**Assignment 9.3**

**Problem Statement:**

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

**1)** **Nosql Databases :**

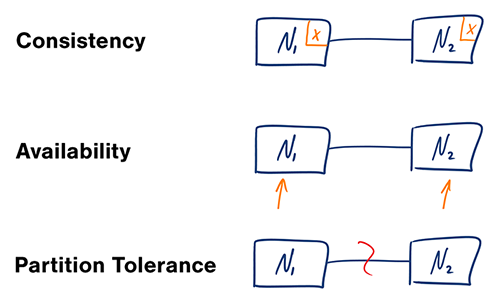
NoSQL is an approach to databases that represents a shift away from traditional relational database management systems (RDBMS). To define NoSQL, it is helpful to start by describing SQL, which is a query language used by RDBMS. Relational databases rely on tables, columns, rows, or schemas to organize 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.

**2) Types of Nosql Databases**

* **Key-value stores** – These databases pair keys to values.  An analogy is a files system where the path acts as the key and the contents act as the file.  There are usually no fields to update, instead, the entire value other than the key must be updated if changes are to be made.  The simplicity of this scales well but it can limit the complexity of the queries and other advanced features.  Examples are: [Dynamo](http://en.wikipedia.org/wiki/Dynamo_(storage_system)), [MemcacheDB](http://en.wikipedia.org/wiki/MemcacheDB" \o "MemcacheDB), [Redis](http://en.wikipedia.org/wiki/Redis" \o "Redis), [Riak](http://en.wikipedia.org/wiki/Riak" \o "Riak), FairCom [c-treeACE](http://en.wikipedia.org/wiki/C-treeACE), [Aerospike](http://en.wikipedia.org/wiki/Aerospike_database), [OrientDB](http://en.wikipedia.org/wiki/OrientDB" \o "OrientDB), [MUMPS](http://en.wikipedia.org/wiki/MUMPS), [HyperDex](http://en.wikipedia.org/wiki/HyperDex), [Azure Table Storage](http://azure.microsoft.com/en-us/services/storage/tables/) (see [Redis vs Azure](http://www.cunningplanning.com/?p=317))
* **Graph stores** – These excel at dealing with interconnected data.  Graph databases consist of connections, or edges, between nodes.  Both nodes and their edges can store additional properties such as key-value pairs.  The strength of a graph database is in traversing the connections between the nodes.  But they generally require all data to fit on one machine, limiting their scalability.  Examples include: [Allegro](http://en.wikipedia.org/wiki/AllegroGraph), [Neo4J](http://en.wikipedia.org/wiki/Neo4J), [InfiniteGraph](http://en.wikipedia.org/wiki/InfiniteGraph), [OrientDB](http://en.wikipedia.org/wiki/OrientDB), [Virtuoso](http://en.wikipedia.org/wiki/Virtuoso_Universal_Server), [Stardog](http://en.wikipedia.org/wiki/Stardog" \o "Stardog), Sesame
* **Column stores** – Relational databases store all the data in a particular table’s rows together on-disk, making retrieval of a particular row fast.  Column-family databases generally serialize all the values of a particular column together on-disk, which makes retrieval of a large amount of a specific attribute fast.  This approach lends itself well to aggregate queries and analytics scenarios where you might run range queries over a specific field.  Examples include: [Accumulo](http://en.wikipedia.org/wiki/Accumulo" \o "Accumulo), [Cassandra](http://en.wikipedia.org/wiki/Apache_Cassandra), [Druid](http://en.wikipedia.org/wiki/Druid_(open-source_data_store)), [HBase](http://en.wikipedia.org/wiki/HBase" \o "HBase), [Vertica](http://en.wikipedia.org/wiki/Vertica)
* **Document stores** – These databases store records as “documents” where a document can generally be thought of as a grouping of key-value pairs (it has nothing to do with storing actual documents such as a Word document).  Keys are always strings, and values can be stored as strings, numeric, Booleans, arrays, and other nested key-value pairs.  Values can be nested to arbitrary depths.  In a document database, each document carries its own schema — unlike an RDBMS, in which every row in a given table must have the same columns.  Examples include: [Lotus Notes](http://en.wikipedia.org/wiki/Lotus_Notes), [Clusterpoint](http://en.wikipedia.org/wiki/Clusterpoint" \o "Clusterpoint), [Apache CouchDB](http://en.wikipedia.org/wiki/Apache_CouchDB), [Couchbase](http://en.wikipedia.org/wiki/Couchbase), [MarkLogic](http://en.wikipedia.org/wiki/MarkLogic), [MongoDB](http://en.wikipedia.org/wiki/MongoDB), [OrientDB](http://en.wikipedia.org/wiki/OrientDB), [Qizx](http://en.wikipedia.org/wiki/Qizx), [Cloudant](http://en.wikipedia.org/wiki/Cloudant), Azure DocumentDB (see [MongoDB vs. Azure DocumentDB](http://justazure.com/mongodb-vs-azure-documentdb/) and [An Overview of Microsoft Azure DocumentDB](https://msdn.microsoft.com/en-us/magazine/mt147238.aspx))

**3) CAP Theorem**

The CAP Theorem states that, in a distributed system (a collection of interconnected nodes that share data.), you 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. However, as you will see below, you don’t have as many options here as you might think.



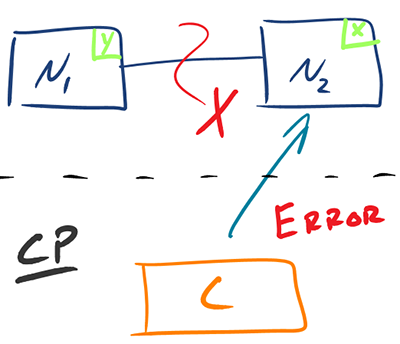
* **Consistency** - A read is guaranteed to return the most recent write for a given client.
* **Availability** - A non-failing node will return a reasonable response within a reasonable amount of time (no error or timeout).
* **Partition Tolerance** - The system will continue to function when network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

Before moving further, we need to set one thing straight. Object Oriented Programming != Network Programming! There are assumptions that we take for granted when building applications that share memory, which break down as soon as nodes are split across space and time.

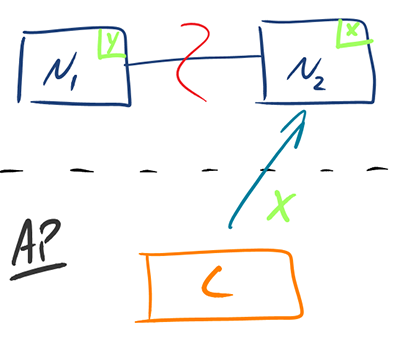
One such [*fallacy of distributed computing*](http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing) is that networks are reliable. They aren’t. Networks and parts of networks go down frequently and unexpectedly. Network failures *happen to your system* and you don’t get to choose when they occur.

Given that networks aren’t completely reliable, you must tolerate partitions in a distributed system, period. Fortunately, though, you get to choose what to do when a partition does occur. According to the CAP theorem, this means we are left with two options: Consistency and Availability.

* **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.

[](http://robertgreiner.com/uploads/images/2014/CAP-CP-full.png)

* **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 (shopping carts, etc.)

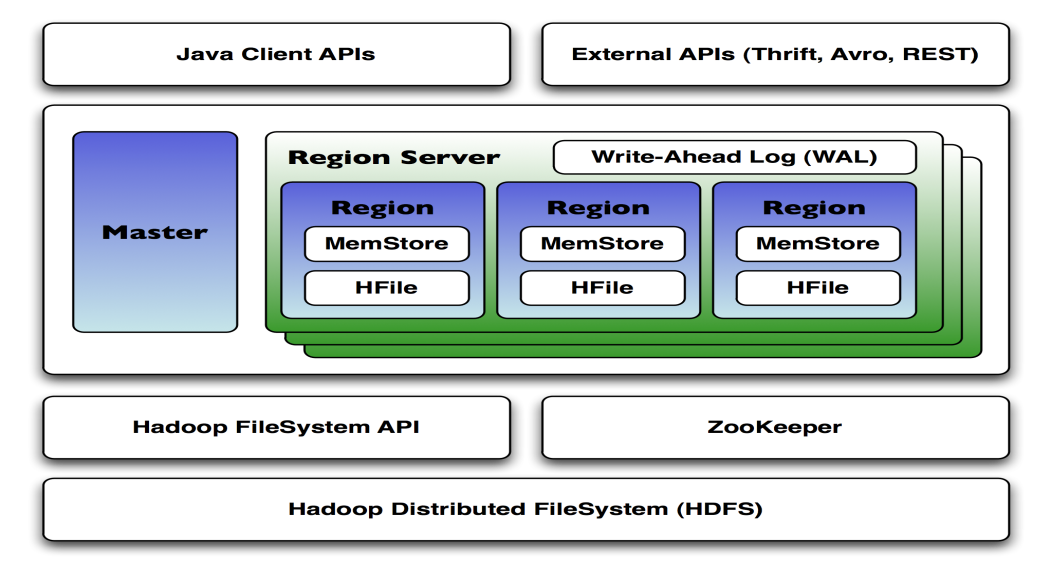
[](http://robertgreiner.com/uploads/images/2014/CAP-AP-full.png)

The decision between Consistency and Availability is a *software trade off*. You can choose what to do in the face of a network partition - the control is in your hands. Network outages, both temporary and permanent, are a fact of life and occur whether you want them to or not - this exists outside of your software.

Building distributed systems provide many advantages, but also adds complexity. Understanding the trade-offs available to you in the face of network errors, and choosing the right path is vital to the success of your application. Failing to get this right from the beginning could doom your application to failure before your first deployment.

**4) HBase Architecture**

HBase is a distributed database, meaning it is designed to run on a cluster of dozens to possibly thousands or more servers. As a result it is more complicated to install than a single RDBMS running on a single server. And all the typical problems of distributed computing begin to come into play such as coordination and management of remote processes, locking, data distribution, network latency and number of round trips between servers. Fortunately HBase makes use of several other mature technologies, such as Apache Hadoop and Apache ZooKeeper, to solve many of these issues. The figure below shows the major architectural components in HBase.



In the above figure you can see there is a single HBase master node and multiple region servers. (Note that it is possible to run HBase in a multiple master setup, in which there is a single active master.) HBase tables are partitioned into multiple regions with each region storing a range of the table's rows, and multiple regions are assigned by the master to a region server.

HBase is a column-oriented data store, meaning it stores data by columns rather than by rows. This makes certain data access patterns much less expensive than with traditional row-oriented relational database systems. For example, in HBase if there is no data for a given column family, it simply does not store anything at all; contrast this with a relational database which must store null values explicitly. In addition, when retrieving data in HBase, you should only ask for the specific column families you need; because there can literally be millions of columns in a given row, you need to make sure you ask only for the data you actually need.

HBase utilizes [ZooKeeper](http://www.sleberknight.com/blog/sleberkn/entry/distributed_coordination_with_zookeeper_part) (a distributed coordination service) to manage region assignments to region servers, and to recover from region server crashes by loading the crashed region server's regions onto other functioning region servers.

Regions contain an in-memory data store (MemStore) and a persistent data store (HFile), and all regions on a region server share a reference to the write-ahead log (WAL) which is used to store new data that hasn't yet been persisted to permanent storage and to recover from region server crashes. Each region holds a specific range of row keys, and when a region exceeds a configurable size, HBase automatically splits the region into two child regions, which is the key to scaling HBase.

As a table grows, more and more regions are created and spread across the entire cluster. When clients request a specific row key or scan a range of row keys, HBase tells them the regions on which those keys exist, and the clients then communicate directly with the region servers where those regions exist. This design minimizes the number of disk seeks required to find any given row, and optimizes HBase toward disk transfer when returning data. This is in contrast to relational databases, which might need to do a large number of disk seeks before transferring data from disk, even with indexes.

The HDFS component is the Hadoop Distributed Filesystem, a distributed, fault-tolerant and scalable filesystem which guards against data loss by dividing files into blocks and spreading them across the cluster; it is where HBase actually stores data. Strictly speaking the persistent storage can be anything that implements the Hadoop FileSystem API, but usually HBase is deployed onto Hadoop clusters running HDFS. In fact, when you first download and install HBase on a single machine, it uses the local filesystem until you change the configuration!

Clients interact with HBase via one of several available APIs, including a native Java API as well as a REST-based interface and several RPC interfaces (Apache Thrift, Apache Avro). You can also use DSLs to HBase from Groovy, Jython, and Scala.

**5) HBase vs RDBMS**

|  |  |
| --- | --- |
| **HBase** | **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 |