Chapter 4: NoSQL Databases

Types of Data

Data, in the context of databases and information systems, can be broadly categorized in several ways based on its structure and nature. Here are some common classifications:

1. Structured Data: This is the most traditional type of data, highly organized and conforming to a fixed schema. It typically resides in relational databases where data is stored in tables with predefined columns and rows. Examples include numbers, dates, strings, and boolean values, all arranged in a rigid format. This structure allows for easy querying and analysis using languages like SQL.

2. Semi-structured Data: This type of data has some organizational properties but does not conform to a strict, fixed schema like structured data. It may contain tags or markers to separate semantic elements, and the hierarchy of the data can be irregular or change. Examples include XML, JSON, and other markup languages. While it has structure, it's more flexible than relational data.

3. Unstructured Data: This is data that has no predefined structure or organization. It does not fit into a traditional row-and-column database. Examples include text documents, images, audio and video files, social media posts, and emails. Extracting information from unstructured data often requires advanced techniques like natural language processing or machine learning.

Beyond Structure, Data can also be classified by its nature:

- **Qualitative Data:** This data describes qualities or characteristics and cannot be measured numerically. It is often subjective and exploratory.
  - **Nominal Data:** Categorical data without any inherent order or ranking (e.g., colors, gender, types of cars).
  - **Ordinal Data:** Categorical data with a defined order or ranking, but the intervals between categories are not uniform or meaningful (e.g., satisfaction ratings like "low," "medium," "high"; educational levels).
- **Quantitative Data:** This data represents quantities and can be measured numerically.
  - **Discrete Data:** Numerical data that can only take specific, distinct values, usually whole numbers. There are gaps between possible values (e.g., number of students in a class, number of cars in a parking lot).
  - **Continuous Data:** Numerical data that can take any value within a given range. The values can be infinitely divided (e.g., height, weight, temperature, time).

Introduction to NoSQL

NoSQL, often interpreted as "Not Only SQL," refers to a diverse group of non-relational database technologies. Unlike traditional relational databases that use fixed schemas and store data in tables with rows and columns, NoSQL databases provide flexible schemas and employ various data models for storing and managing data.

The rise of NoSQL databases was driven by the need to handle the challenges posed by modern web applications, big data, and real-time systems, which often involve massive volumes of diverse and rapidly changing data that don't fit well into the rigid structure of relational databases.

Key characteristics of NoSQL databases include:

- **Flexible Schemas:** They can handle semi-structured and unstructured data without requiring a predefined schema. This allows for easier and faster development, especially in agile environments.
- **Horizontal Scalability:** Many NoSQL databases are designed to scale horizontally across multiple servers or nodes, making it easier and more cost-effective to handle increasing data volumes and traffic compared to the vertical scaling (upgrading a single server) typical of many relational databases.
- **High Availability and Fault Tolerance:** Distributed NoSQL systems are often designed with replication and distribution mechanisms to ensure data availability and system resilience in case of node failures.
- **Optimized for Specific Data Models:** Different types of NoSQL databases are optimized for particular data structures and access patterns, offering better performance for specific use cases.

Need of NoSQL

The need for NoSQL databases arose primarily due to the limitations of traditional relational databases in addressing the requirements of modern applications and the challenges of managing massive datasets. Here are some key reasons why NoSQL is needed:

- **Handling Big Data:** Relational databases can struggle with the sheer volume, velocity, and variety of data generated by today's applications. NoSQL databases are built to handle large-scale distributed datasets efficiently.
- **Flexibility for Evolving Data:** In agile development environments, data requirements can change frequently. The rigid schema of relational databases makes modifications difficult and time-consuming. NoSQL's flexible schemas allow for easier adaptation to changing data structures.
- **Scalability Demands:** Modern applications often require the ability to scale rapidly and cost-effectively to accommodate a growing number of users and data. NoSQL databases are designed for horizontal scaling, distributing data and load across multiple servers.
- **Performance for Specific Workloads:** For certain types of operations, such as key-value lookups or handling graph-like data, specific NoSQL database types can offer significantly better performance than relational databases.
- **Support for Diverse Data Types:** NoSQL databases are well-suited for storing and managing various data formats, including structured, semi-structured, and unstructured data, which is common in web content, social media, and IoT data.
- **Cloud Computing and Distributed Systems:** NoSQL databases are often designed with distributed architectures in mind, making them a natural fit for cloud deployments and distributed applications.

Types of NoSQL Databases

NoSQL databases are not a single technology but rather a category encompassing various database technologies, each with a different data model and optimized for specific use cases. The four main types are:

1. Key-Value Stores: This is the simplest type of NoSQL database. Data is stored as a collection of key-value pairs, where each unique key is associated with a value. The value can be a simple data type (like a string or number) or a more complex object. Key-value stores offer high performance for read and write operations based on the key.

- **How they work:** Data is accessed by providing a key to retrieve the associated value. There is no concept of tables or relationships in the relational sense.
- **Use Cases:** Caching, session management, user profiles, shopping carts.
- **Examples:** Redis, Memcached, Amazon DynamoDB (can operate as a key-value store).

2. Document Databases: Document databases store data in flexible, semi-structured units called documents. These documents are typically in formats like JSON, BSON, or XML. Each document can have a different structure, and nested data structures are easily supported. Document databases offer rich query capabilities within the documents.

- **How they work:** Data is organized into collections (similar to tables), and each item in a collection is a document. Queries can be performed based on the content of the documents.
- **Use Cases:** Content management systems, blogging platforms, e-commerce product catalogs, user profiles with varying attributes.
- **Examples:** MongoDB, Couchbase, Amazon DocumentDB.

3. Wide-Column Stores (Column-Oriented Databases): Unlike relational databases that store data row by row, wide-column stores store data in columns. Data is organized into column families, and within each row, you can have varying columns. This structure is optimized for queries that access specific columns across many rows, making them suitable for analytical workloads and time-series data.

- **How they work:** Data is stored in families of columns. Reading data involves accessing only the required columns, which can be very efficient for certain types of queries.
- **Use Cases:** Time-series data, analytical databases, IoT data management, large-scale data warehousing.
- **Examples:** Apache Cassandra, HBase, Google Bigtable.

4. Graph Databases: Graph databases are designed to store and traverse relationships between data entities efficiently. Data is represented as nodes (entities) and edges (relationships between nodes). Both nodes and edges can have properties. This model is ideal for applications where the connections between data points are as important as the data itself.

- **How they work:** Data is stored as a network of interconnected nodes and edges. Queries traverse these connections to find patterns and relationships.
- **Use Cases:** Social networks, recommendation engines, fraud detection, knowledge graphs, network management.
- **Examples:** Neo4j, Amazon Neptune, ArangoDB (multi-model, includes graph).

NoSQL vs. Relational Databases

Here's a comparison highlighting the key differences between NoSQL and Relational databases:

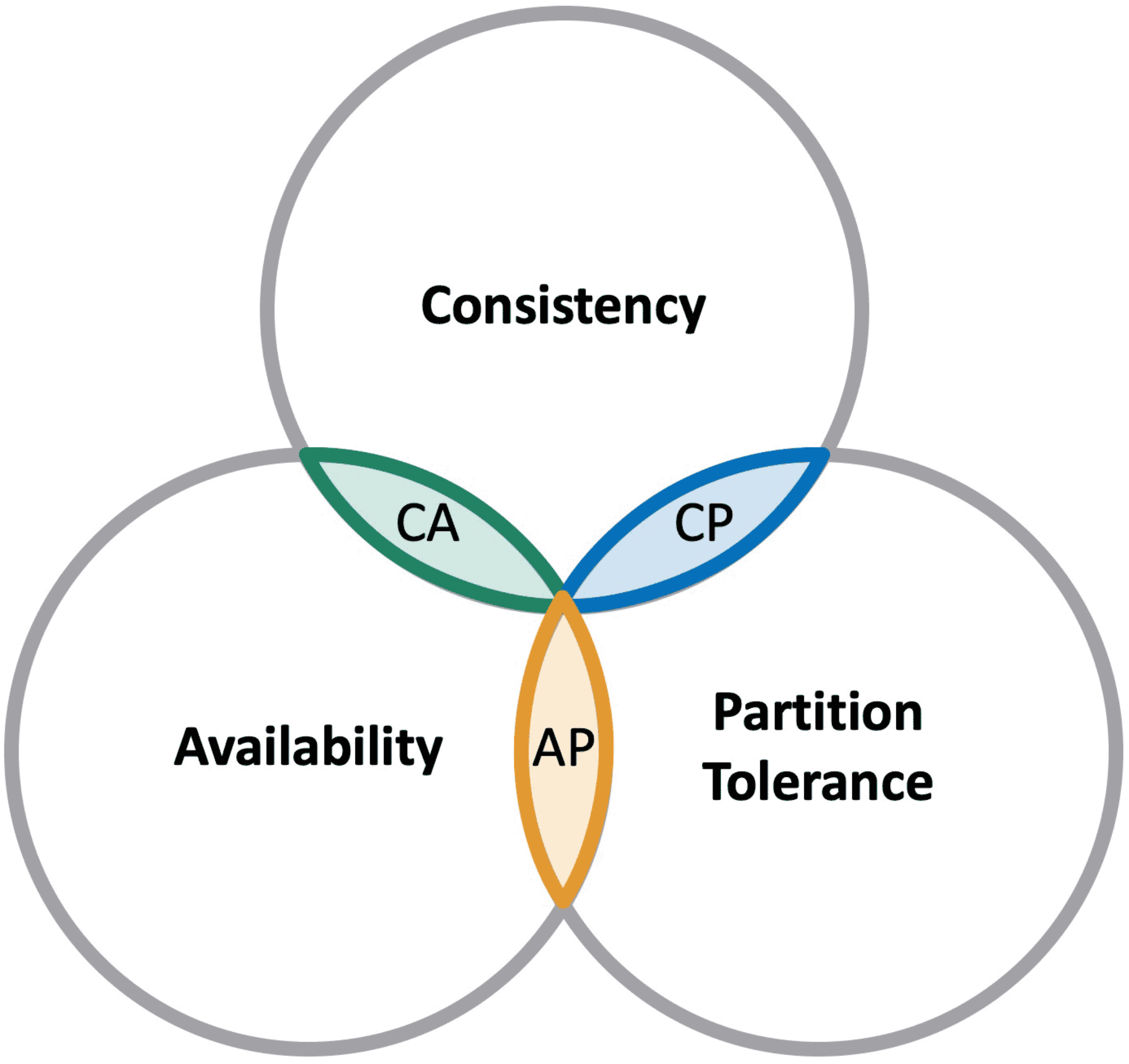
Feature	Relational Databases (SQL)	NoSQL Databases
Data Model	Tabular (tables with fixed rows and columns)	Various (Document, Key-Value, Column-Family, Graph)
Schema	Rigid, predefined schema	Flexible schema, schema-less (for some types)

Feature	Relational Databases (SQL)	NoSQL Databases
Query Language	SQL (Structured Query Language)	Various (query languages depend on the database type, some use APIs)
Scalability	Primarily vertical (scale up), horizontal scaling is complex (sharding)	Primarily horizontal (scale out)
Consistency	ACID (Atomicity, Consistency, Isolation, Durability) - Strong Consistency	BASE (Basically Available, Soft State, Eventually Consistent) - Eventual Consistency is common, some offer stronger consistency options
Data Handling	Best for structured data	Best for structured, semi-structured, and unstructured data
Relationships	Explicitly defined using foreign keys and joins	Relationships are modeled differently depending on the database type (e.g., embedded documents, links, edges in graphs)
Complexity	Can become complex with many tables and joins	Can be simpler for certain data models, but managing consistency in distributed systems can be complex
Use Cases	Transaction processing, financial systems, applications requiring high data integrity	Big data, real-time web applications, content management, mobile apps, IoT, social networks

The CAP Theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental concept in distributed systems. It states that a distributed database system cannot simultaneously guarantee all three of the following properties:

- **Consistency (C):** Every read request receives the most recent write or an error. In a consistent system, all nodes have the same view of the data at any given time.
- **Availability (A):** Every request receives a response, without guarantee that it contains the most recent write. The system remains operational and responsive to queries, even if some nodes are down.
- **Partition Tolerance (P):** The system continues to operate despite network partitions. A network partition occurs when a communication breakdown divides the distributed system into multiple isolated subgroups of nodes that cannot communicate with each other.



The CAP theorem states that in the presence of a network partition (P), a distributed system must choose between Consistency (C) and Availability (A). It's impossible to guarantee both simultaneously during a partition.

- **CP System (Consistency and Partition Tolerance):** In a CP system, if a network partition occurs, the system will prioritize consistency. If a node cannot synchronize with other nodes to ensure it has the latest data, it will become unavailable to avoid returning potentially stale data. This means some parts of the system might be inaccessible during a partition, but the data served will be consistent.
- **AP System (Availability and Partition Tolerance):** In an AP system, if a network partition occurs, the system will prioritize availability. Nodes will continue to respond to requests even if they cannot communicate with other nodes. This might result in returning data that is not the most up-to-date, leading to eventual consistency (data will become consistent once the partition is resolved).

Partition Tolerance (P) is generally considered a mandatory requirement for any distributed system, as network failures are inevitable. Therefore, the practical implication of the CAP theorem is that when designing a distributed database, you must decide whether to prioritize consistency or availability during a network partition.

Relational databases, traditionally designed for single servers or tightly coupled clusters, often prioritize consistency (ACID). Many NoSQL databases, designed for distributed environments, often lean towards availability and partition tolerance (BASE), accepting eventual consistency. The choice between prioritizing consistency or availability depends on the specific requirements and use case of the application.

HBase

Apache HBase is a distributed, scalable, NoSQL database designed to store and manage massive amounts of structured data. It is part of the Hadoop ecosystem and is built to run on top of the Hadoop Distributed File System (HDFS). HBase is ideal for scenarios requiring random, real-time read and write access to Big Data. It excels at hosting extremely large tables, potentially containing billions of rows and millions of columns, distributed across clusters of commodity hardware. HBase is an open-source project modeled after Google's Bigtable, a distributed storage system for structured data. Similar to how Bigtable utilizes the Google File System, HBase provides Bigtable-like capabilities leveraging HDFS for distributed data storage.

Features of HBase

HBase offers a range of features that make it suitable for Big Data applications:

- **Scalability**: It is linearly and modularly scalable, meaning it can scale by adding more nodes to the cluster.
- **Consistent Reads and Writes**: HBase ensures consistency for read and write operations.
- **Atomic Operations**: It provides atomic read and write operations, ensuring that during a read or write process, other processes are prevented from performing conflicting operations.
- **Java API**: It offers an easy-to-use Java API for client access.
- **Thrift and REST API**: HBase supports Thrift and REST APIs for non-Java front-ends, enabling interaction with data using XML, Protobuf, and binary data encoding options.
- **Real-time Query Optimization**: It supports Block Cache and Bloom Filters for optimizing real-time queries and handling high-volume query loads.
- **Automatic Failover**: HBase provides automatic failure support between Region Servers.
- **Metrics Export**: It supports exporting metrics via the Hadoop metrics subsystem to files.
- **Flexible Data Model**: HBase does not enforce relationships within data, offering a flexible schema.
- **Random Access**: It is a platform designed for storing and retrieving data with random access patterns.

HBase vs. RDBMS

HBase and traditional Relational Database Management Systems (RDBMS) differ significantly in their design and use cases:

Feature	HBase	RDBMS
Data Model	Column-oriented	Row-oriented
Schema	Schema-less (flexible)	Fixed Schema
Scalability	Horizontally scalable (add more servers)	Vertically scalable (increase server resources)
Transactions	Limited Support	Full ACID compliance
Consistency	Eventual consistency	Strong consistency

Feature	HBase	RDBMS
Use Case	Big data, real-time random access	Structured data, complex queries

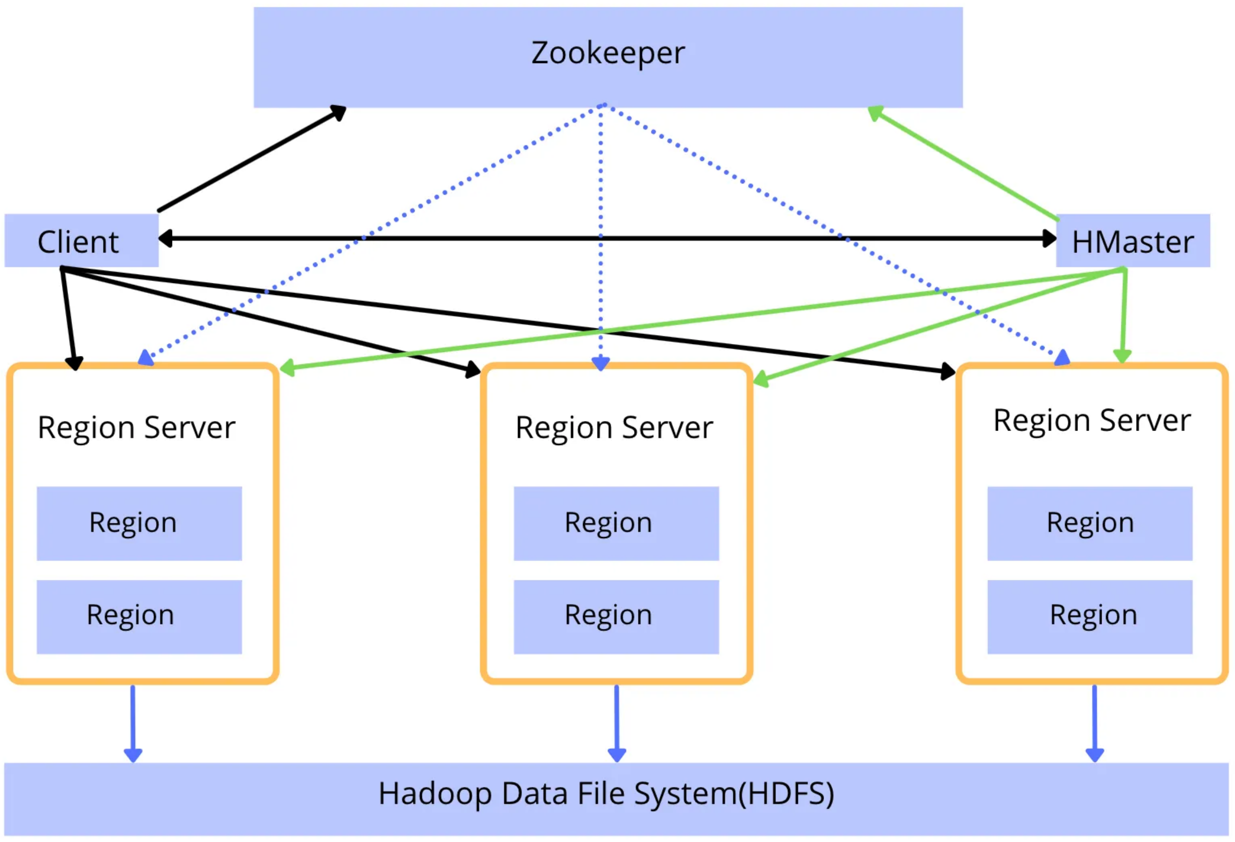
HBase vs. HDFS

While HBase relies on HDFS for storage, they serve different purposes:

Feature	HBase	HDFS
Data Access	Real-time read/write	Batch Processing
Schema	Column-oriented	File-based
Latency	Low Latency	High Latency
Use Case	Real-time queries, random access	Large-scale batch processing, data archiving

HBase Architecture

The HBase architecture consists of several key components that work together to provide a distributed and scalable data store.



Components:

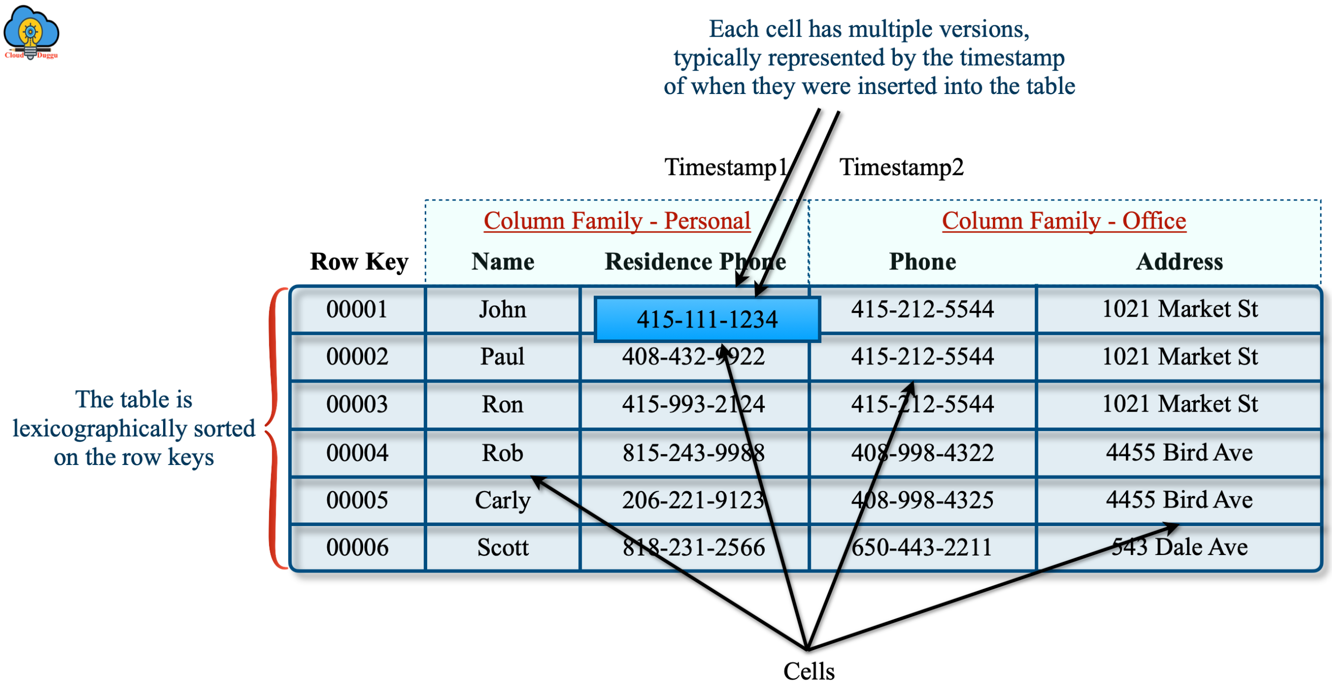
1. **HMaster**:
   - The master server responsible for critical administrative functions.
   - Assigns regions to Region Servers.

- Handles Data Definition Language (DDL) operations such as creating and deleting tables.
- Manages load balancing across Region Servers.

2. **RegionServer**:
   - Responsible for serving data to clients for read and write operations.
   - Each RegionServer manages one or more regions.
   - Typically runs on an HDFS DataNode in a distributed cluster.
3. **Regions**:
   - A Region represents a subset of a table's data, defined by a contiguous range of row keys.
   - Regions are the basic units of distribution and are comprised of Column Families.
   - Each region is served by a specific RegionServer.
   - Regions are automatically split when they grow too large (default size is 256 MB).
   - Region Servers handle, manage, and execute read/write operations for the regions they are responsible for.
4. **ZooKeeper**:
   - A distributed coordination service that plays a vital role in managing the HBase cluster.
   - Maintains configuration information and naming services.
   - Provides distributed synchronization and tracks the status of servers (which HMaster is active, which Region Servers are available).
   - Notifies about server failures.
5. **HDFS (Hadoop Distributed File System)**:
   - The underlying distributed storage system where HBase actually stores its data files (HFiles).

HBase Data Model

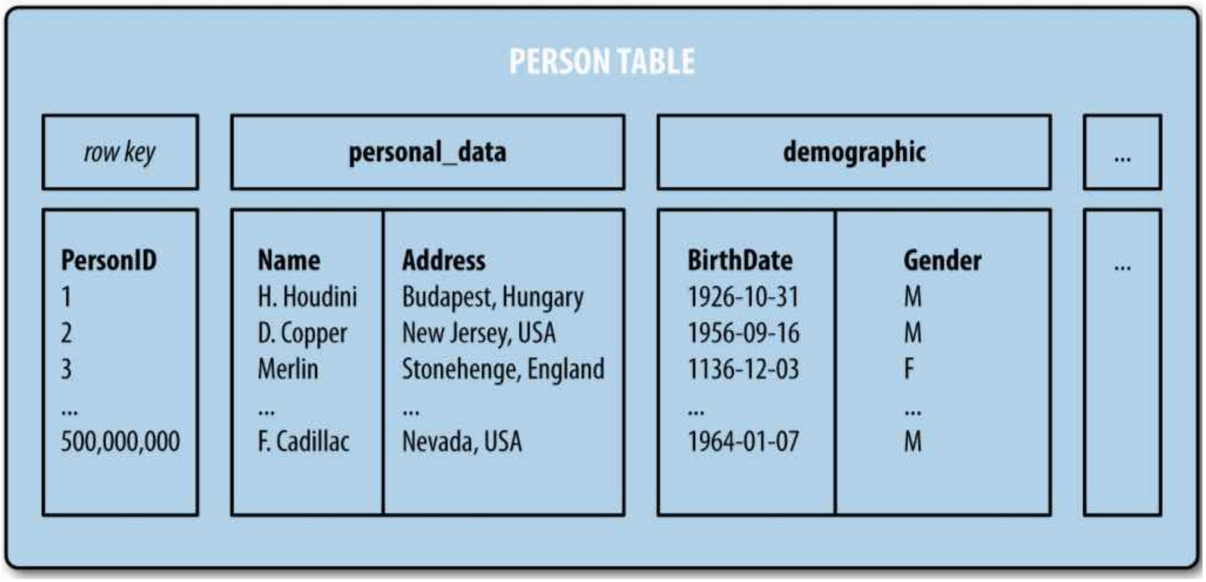
HBase stores data in tables, which consist of rows and columns. While this terminology is similar to RDBMS, the underlying model is quite different and is better conceptualized as a multi-dimensional map.


Each cell has multiple versions, typically represented by the timestamp of when they were inserted into the table
Timestamp1
Timestamp2
The table is lexicographically sorted on the row keys
Cells
415-111-1234

	Column Family - Personal		Column Family - Office	
Row Key	**Name**	**Residence Phone**	**Phone**	**Address**
00001	John	415-111-1234	415-212-5544	1021 Market St
00002	Paul	408-432-9922	415-212-5544	1021 Market St
00003	Ron	415-993-2124	415-212-5544	1021 Market St
00004	Rob	815-243-9988	408-998-4322	4455 Bird Ave
00005	Carly	206-221-9123	408-998-4325	4455 Bird Ave
00006	Scott	818-231-2566	650-443-2211	543 Dale Ave

- **Table**: An HBase table is a collection of rows.
- **Row**:
  - A row in HBase is uniquely identified by a **Row Key**.

- Each row contains one or more columns with their associated values.
- Rows are stored lexicographically (alphabetically) sorted by their row key.
- **Row Key Design**: The design of the row key is crucial for performance. The goal is to store related rows close to each other to optimize scans and lookups. A common pattern for row keys like website domains is to store them in reverse (e.g., `org.apache.www` instead of `www.apache.org`) so that all domains from the same top-level domain (e.g., `org.apache`) are grouped together.



- **Column**:
  - A column in HBase is defined by a combination of a **Column Family** and a **Column Qualifier**. These are typically delimited by a colon (`:`), for example, `family:qualifier`.
- **Column Family**:
  - Column families physically group a set of columns and their values together. This co-location is often done for performance reasons.
  - Each column family has its own set of storage properties, such as whether its values should be cached in memory, compression settings, row key encoding, etc.
  - All rows in a table share the same column families, which are defined at the time of table creation. However, a specific row might not have data (values) stored in every column family.
  - Column families cannot be changed dynamically after table creation. All columns within a column family are stored together on disk.
- **Column Qualifier**:
  - A column qualifier is added to a column family to provide a specific index or identifier for a piece of data within that family.
  - For example, within a column family named `content`, you might have column qualifiers like `html` (for `content:html`) and `pdf` (for `content:pdf`).
  - Unlike column families (which are fixed at table creation), column qualifiers are mutable and can vary significantly between rows.
- **Cell**:
  - A cell is the fundamental unit of data storage in HBase. It is defined by the unique combination of a row key, column family, and column qualifier.
  - Each cell contains a value and a timestamp.
- **Timestamp**:
  - A timestamp is associated with each value stored in a cell and serves as the identifier for a specific version of that value.

    - By default, the timestamp reflects the time on the RegionServer when the data was written. However, users can specify a custom timestamp when inserting data into a cell. This allows HBase to store multiple versions of a value for the same row and column.

HBase Regions

- **Region**: As mentioned earlier, an HBase table is horizontally divided by row key range into Regions. Each region is a subset of the table's data and is managed by a RegionServer. Regions are the fundamental building blocks of an HBase cluster, responsible for the distribution of tables and are comprised of column families. When a region grows too large, it is automatically split into two smaller regions. The default size of a region is 256MB.

- **RegionServer**: A server that manages one or more regions. It handles all read and write requests for the regions it is responsible for. In a distributed HBase cluster, a RegionServer typically runs on an HDFS DataNode. The RegionServer is responsible for handling, managing, and executing HBase operations (reads and writes) on its set of regions.

Basic HBase Operations

Creating a Table

Tables in HBase can be created using the HBase shell or the HBase Java API. Example using the HBase shell:

```
create 'my_table', 'cf1', 'cf2'
```

This command creates a table named `my_table` with two column families: `cf1` and `cf2`.

Inserting Data

Data is inserted into HBase using the `put` command in the HBase shell. Example:

```
put 'my_table', 'row1', 'cf1:col1', 'value1'
put 'my_table', 'row1', 'cf2:col2', 'value2'
```

These commands insert two cells into the `my_table` table for the row key `row1`.

- The first cell is in column family `cf1` with column qualifier `col1` and value `value1`.
- The second cell is in column family `cf2` with column qualifier `col2` and value `value2`.

Retrieving Data

Data can be retrieved from HBase using the `get` command in the HBase shell. To retrieve all cells associated with a specific row key:

```
get 'my_table', 'row1'
```

This command retrieves all data for `row1` in `my_table`.

To retrieve a specific column (column family and qualifier):

```
get 'my_table', 'row1', 'cf1:col1'
```

This command retrieves the value of the column `cf1:col1` for the row key `row1` in `my_table`.

(Note: The document mentions "Java Code" on page 15 but does not provide the actual code snippets.)

ZooKeeper's Role in HBase

HBase operates in a distributed environment, and the HMaster alone cannot manage all aspects of the cluster. This is where ZooKeeper comes in.

- **Distributed Coordination**: ZooKeeper is a distributed coordination service used to maintain server state and manage various aspects of the HBase cluster.
- **Server Status Tracking**: It maintains and tracks which servers (HMaster, Region Servers) are alive and available.
- **Failure Notification**: ZooKeeper provides server failure notification, which is crucial for HBase's fault tolerance mechanisms.

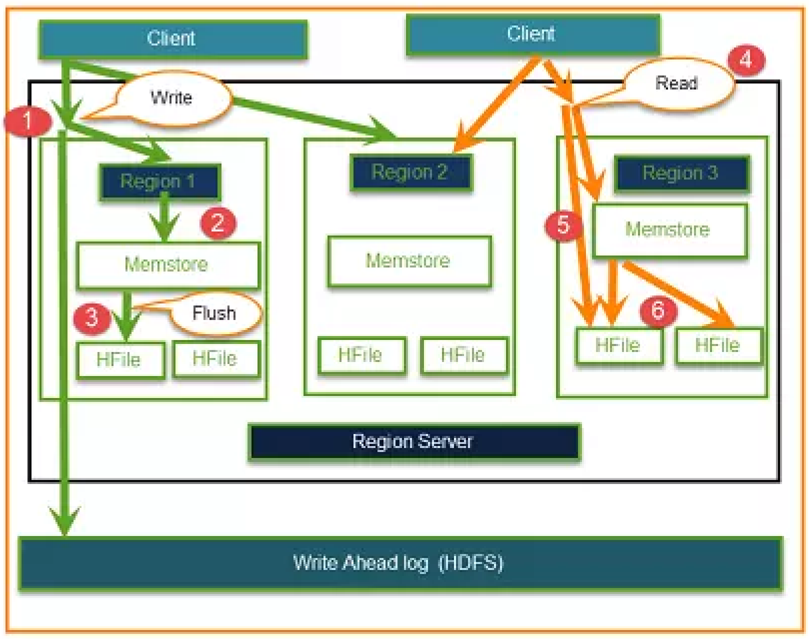
How ZooKeeper is used:

1. The **active HMaster** sends heartbeat signals to ZooKeeper to indicate that it is active and operational.
2. **Region Servers** send their status to ZooKeeper, indicating that they are ready to handle read and write operations for their assigned regions.
3. An **inactive HMaster** (if configured for high availability) acts as a backup. If the active HMaster fails, the inactive HMaster can take over, a process often coordinated via ZooKeeper.

HBase Read Operation

When a client needs to read data from HBase, the following steps typically occur:

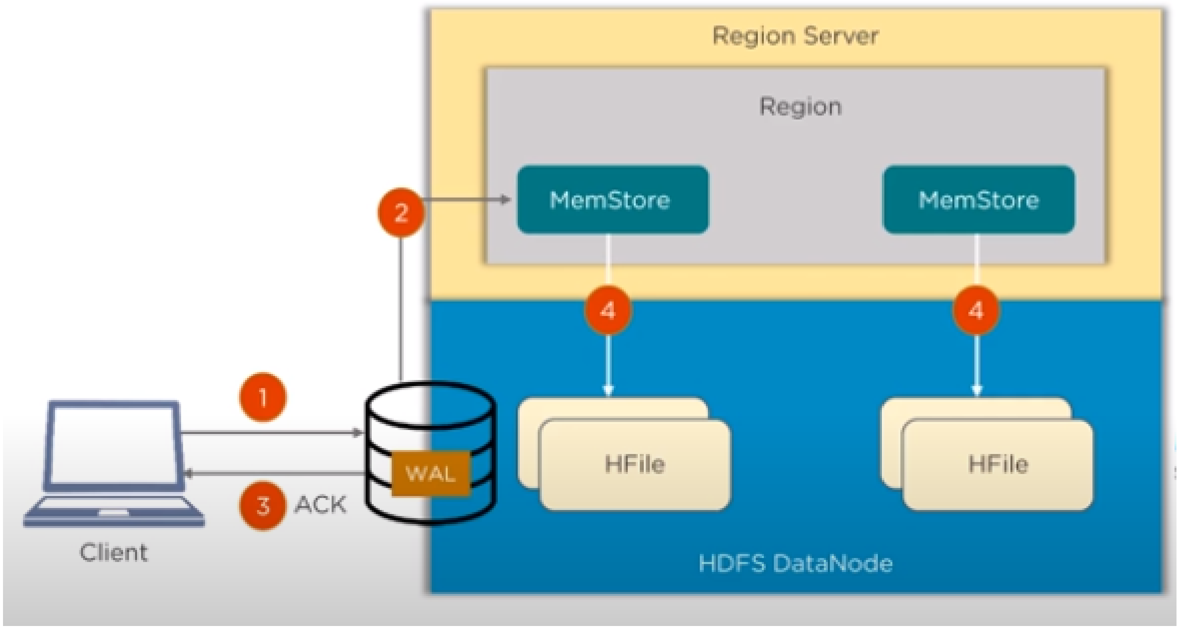
1. **Locate META Table**: HBase maintains a special catalog table called the **META table**. This table holds the locations (i.e., which RegionServer is serving which region) of all regions in the cluster.
2. **ZooKeeper Interaction**: The client first contacts ZooKeeper to find the RegionServer that hosts the META table.
3. **Query META Table**: The client then queries the META table (hosted on the identified RegionServer) to determine which RegionServer is responsible for the specific row key it wants to access.
4. **Cache Location**: The client caches this information, including the location of the META table and the location of the target region, to avoid repeated lookups for subsequent operations on nearby data.
5. **Direct Data Read**: Finally, the client contacts the appropriate RegionServer directly to read the data.



HBase Write Operation

When a client issues a write request (e.g., a put operation) to HBase, the data goes through the following stages:

1. **Write to Write-Ahead Log (WAL)**:
   - The first step is to write the new data to a **Write-Ahead Log (WAL)**.
   - The WAL is a file stored on the distributed file system (HDFS). Its purpose is to ensure data durability and to be used for recovery in case of a RegionServer failure. New data that has not yet been persisted to permanent storage (HFiles) is first logged here.
2. **Write to MemStore**:
   - After the data is written to the WAL, it is then copied to the **MemStore**.
   - The MemStore is an in-memory write cache.
   - Each column family within a region has its own MemStore. Storing new data in memory allows for fast write operations.
3. **Acknowledgment to Client**:
   - Once the data is successfully placed in the MemStore (after being written to WAL), the client receives an acknowledgment that the write operation was successful.
4. **Flush MemStore to HFile**:
   - As the MemStore fills up with data and reaches a certain threshold size, its contents are flushed (or "committed") to disk.
   - This flush operation creates a new **HFile** on HDFS. HFiles store the actual data (rows as sorted KeyValues) persistently on disk. This process is also known as a "memstore flush" or "compaction" (minor compaction often involves writing out MemStore data).



MongoDB

MongoDB is a NoSQL database that stores data in a flexible, document-oriented format. It differs from relational databases in how it structures and manages data. MongoDB is designed around a flexible document model, offering a dynamic schema and horizontal scalability.

Collections

In MongoDB, collections are containers for documents. They are similar in concept to tables in relational databases, but they do not enforce a strict schema on the documents within them.

- **Creating a Collection:** You can explicitly create a collection using the `createCollection()` method on the database object. If you insert a document into a non-existent collection, MongoDB will create the collection implicitly.

  ```
  db.createCollection('mongo-collection')
  ```

- **Viewing Collections:** To see the collections in your current database, you can use the `show collections` command.

  ```
  show collections
  ```

Documents

Documents are the fundamental units of data in MongoDB, analogous to rows in a relational database. MongoDB stores data in BSON documents, which are a binary representation of JSON. Documents are flexible and can have different structures within the same collection. They consist of field-value pairs.

- **Inserting Documents:** You can add documents to a collection using the `insertOne()` for a single document or `insertMany()` for multiple documents.

```
// Create a single document
db.mongoCollection.insertOne({
    name: "Sudarshan",
    age: 30,
    city: "Kathmandu"
});

// Create multiple documents
db.mongoCollection.insertMany([
    { name: "Nishan", age: 25, city: "Lalitpur" },
    { name: "Keshab", age: 35, city: "Kirtipur" }
]);
```

Object Ids

Every document in MongoDB requires a unique `_id` field. If you don't provide one during insertion, MongoDB automatically generates a unique **`ObjectId`** for the document. `ObjectId` is a special 12-byte BSON type designed to be unique identifiers. You can see the generated `_id` with `ObjectId` when you query documents.

Queries on MongoDB

Querying in MongoDB involves selecting documents from collections based on specified criteria.

- **Finding All Documents:** The `find()` method with an empty object `{}` as the query selects all documents in a collection.

  ```
  db.mongoCollection.find()
  ```

- **Finding Specific Documents:** You can provide a query document to `find()` to filter results based on field values.

  ```
  // Find specific documents with a condition
  db.mycollection.find({ age: 30 })
  ```

- **Finding a Single Document:** The `findOne()` method returns the first document that matches the query criteria.

  ```
  // Find a single document
  db.mycollection.findOne({ name: "Nishan" })
  ```

Aggregation Pipeline

The **Aggregation Pipeline** is a powerful framework in MongoDB for performing advanced data processing and analysis on documents within a collection. It allows you to transform documents, perform calculations, group data, and reshape the output. The pipeline consists of a series of stages, processed in order, where the output of one stage becomes the input for the next.

The basic structure of an aggregation pipeline is:

```
db.collection.aggregate([
    { stage1 },
    { stage2 },
    { stage3 },
    ...
])
```

Here are some common aggregation stages described in the document:

- `$match`: Filters documents to pass only those that match the specified condition(s). This is similar to the `WHERE` clause in SQL.

  ```
  { $match: { status: "A" } }
  ```

- `$group`: Groups documents by a specified field or fields and performs aggregation operations on the grouped data. Common aggregation operators include `$sum`, `$avg`, `$min`, and `$max`.

  ```
  {
   $group: {
   _id: "$category",
   totalSales: { $sum: "$sales" }
   }
  }
  ```

  In this example, documents are grouped by the `category` field, and the `$sum` operator calculates the total of the `sales` field for each category. The `_id` field in the output represents the grouping key.

- `$sort`: Sorts the documents based on the values of specified fields. You specify the sort order using `1` for ascending and `-1` for descending.

  ```
  { $sort: { totalSales: -1 } }
  ```

  This stage sorts the documents by the `totalSales` field in descending order.

- `$project`: Reshapes each document in the pipeline, allowing you to include, exclude, or rename fields. You can specify which fields to include by setting their value to `1` and exclude by setting to `0`. The `_id` field is included by default unless explicitly excluded.

```
{
 $project: {
 product: 1,
 totalSales: 1,
 _id: 0
 }
}
```

This projects the documents to include only the `product` and `totalSales` fields, while excluding the `_id` field.

- **`$limit`**: Passes only the first `n` documents to the next stage, where `n` is the specified limit.

```
{ $limit: 5 }
```

- **`$skip`**: Skips the specified number of documents and passes the remaining documents to the next stage.

```
{ $skip: 10 }
```

- **`$unwind`**: Deconstructs an array field from the input documents. For each element in the array, it outputs a separate document with the array field replaced by the element.

```
{ $unwind: "$categories" }
```

If a document has a `categories` array like `["Electronics", "Furniture"]`, `$unwind` would produce two documents, one for "Electronics" and one for "Furniture".

- **`$lookup`**: Performs a left outer join from one collection to another, allowing you to combine documents from different collections based on a shared field.

```
{
 $lookup: {
 from: "orders",
 localField: "order_id",
 foreignField: "_id",
 as: "order_details"
 }
}
```

This stage joins documents from the current collection with documents from the "orders" collection where the `order_id` in the current collection matches the `_id` in the "orders" collection. The matching documents from "orders" are added as an array in the `order_details` field.

- **`$addFields`**: Adds new fields to documents or modifies existing fields. This is useful for calculating new values based on existing fields.

  ```
  { $addFields: { totalPrice: { $multiply: ["$price", "$quantity"] } } }
  ```

  This adds a new field `totalPrice` to each document by multiplying the values of the `price` and `quantity` fields.

Aggregation Pipeline Example:

Consider a collection named `sales` with documents like this:

```
[
 { "_id": 1, "product": "A", "sales": 100, "category": "Electronics" },
 { "_id": 2, "product": "B", "sales": 200, "category": "Electronics" },
 { "_id": 3, "product": "C", "sales": 150, "category": "Furniture" }
]
```

To calculate the total sales per category and sort the results, you can use the following aggregation pipeline:

```
db.sales.aggregate([
 { $group: { _id: "$category", totalSales: { $sum: "$sales" } } },
 { $sort: { totalSales: -1 } }
])
```

This pipeline first groups the sales documents by the `category` field (`$group`) and calculates the sum of the `sales` for each category, storing the result in `totalSales`. Then, it sorts the resulting documents in descending order based on `totalSales` (`$sort`).

Nested Documents

MongoDB allows you to embed documents within other documents. These are referred to as **Nested Documents** or Embedded Documents. This is a way to represent one-to-one or one-to-many relationships in a denormalized way, without the need for joins as in relational databases. For example, instead of having separate `customers` and `addresses` collections and linking them with a foreign key, you could embed the address document(s) directly within the customer document. This can improve read performance as the related data is retrieved in a single query.

Here's an example of inserting a document with a nested address document into a customers collection:

```
db.customers.insertOne({
  name: "Alice Smith",
  email: "alice.smith@example.com",
```

```
  address: {
    street: "123 Main St",
    city: "Anytown",
    country: "USA"
  },
  hobbies: ["reading", "hiking"]
});
```

In this example, the address field contains a nested document with its own fields (street, city, country).

You can query based on fields within nested documents using dot notation:

```
// Find customers living in Anytown
db.customers.find({ "address.city": "Anytown" });

// Find customers who have 'reading' as a hobby
db.customers.find({ hobbies: "reading" });
```

The first query uses `address.city` to access the city field within the nested address document. The second query demonstrates querying within an array field (hobbies), which is also a common pattern in MongoDB documents.