Unit # 3

- Unit test with MRUnit & Test data and local Test
- MemCached, Memcachedb
- TokyoCabinet, HBase,
- Zookeeper, Accumulo,
- SimpleDB, CouchDB

Unit test with MRUnit

- Writing a program in MapReduce follows a certain pattern. We start by writing the map and reduce functions, ideally then test it with unit tests to make sure they do what we expect. Then we proceed to write a driver program to run a job, which can run the map reduce functions as if it is running on a clusters environment. This uses the actual data which is small subset of the final data to check if the developed mapreduce functionality is working correctly or not.
- MRUnit is a testing framework specifically designed to facilitate unit testing for Hadoop MapReduce jobs. It allows developers to test the correctness of their Mapper, Reducer, and Driver logic without running the entire job on a cluster. This makes the testing process faster and more efficient during the development phase.
- When the program runs as expected against the small dataset, we are ready to host it on a cluster. Running against the full dataset is likely to expose some more issues, which we need to fix as before, i.e by expanding our tests and altering our mapper or reducer functions to handle the new cases. The core componenet of the MRUnit are :
  - ❑ MapDriver
  - ❑ ReduceDriver
  - ❑ MapReduceDrvier
  - ❑ Test Input and Output

MapDriver

MapDriver focuses on testing the map method /function or class of the Mapper in isolation by providing mock inputs and validating the generated outputs.

Unit test with MRUnit (contd)

- We know that in a MapReduce job, the Mapper transforms input data into intermediate key-value pairs. The MapDriver helps to verify that this transformation happens as expected.
- It test the Mapper logic without involving other components like the Reducer or even the Hadoop runtime environment (HDFS or YARN).
- It ensures that for a given input, the Mapper produces the correct set of intermediate key-value pairs as output.

ReduceDriver

- The ReduceDriver allows to test the Reducer in isolation, independent of other components like the Mapper or the Hadoop runtime.
- The Reducer is responsible for grouping and processing intermediate key-value pairs (grouped by key) received from the Mapper. ReduceDriver ensures this aggregation works as expected.

MapReduceDriver

- The MapReduceDriver is a crucial component of the MRUnit testing framework used for testing the entire MapReduce pipeline—both the Mapper and Reducer together.
- Ensure that the intermediate data output by the Mapper is correctly shuffled, sorted, and passed to the Reducer.
- Verify that the final output of the MapReduce job matches the expected results for a given input.

Unit test with MRUnit (contd)

Test Input and Output

- When testing a MapReduce job using the MRUnit framework, the key elements are the test input and test output. These represent the data we simulate for testing purposes, ensuring that the logic of our Mapper, Reducer, or the full MapReduce workflow behaves as expected. Understanding how test input and output work is essential for validating the correctness of the MapReduce logic during the development phase.
- In the context of MRUnit testing:
  - Test Input: Represents the data (key-value pairs) that is fed into the Mapper or Reducer.
  - Test Output: Represents the expected result (key-value pairs) produced by the Mapper, Reducer, or the complete MapReduce pipeline.
  - This is not the same

Memcached and Memcachedb

- **Memcached** and **Memcachedb** are distributed caching and key-value store systems, used to improve the performance of applications by storing frequently accessed data in memory. It store from a small arbitrary data (strings, objects) from results of database calls, API calls, or up to page rendering. It is a Free & open source and generic in nature, mainly used in speeding up dynamic web applications by alleviating database load.
- Although Memcached and Memcachedb share similar names and concepts (both are key-value stores), they serve distinct purposes and operate differently. While Memcached focuses on fast, in-memory caching to optimize data retrieval and intermediate processing, Memcachedb provides a persistent key-value store for scenarios requiring data durability.
- Both Memcached & Memcachedb can complement Hadoop workflows in various contexts. These different contexts are explained below .

Memcached applications

(1) Map Reduce operation:

We know that MapReduce jobs often have intermediate outputs from the Mapper phase that need to be shuffled and sorted before being consumed by the Reducer. If this intermediate data is repeatedly required across multiple iterations of a job or pipeline , recomputing or retrieving this data can be slow. Memcached are perfect solutions in suchs cenerio and it can store intermediate key-value pairs emitted by the Mapper. Also, if a Reducer needs the same data again (e.g., in an iterative job), it can fetch the data directly from Memcached instead of repeating the Mapper computation or waiting for shuffling from distributed storage like HDFS.

Memcached and Memcachedb (contd)

(2) HDFS operation:

In large-scale MapReduce workflows, Mapper tasks often read data from HDFS, which incurs disk I/O overhead.Memcached can accesss frequently or recently processed data from HDFS. Mapper tasks can access this cached data quickly, reducing the need for repeated reads from HDFS.

(3) Join operation:

Some MapReduce jobs require joins between datasets (e.g., joining user logs with user profile data). If one of the datasets is small and frequently reused, it can be loaded into Memcached. Mappers can then perform **in-memory lookups** on the cached dataset instead of repeatedly loading it from HDFS.

Memcachedb applications

While Memcached is used for volatile, in-memory caching, Memcachedb provides persistent key-value storage. The durability of Memcachedb makes it useful in scenarios where data persistence is crucial during or after a MapReduce job.

(1) Storing MapReduce job Results:

After a MapReduce job completes, its results (often written to HDFS) may need to be reused by downstream processes or subsequent jobs. Instead of writing intermediate results to HDFS (which is disk-heavy), they can be stored in Memcachedb, offering faster access while ensuring persistence. This is particularly useful for chained

Memcached and Memcachedb (contd)

MapReduce workflows, where the output of one job serves as input to another or storing the results of commonly queried data to avoid re-running MapReduce jobs for the same inputs.

(2) Integration in Iterative Workflows :

In algorithms like K-means clustering , intermediate results need to be stored and reused in multiple iterations. Memcachedb can provide persistence, ensuring that intermediate data is available even if the system fails or restarts. If data is saved in memcached then data in Memcached can get lost if a node crashes or restarts. Hence in certain scenarios , this may not be suitable for workflows requiring guaranteed data persistence across job stages and instead Memcachedb can be used here.

Tokyo Cabinet

- **Tokyo Cabinet** is a Single Node , high-speed, lightweight, and efficient database library designed for managing key-value data. Its chief characteristics is that it is super fast. To get an idea it just takes 0.7 seconds to store 1 million records in the regular hash table. It is written in C, with native support for C and C++ to achieve fast access.
- In Big Data pipelines, data needs to be transformed before being ingested into Hadoop HDFS. Tokyo Cabinet is used as a fast local store for intermediate data storage during these pre-processing or post-processing stages.
- Hadoop Distributed File System (HDFS) is designed for high-throughput rather than low-latency access. So if latency becomes important then Tokyo cabinet comes to rescue and critical hot data can be stored on Tokyo cabinet as local persistent store, while HDFS retains the complete dataset.
- Tokyo Cabinet can also work alongside other NoSQL databases like HBase or Cassandra, both of which are frequently used in Big Data environments.
- Tokyo Cabinet can improve the performance of Hadoop MapReduce, Spark, and other Big Data frameworks, especially in scenarios where fast local access and persistence are needed.

Hash Database Engine:

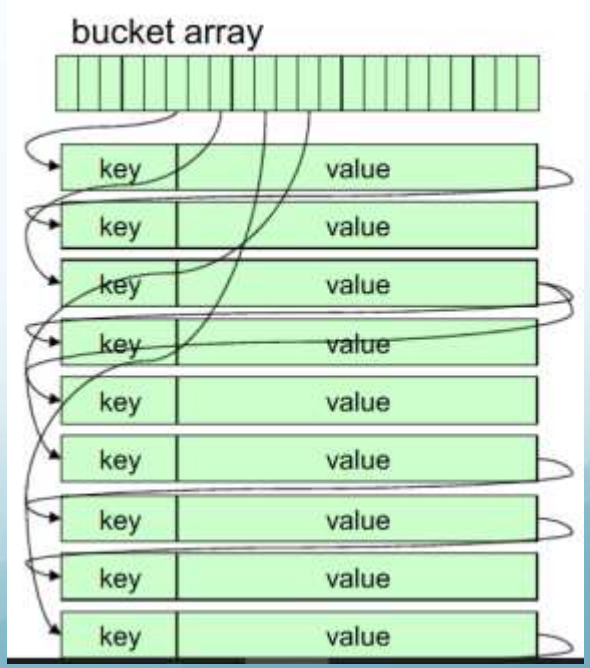
Tokyo Cabinet uses hash database engine that is a highly efficient key-value storage system and as name suggest uses a hash table for fast data retrieval. The working happens as follows.

Tokyo Cabinet (contd)

Working of Hash database Engine in Tokyo cabinet:

- When a key-value pair is inserted, the key is passed through a hashing function.
- The function computes a **hash code** that determines the location (bucket) where the value will be stored on disk. For example say the user enters the key value pair say
  - Key: "user:123“
  - Value: {“MCA”: “3rd Sem”, “Exams”: “EndSem”, “Date”: “10-Dec-2024”}
- The hash function computes a hash for "user:123" say Hashcode: 4567. The hash code is mapped to a specific bucket or slot on disk. A bucket corresponds to a region on disk.
- Each bucket contains the key , associated value and the meta data such as key length and value length etc.

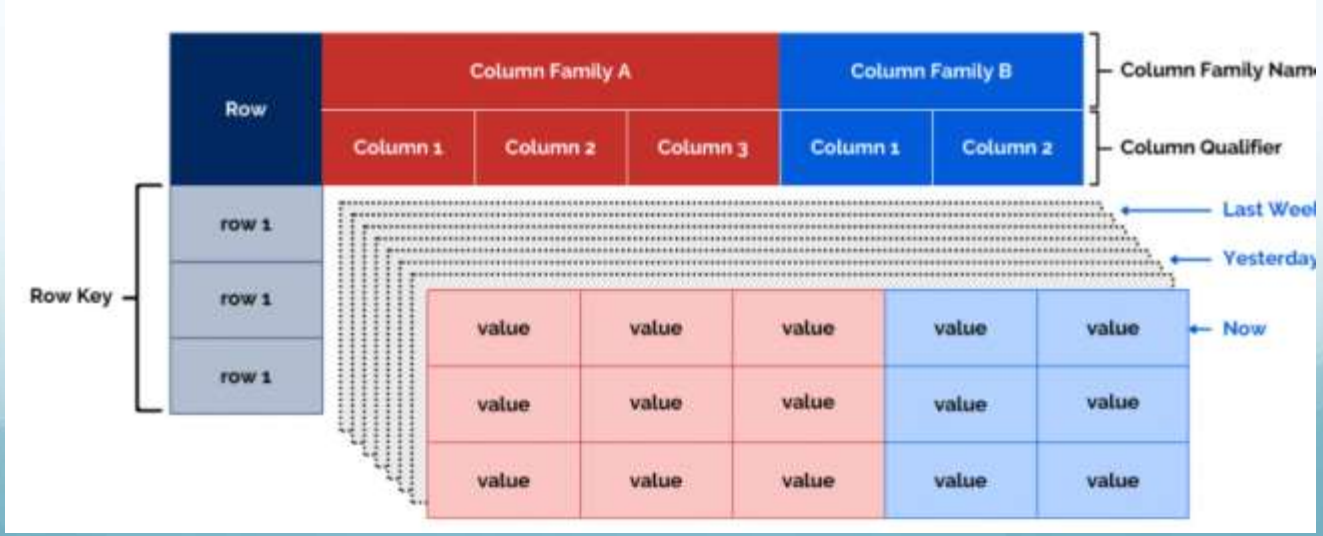
- If the hashing function generates a same hash code then through chaining method.
- Tokyo Cabinet supports multiple data models to handle different types of data storage needs:
  - ✔ Hash Database
  - ✔ B+ Tree Database
  - ✔ Fixed-Length Records Database
  - ✔ Table Database



HBase

HBase is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random access to very large datasets. modeled after Google's Bigtable (Refer PPT #3 , slide # 5 onwards) and the first HBase release was bundled as part of Hadoop 0.15.0 in 2007. By 2010, HBase graduated from a Hadoop subproject to become an Apache Top Level Project. Today, HBase is a mature technology used in production across a wide range of industries.

- HBase organizes data in a **column-family** format, where data is stored in rows and columns, grouped into column families. Unlike relational databases, columns can vary between rows, making HBase more flexible.
- Thus due to its architecture like Big Table , HBase doesn't enforce a fixed schema and can supports unstructured and semi-structured data, allowing different rows to have different sets of columns. Refer to slide# 13 in PPT-3

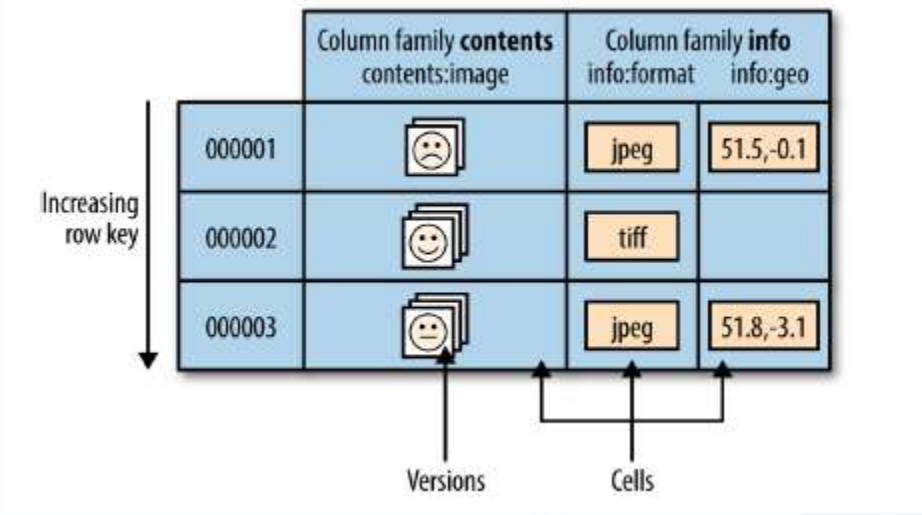


Hbase (contd)

An example HBase table for storing photos is shown in Figure below.

In this figure ,

- Row columns are grouped into column families. All column family members have a common prefix, so, for example, the columns ***info:format*** and ***info:geo*** are both members of the info column family, whereas ***contents:image*** belongs to the contents family.
- A table's column families must be specified up front as part of the table schema definition,but new column family members can be added on demand.



Region Server :

- A region is a subset of a table, comprising a range of rows. A region is ear marked by the table it belongs to, its first row , and its last row.
- Initially, a table comprises a single region, but as the region grows it eventually crosses a configurable size threshold, at which point it splits at a row boundary into two new regions of approximately equal size.
- As the table grows, the number of its regions grows. Regions are the units that get distributed over an HBase cluster. The regions are managed by what is known as Region Server.

Hbase (contd)

Master Server :

- The HBase Master oversees the overall functioning of the HBase cluster.
- It is responsible for managing metadata, monitoring region servers, and balancing the load across the cluster.
- It assigns regions to region servers , manages schema changes (e.g., creating or deleting tables) & Coordinates region server activities.

Key Features of Hbase

1. *Random and Real-Time Access:* HBase supports low-latency access for real-time analytics or applications.

1. *Fault Tolerance:* Built on HDFS, it benefits from Hadoop's replication and fault tolerance.

1. *Horizontal Scalability:* Scales by adding more region servers to distribute data and workloads.

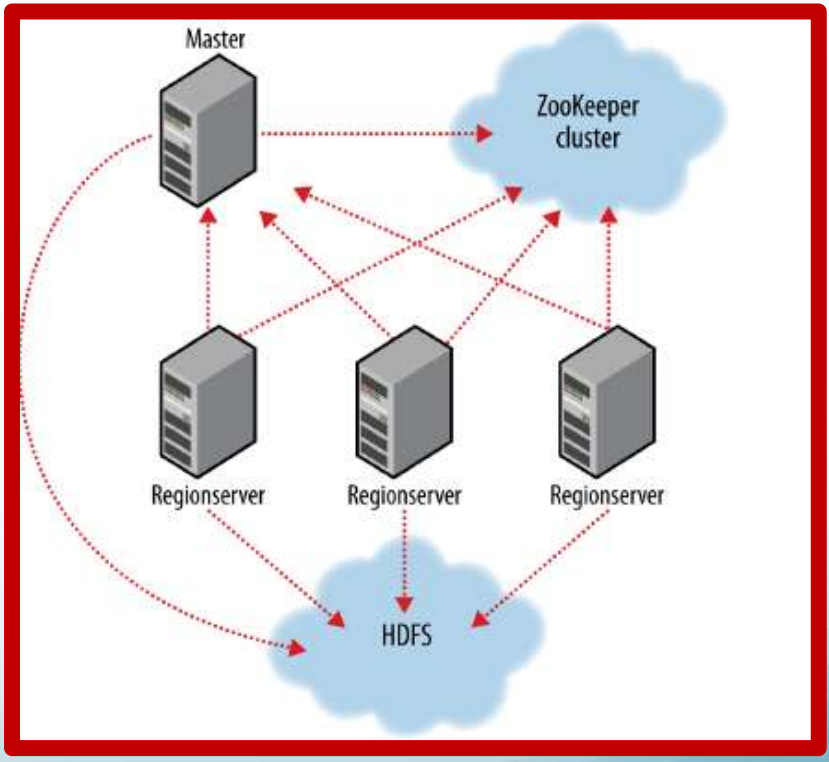
1. *Versioning:* Stores multiple versions of data for each cell, identified by timestamps.

1. *Flexible Schema:* Schema-less design allows for dynamic addition of columns, ideal for semi-structured data.

Zookeper

Apache ZooKeeper is a distributed coordination service designed to help manage and synchronize distributed systems. ZooKeeper is a critical component of the HBase ecosystem, providing the foundation for coordination, synchronization, and fault tolerance in distributed clusters. It ensures smooth functioning of HBase by handling master election, region server tracking, and region assignments and ensures different nodes or components in a cluster can work together consistently and handle failures gracefully. The placement of Zoomaker in Hbase ecosystem is as shown in following fig.

- When a message is sent across the network between two nodes and the network fails, the sender does not know whether the receiver got the message. It may have gotten through before the network failed, or it may not have as the network failed before it reached the other node.
- This particular scenario i.e when we don't even know if an operation failed is known as Partial failure.
- Apache ZooKeeper gives a set of tools to build distributed applications that can safely handle partial failures.



- HBase depends on ZooKeeper and by default it manages a ZooKeeper instance as the authority on clusters.

Zookeper (contd)

Specifically Hbase uses Zookeeper in following ways :

1. Distributed Coordination

Zookeeper helps Hbase maintaining synchronization between nodes in a distributed system. It provides tools to manage shared resources like configurations and metadata.

2. Leader Election

ZooKeeper help Hbase managing leader election among distributed nodes, ensuring only one node acts as the leader at any time. HBase uses ZooKeeper to elect the active HBase Master if there are multiple masters running for high availability. ZooKeeper shall ensures that only one HBase Master is active at a time, while the others remain in standby mode.

3. Region Server Registration

Region Servers register themselves with ZooKeeper when they start up. ZooKeeper helps track the **heartbeat** of Region Servers to ensure they are operational. Further , it helps the HBase Master balance the load by redistributing regions across Region Servers if needed. If a Region Server crashes, ZooKeeper notifies the HBase Master, which then reassigns the affected regions to other Region Servers.

4. Atomic Broadcast

Zookeeper helps maintaining atomicity between node. All updates to ZooKeeper's data are applied consistently across all nodes.

Accumulo , Simple DB & CouchDB

In the world of NoSQL databases, numerous systems have emerged to address different data storage and processing requirements of Big data applications.
Accumulo, SimpleDB, and CouchDB are three prominent databases, each catering to specific use cases but sharing some conceptual underpinnings. They are described below.

1. Accumulo

Accumulo is provided by Apache. It is a distributed, scalable, and secure key-value store built on top of the Hadoop Distributed File System (HDFS) in 2011. It was inspired by Google's Bigtable and is designed to provide advanced features like cell-level access control and server-side programming mechanisms.
Apache Accumulo's architecture consists of a few key components:
Tablet Server, Master, Garbage Collector, Monitor, and Client.

1. Tablet Server

Each table in Accumulo is split into multiple **tablets**, which represent contiguous ranges of rows. The Tablet Server is a core component responsible for storing and managing read and write operations for a set of tablets—horizontal partitions of tables. It handles Put (write) & Get (read) requests from clients , merges MemTable data with existing HFiles.

Accumulo (contd)

2. Master Server

The Master Server is the central coordinator of an Accumulo cluster. It oversees the health and distribution of tablets across Tablet Servers. It assigns tablets to Tablet Servers and ensures a balanced distribution of tablets across the cluster to prevent any single Tablet Server from being overloaded. During running state , it continuously monitors the cluster and redistributes tablets as necessary to achieve **load balancing**.

3. Garbage Collector

As the name suggest the Garbage Collector (GC) is responsible for cleaning up unused data files and Write-Ahead Logs (WALs) to free up disk space. The GC identifies these **orphaned files** and removes them from HDFS.

4. Monitor

The Accumulo Monitor provides a web-based interface to visualize the health and performance of the Accumulo cluster. It displays the status of the **Master** and **Tablet Servers**, including their load, memory usage, and number of assigned tablets. It also provides metrics such as read/write latency, throughput, and the number of operations performed.

4. Client

The Client is the interface that allows external applications to interact with Accumulo. It provides an API for performing read ,write and to configure Iterators operations.

Simple DB

SimpleDB is provided by Amazon AWS . It is a highly available, eventually consistent, and fully managed NoSQL database for small to medium-scale applications. Its emphasis on simplicity, high availability, and automatic scaling, makes it an ideal choice for lightweight applications. Its Key features are described below :

1. Schema Free design

SimpleDB supports dynamic, schema-less data models, allowing users to add attributes without predefined structures. We can store data with different attributes for each item, providing flexibility in handling semi-structured or unstructured data.

2. Eventual Consistency

SimpleDB provides eventual consistency, ensuring that updates are propagated to all replicas, but not necessarily immediately. It also offers strong consistency for read operations if required, ensuring the most up-to-date data.

3. High Availability

SimpleDB automatically replicate data across multiple availability zones (AZs) for fault tolerance and high availability. automatically indexes all attributes, enabling fast query execution without requiring manual index management.

2. Cost-Effective

SimpleDB is designed for low-cost data storage, with a pricing model based on the amount of data stored & number of read/write operations & the amount of data transfer.

CouchDB

- Apache CouchDB is an open-source NoSQL database designed for storing, retrieving, and managing large amounts of semi-structured or unstructured data. It emphasizes ease of **use** , **scalability**, and **robust replication** across distributed systems.
- CouchDB is unique among NoSQL databases due to its open access. While simpleDB is available through the AWS API and Accumulo depends on HDFS. CouchDB stands out as it is open source, free to use, and easily integrates within current data management infrastructure .

The key characteristics of CouchDB is explained below :

1. Data Oriented Storage

The fundamental data unit in CouchDB is a JSON document. Each document has a unique **_id** field (analogous to a primary key in relational databases) and a **_rev** field (revision identifier). The documents can store a variety of data types, including strings, numbers, arrays, and nested JSON objects. This makes CouchDB highly flexible, as each document can have its own structure without requiring a predefined schema.

2. Eventual Consistency and Conflict Resolution

CouchDB ensure all replicas on multiple nodes converge to the same state over time. It provides automatic conflict detection and allows developers to handle conflicts through custom resolution logic. When multiple nodes modify the same document, CouchDB detects **conflicts**.

CouchDB (contd)

3. High Availability and Fault Tolerance

CouchDB is designed for distributed environments and provides built-in fault tolerance through replication. If a node fails, another replica can take over seamlessly.

4. CouchDB Server

The CouchDB server runs as a standalone service and manages databases, documents, and replication. It listens for incoming HTTP requests on a specific port and serves them using its RESTful API. The developers can interact with the database using standard HTTP methods such as GET, POST, PUT, and DELETE.

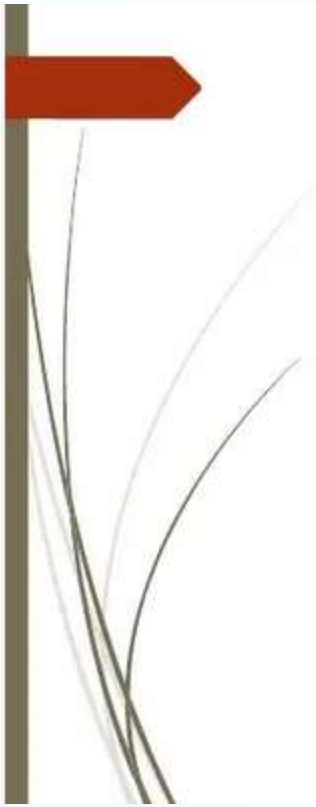
Comparison
Accumulo , Simple DB & CouchDB

Conclusion and Comparison :

Although Accumulo, SimpleDB, and CouchDB serve different primary purposes, they share common goals of handling large-scale, distributed data efficiently. They can complement each other in a variety of scenarios, especially in complex systems requiring multiple types of data storage and processing.

- All three databases fall under the **NoSQL paradigm**, to address scalability, and the ability to handle semi-structured or unstructured data.
  - Accumulo: Optimized for key-value pairs with strong consistency.
  - SimpleDB: Focuses on flexibility and high availability.
  - CouchDB: Emphasizes document-oriented storage and multi-master replication.

- All three databases are designed for **distributed environments**, ensuring that they can scale horizontally as data volumes grow.
  - Accumulo leverages Hadoop and HDFS for distributed storage.
  - SimpleDB uses Amazon's cloud infrastructure to provide high availability and fault tolerance.
  - CouchDB employs multi-master replication to synchronize data across geographically distributed nodes.



Thanks