org/web/20140922065727/http:/en.wikipedia.org/wiki/Graph_theory), these databases are designed for data whose relations are well represented as a graph and has elements which are interconnected, with an undetermined number of relations between them. Examples include: Neo4j and Titan.
* Key-Value store – we start with this type of database because these are some of the least complex NoSQL options. These databases are designed for storing data in a schema-less way. In a key-value store, all of the data within consists of an indexed key and a value, hence the name. Examples of this type of database include:[Cassandra](https://web.archive.org/web/20140922065727/http:/www.datastax.com/resources/tutorials/cassandra-overview), DyanmoDB, Azure Table Storage (ATS), Riak, BerkeleyDB.
* Column store – (also known as wide-column stores) instead of storing data in rows, these databases are designed for storing data tables as sections of columns of data, rather than as rows of data. While this simple description sounds like the inverse of a standard database, wide-column stores offer very high performance and a highly scalable architecture. Examples include: **HBase**, BigTable and HyperTable.
* Document database – expands on the basic idea of key-value stores where “documents” contain more complex in that they contain data and each document is assigned a unique key, which is used to retrieve the document. These are designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Examples include: MongoDB and CouchDB.

The following table lays out some of the key attributes that should be considered when evaluating NoSQL databases.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Datamodel | Performance | Scalability | Flexibility | Complexity | Functionality |
| Key-value store | High | High | High | None | Variable (None) |
| Column Store | High | High | Moderate | Low | Minimal |
| Document Store | High | Variable (High) | High | Low | Variable (Low) |
| Graph Database | Variable | Variable | High | High | Graph Theory |

## CAP Theorem

Amazon found that every 100ms of latency cost them 1% of sales. Google discovered that a half-second increase in search latency dropped traffic by 20%.

The need for scaling up/down/out is growing and so are the challenges of dealing with huge distributed systems. So, when designing such applications, it’s important to keep three core requirements in mind as described by Brewer’s CAP theorem:

**1. Consistency  
2. Availability  
3. Partition-Tolerance**

The CAP theorem was first proposed by Eric Brewer of the University of California, Berkeley, in 2000, and then proven by Seth Gilbert and Nancy Lynch of MIT in 2002. Read more [**here**](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf).

Defining The Three Core Requirements

**Consistency (C)** requires that all reads initiated after a successful write return the same and latest value at any given logical time.

**Availability (A)** requires that every node (not in failed state) always execute queries. Let’s say we have “n” servers serving our application. To ensure better availability we would add an additional “x” servers.

**Partition Tolerance (P)** requires that a system be able to re-route a communication when there are temporary breaks or failures in the network. The goal is to maintain synchronization among the involved nodes.

Brewer’s theorem states that it’s typically not possible for a distributed system to provide all three requirements simultaneously because one of them will always be compromised.

**Trade-offs Between Requirements**

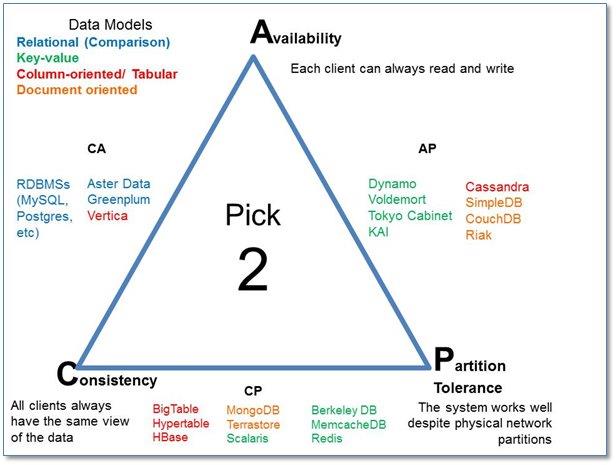
1. Available and Partition-Tolerant: Say you have two nodes and the link between the two is severed. Since both nodes are up, you can design the system to accept requests on each of the nodes, which will make the system available despite the network being partitioned. However, each node will issue its own results, so by providing high availability and partition tolerance you’ll compromise consistency.
2. Consistent and Partition-Tolerant: Say you have three nodes and one node loses its link with the other two. You can create a rule that a result will be returned only when a majority of nodes agree. So, despite having a partition, the system will return a consistent result. However, since the separated node won’t be able to reach consensus it won’t be available even though it’s up.
3. Finally, a system can be both consistent and available, but it may have to block on a partition.

**Selecting Two Requirements at a Time**

|  |  |
| --- | --- |
| CA | All relational DBs are CA.  Single site clusters to ensure that all nodes are always in contact, e.g., 2 PCs.  When a partition occurs, the system blocks. |
| CP | Some data may be inaccessible (availability sacrificed), but the rest is still consistent/accurate, e.g., shared database. |
| AP | The system is still available under partitioning, but some returned data may be inaccurate, e.g., DNS, caches, Master/Slave replication.  Needs a conflict resolution strategy. |

Databases and CAP Theorem

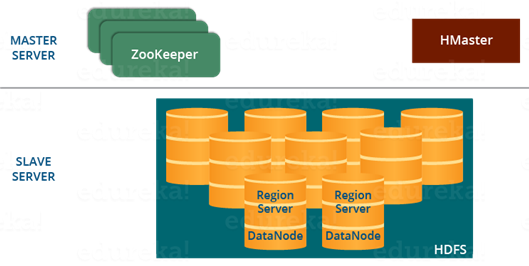
The following diagram depicts where each database falls and which two of the three features they select.



## HBase Architecture

HBase has three major components i.e., **HMaster Server**, **HBase Region Server**, **Regions and Zookeeper**.

The below figure explains the hierarchy of the HBase Architecture.



Before going to the HMaster, let’s understand Regions as all these Servers (HMaster, Region Server, and Zookeeper) are placed to coordinate and manage Regions and perform various operations inside the Regions.

**HBase Architecture: Region**

A region contains all the rows between the start key and the end key assigned to that region. HBase tables can be divided into a number of regions in such a way that all the columns of a column family is stored in one region. Each region contains the rows in a sorted order.

Many regions are assigned to a Region Server, which is responsible for handling, managing, executing reads and writes operations on that set of regions.

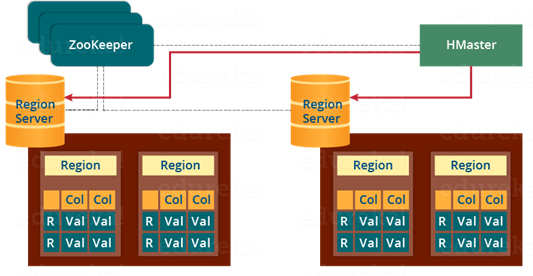
So, concluding in a simpler way:

1. A table can be divided into a number of regions. A Region is a sorted range of rows storing data between a start key and an end key.
2. A Region has a default size of 256MB which can be configured according to the need.
3. A Group of regions is served to the clients by a Region Server.
4. A Region Server can serve approximately 1000 regions to the client.

Now starting from the top of the hierarchy, let’s look at the HMaster Server which acts similarly as a NameNode in **HDFS**. Then, moving down in the hierarchy, I will take you through ZooKeeper and Region Server.

**HBase Architecture: HMaster**

As in the below image, you can see the HMaster handles a collection of Region Server which resides on DataNode. Let us understand how HMaster does that.



HBase HMaster performs DDL operations (create and delete tables) and assigns regions to the Region servers as you can see in the above image.

It coordinates and manages the Region Server (similar as NameNode manages DataNode in HDFS).

It assigns regions to the Region Servers on startup and re-assigns regions to Region Servers during recovery and load balancing.

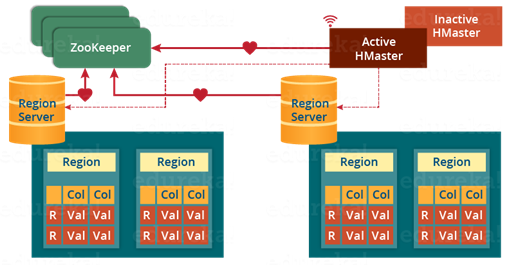
It monitors all the Region Server’s instances in the cluster (with the help of Zookeeper) and performs recovery activities whenever any Region Server is down.

It provides an interface for creating, deleting and updating tables.

HBase has a distributed and huge environment where HMaster alone is not sufficient to manage everything. So, you would be wondering what helps HMaster to manage this huge environment? That’s where ZooKeeper comes into the picture. After we understood how HMaster manages HBase environment, we will understand how Zookeeper helps HMaster in managing the environment.

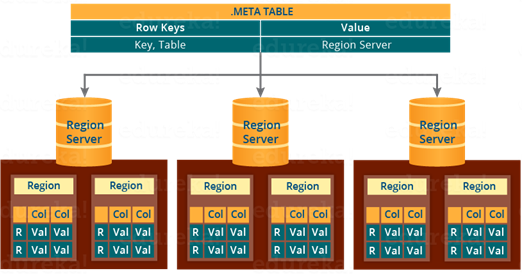
**HBase Architecture: ZooKeeper – The Coordinator**

This below image explains the ZooKeeper’s coordination mechanism.



* Zookeeper acts like a coordinator inside HBase distributed environment. It helps in maintaining server state inside the cluster by communicating through sessions.
* Every Region Server along with HMaster Server sends continuous heartbeat at regular interval to Zookeeper and it checks which server is alive and available as mentioned in above image. It also provides server failure notifications so that, recovery measures can be executed.
* Referring from the above image you can see, there is an inactive server, which acts as a backup for active server. If the active server fails, it comes for the rescue.
* The active HMaster sends heartbeats to the Zookeeper while the inactive HMaster listens for the notification send by active HMaster. If the active HMaster fails to send a heartbeat the session is deleted and the inactive HMaster becomes active.
* While if a Region Server fails to send a heartbeat, the session is expired and all listeners are notified about it. Then HMaster performs suitable recovery actions which we will discuss later in this blog.
* Zookeeper also maintains the .META Server’s path, which helps any client in searching for any region. The Client first has to check with .META Server in which Region Server a region belongs, and it gets the path of that Region Server.

**HBase Architecture: Meta Table**

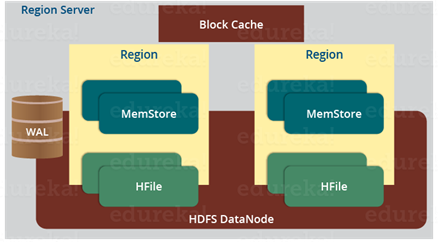


The META table is a special HBase catalog table. It maintains a list of all the Regions Servers in the HBase storage system, as you can see in the above image.

Looking at the figure you can see, .META file maintains the table in form of keys and values. Key represents the start key of the region and its id whereas the value contains the path of the Region Server.

**HBase Architecture: Components of Region Server**

This below image shows the components of a Region Server.



A Region Server maintains various regions running on the top of [*HDFS*](https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/). Components of a Region Server are:

* WAL: As you can conclude from the above image, Write Ahead Log (WAL) is a file attached to every Region Server inside the distributed environment. The WAL stores the new data that hasn’t been persisted or committed to the permanent storage. It is used in case of failure to recover the data sets.
* Block Cache: From the above image, it is clearly visible that Block Cache resides in the top of Region Server. It stores the frequently read data in the memory. If the data in BlockCache is least recently used, then that data is removed from BlockCache.
* MemStore: It is the write cache. It stores all the incoming data before committing it to the disk or permanent memory. There is one MemStore for each column family in a region. As you can see in the image, there are multiple MemStores for a region because each region contains multiple column families. The data is sorted in lexicographical order before committing it to the disk.
* HFile: From the above figure you can see HFile is stored on HDFS. Thus it stores the actual cells on the disk. MemStore commits the data to HFile when the size of MemStore exceeds.

**HBase Architecture: How Search Initializes in HBase?**

As discussed, Zookeeper stores the META table location. Whenever a client approaches with a read or writes requests to HBase following operation occurs:

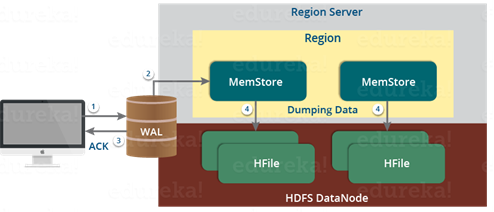
* The client retrieves the location of the META table from the ZooKeeper.
* The client then requests for the location of the Region Server of corresponding row key from the META table to access it. The client caches this information with the location of the META Table.
* Then it will get the row location by requesting from the corresponding Region Server.

For future references, the client uses its cache to retrieve the location of META table and previously read row key’s Region Server. Then the client will not refer to the META table, until and unless there is a miss because the region is shifted or moved. Then it will again request to the META server and update the cache.

As every time, clients does not waste time in retrieving the location of Region Server from META Server, thus, this saves time and makes the search process faster.

**HBase Architecture: HBase Write Mechanism**

This below image explains the write mechanism in HBase.



The write mechanism goes through the following process sequentially (see the above image):

Step 1: Whenever the client has a write request, the client writes the data to the WAL (Write Ahead Log).

* The edits are then appended at the end of the WAL file.
* This WAL file is maintained in every Region Server and Region Server uses it to recover data which is not committed to the disk.

Step 2: Once data is written to the WAL, then it is copied to the MemStore.

Step 3: Once the data is placed in MemStore, then the client receives the acknowledgment.

Step 4: When the MemStore reaches the threshold, it dumps or commits the data into a HFile.

## 

**HBase Write Mechanism- MemStore**

* The MemStore always updates the data stored in it, in a lexicographical order (sequentially in a dictionary manner) as sorted KeyValues. There is one MemStore for each column family, and thus the updates are stored in a sorted manner for each column family.
* When the MemStore reaches the threshold, it dumps all the data into a new HFile in a sorted manner. This HFile is stored in HDFS. HBase contains multiple HFiles for each Column Family.
* Over time, the number of HFile grows as MemStore dumps the data.
* MemStore also saves the last written sequence number, so Master Server and MemStore both knows, that what is committed so far and where to start from. When region starts up, the last sequence number is read, and from that number, new edits start.

The HFile is the main persistent storage in an HBase architecture. At last, all the data is committed to HFile which is the permanent storage of HBase. Hence, let us look at the properties of HFile which makes it faster for search while reading and writing.

**HBase Architecture: HBase Write Mechanism- HFile**

* The writes are placed sequentially on the disk. Therefore, the movement of the disk’s read-write head is very less. This makes write and search mechanism very fast.
* The HFile indexes are loaded in memory whenever an HFile is opened. This helps in finding a record in a single seek.
* The trailer is a pointer which points to the HFile’s meta block. It is written at the end of the committed file. It contains information about timestamp and bloom filters.
* Bloom Filter helps in searching key value pairs, it skips the file which does not contain the required rowkey. Timestamp also helps in searching a version of the file, it helps in skipping the data.

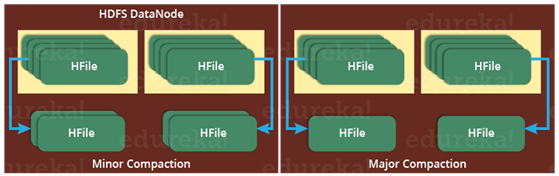
**HBase Architecture: Read Mechanism**

As discussed in our search mechanism, first the client retrieves the location of the Region Server from .META Server if the client does not have it in its cache memory. Then it goes through the sequential steps as follows:

* For reading the data, the scanner first looks for the Row cell in Block cache. Here all the recently read key value pairs are stored.
* If Scanner fails to find the required result, it moves to the MemStore, as we know this is the write cache memory. There, it searches for the most recently written files, which has not been dumped yet in HFile.
* At last, it will use bloom filters and block cache to load the data from HFile.

So far, we have discussed search, read and write mechanism of HBase. The HBase mechanism which makes search, read and write quick in HBase. First, we will understand Compaction, which is one of those mechanisms.

**HBase Architecture: Compaction**

****

**HBase** combines HFiles to reduce the storage and reduce the number of disk seeks needed for a read. This process is called **compaction**. Compaction chooses some HFiles from a region and combines them. There are two types of compaction as you can see in the above image.

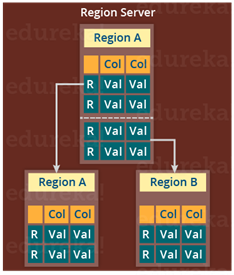
1. Minor Compaction: HBase automatically picks smaller HFiles and recommits them to bigger HFiles as shown in the above image. This is called Minor Compaction. It performs merge sort for committing smaller HFiles to bigger HFiles. This helps in storage space optimization.
2. Major Compaction: As illustrated in the above image, in Major compaction, HBase merges and recommits the smaller HFiles of a region to a new HFile. In this process, the same column families are placed together in the new HFile. It drops deleted and expired cell in this process. It increases read performance.

But during this process, input-output disks and network traffic might get congested. This is known as write amplification. So, it is generally scheduled during low peak load timings.

Now another performance optimization process which will be discussed is Region Split. This is very important for load balancing.

**HBase Architecture: Region Split**

The below figure illustrates the Region Split mechanism.



Whenever a region becomes large, it is divided into two child regions, as shown in the above figure. Each region represents exactly a half of the parent region. Then this split is reported to the HMaster. This is handled by the same Region Server until the HMaster allocates them to a new Region Server for load balancing.

Moving down the line, last but the not least, I will explain you how does HBase recover data after a failure. As we know that Failure Recovery is a very important feature of HBase, thus lets us know how HBase recovers data after a failure.

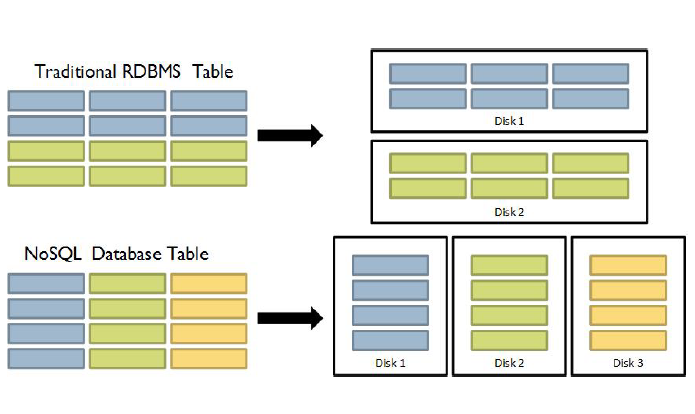
**HBase Architecture: HBase Crash and Data Recovery**

* Whenever a Region Server fails, ZooKeeper notifies to the HMaster about the failure.
* Then HMaster distributes and allocates the regions of crashed Region Server to many active Region Servers. To recover the data of the MemStore of the failed Region Server, the HMaster distributes the WAL to all the Region Servers.
* Each Region Server re-executes the WAL to build the MemStore for that failed region’s column family.
* The data is written in chronological order (in a timely order) in WAL. Therefore, Re-executing that WAL means making all the change that were made and stored in the MemStore file.
* So, after all the Region Servers executes the WAL, the MemStore data for all column family is recovered.

## HBase vs RDBMS

The differences between RDBMS and HBase are given below.

* Schema/Database in RDBMS can be compared to namespace in Hbase.
* A table in RDBMS can be compared to column family in Hbase.
* A record (after table joins) in RDBMS can be compared to a record in Hbase.
* A collection of tables in RDBMS can be compared to a table in Hbase.



**Comparison between H BASE and other column-oriented Database and the more traditional and RDBMS**.

|  |  |
| --- | --- |
| **H Base** | **RDBMS** |
| 1. Column-oriented | 1. Row-oriented(mostly) |
| 2. Flexible schema, add columns on the Fly | 2. Fixed schema |
| 3. Good with sparse tables. | 3. Not optimized for sparse tables. |
| 4. No query language | 4. SQL |
| 5. Wide tables | 5. Narrow tables |
| 6. Joins using MR – not optimized | 6. optimized for Joins(small, fast ones) |
| 7. Tight – Integration with MR | 7. Not really |
| 8. De-normalize your data. | 8. Normalize as you can |
| 9. Horizontal scalability-just add hard war. | 9. Hard to share and scale. |
| 10. Consistent | 10. Consistent |
| 11. No transactions. | 11. transactional |
| 12. Good for semi-structured data as well as structured data. | 12. Good for structured data. |