**Session 09: Hbase**

**Assignment 3 Question**

**Explain the below concepts with an example in brief.**

● **Nosql Databases**

A NoSQL database 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 were developed in response to the sheer volume of data being generated, stored and analyzed 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.

● **Types of Nosql Databases**

**Key-Value databases**

Key-value stores are the simplest NoSQL data stores to use from an API perspective. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.

Some of the popular key-value databases are [Riak](http://basho.com/riak/), [Redis](http://redis.io/) (often referred to as Data Structure server), [Memcached](http://memcached.org/) and its flavors, [Berkeley DB](http://www.oracle.com/technetwork/database/berkeleydb/index.html), [upscaledb](http://upscaledb.com/)(especially suited for embedded use), Amazon DynamoDB (not open-source), Project Voldemort and [Couchbase](http://www.couchbase.com/).

All key-value databases are not the same, there are major differences between these products, for example: Memcached data is not persistent while in Riak it is, these features are important when implementing certain solutions. Lets consider we need to implement caching of user preferences, implementing them in memcached means when the node goes down all the data is lost and needs to be refreshed from source system, if we store the same data in Riak we may not need to worry about losing data but we must also consider how to update stale data. Its important to not only choose a key-value database based on your requirements, it's also important to choose which key-value database.

**Document databases**

Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly the same. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable. Document databases such as MongoDB provide a rich query language and constructs such as database, indexes etc allowing for easier transition from relational databases.

Some of the popular document databases we have seen are [MongoDB](https://www.mongodb.org/), [CouchDB](http://couchdb.apache.org/) , [Terrastore](https://code.google.com/p/terrastore/), [OrientDB](http://www.orientechnologies.com/orientdb/), [RavenDB](http://ravendb.net/), and of course the well-known and often reviled Lotus Notes that uses document storage.

**Column family stores**

 Column-family databases store data in column families as rows that have many columns associated with a row key (Figure 10.1). Column families are groups of related data that is often accessed together. For a Customer, we would often access their Profile information at the same time, but not their Orders.

Each column family can be compared to a container of rows in an RDBMS table where the key identifies the row and the row consists of multiple columns. The difference is that various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows.

When a column consists of a map of columns, then we have a super column. A super column consists of a name and a value which is a map of columns. Think of a super column as a container of columns.

[Cassandra](http://www.datastax.com/) is one of the popular column-family databases; there are others, such as [HBase](https://hbase.apache.org/), [Hypertable](http://hypertable.org/), and Amazon DynamoDB. Cassandra can be described as fast and easily scalable with write operations spread across the cluster. The cluster does not have a master node, so any read and write can be handled by any node in the cluster.

**Graph Databases**

Graph databases allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application. Relations are known as edges that can have properties. Edges have directional significance; nodes are organized by relationships which allow you to find interesting patterns between the nodes. The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.

Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship ("who is my manager" is a common example). Adding another relationship to the mix usually means a lot of schema changes and data movement, which is not the case when we are using graph databases. Similarly, in relational databases we model the graph beforehand based on the Traversal we want; if the Traversal changes, the data will have to change.

In graph databases, traversing the joins or relationships is very fast. The relationship between nodes is not calculated at query time but is actually persisted as a relationship. Traversing persisted relationships is faster than calculating them for every query.

**● CAP Theorem**

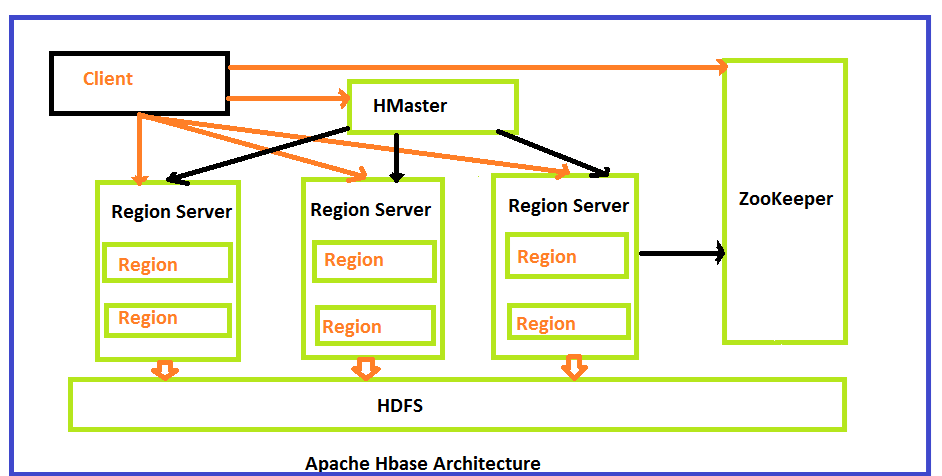
The CAP Theorem states that, in a distributed system (a collection of interconnected nodes that share data.), we can only have two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed.

* Consistency - A read is guaranteed to return the most recent write for a given client.
* Availability - A non-failing node will return a reasonable response within a reasonable amount of time (no error or timeout).
* Partition Tolerance - The system will continue to function when network partitions occur.

**CP** - Consistency/Partition Tolerance - Wait for a response from the partitioned node which could result in a timeout error. The system can also choose to return an error, depending on the scenario you desire. Choose Consistency over Availability when your business requirements dictate atomic reads and writes.

**AP** - Availability/Partition Tolerance - Return the most recent version of the data you have, which could be stale. This system state will also accept writes that can be processed later when the partition is resolved. Choose Availability over Consistency when your business requirements allow for some flexibility around when the data in the system synchronizes. Availability is also a compelling option when the system needs to continue to function in spite of external errors

**● HBase Architecture**



HBase architecture consists mainly of four components

* HMaster
* HRegionserver
* HRegions
* Zookeeper

**HMaster:**

HMaster is the implementation of Master server in HBase architecture. It acts like monitoring agent to monitor all Region Server instances present in the cluster and acts as an interface for all the metadata changes. In a distributed cluster environment, Master runs on NameNode

**HRegions Servers:**

When Region Server receives writes and read requests from the client, it assigns the request to a specific region, where actual column family resides. However, the client can directly contact with HRegion servers, there is no need of HMaster mandatory permission to the client regarding communication with HRegion servers. The client requires HMaster help when operations related to metadata and schema changes are required.

HRegionServer is the Region Server implementation. It is responsible for serving and managing regions or data that is present in distributed cluster. The region servers run on Data Nodes present in the Hadoop cluster

**HRegions:**

HRegions are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of Column families. It contains multiple stores, one for each column family. It consists of mainly two components, which are Memstore and Hfile

**ZooKeeper:**

In Hbase, Zookeeper is a centralized monitoring server which maintains configuration information and provides distributed synchronization. Distributed synchronization is to access the distributed applications running across the cluster with the responsibility of providing coordination services between nodes. If the client wants to communicate with regions, the servers client has to approach ZooKeeper first

**● HBase vs RDBMS**

|  |  |
| --- | --- |
| **HBase** | **RDBMS** |
| Column oriented | Row oriented |
| Flexible schema, columns can be added on the fly | Fixed schema |
| Joins using MR – not optimized | Optimized for joins |
| Supports horizontal scaling | Supports scaling up |
| Supports semi structured and unstructured data as well | Supports structured data |