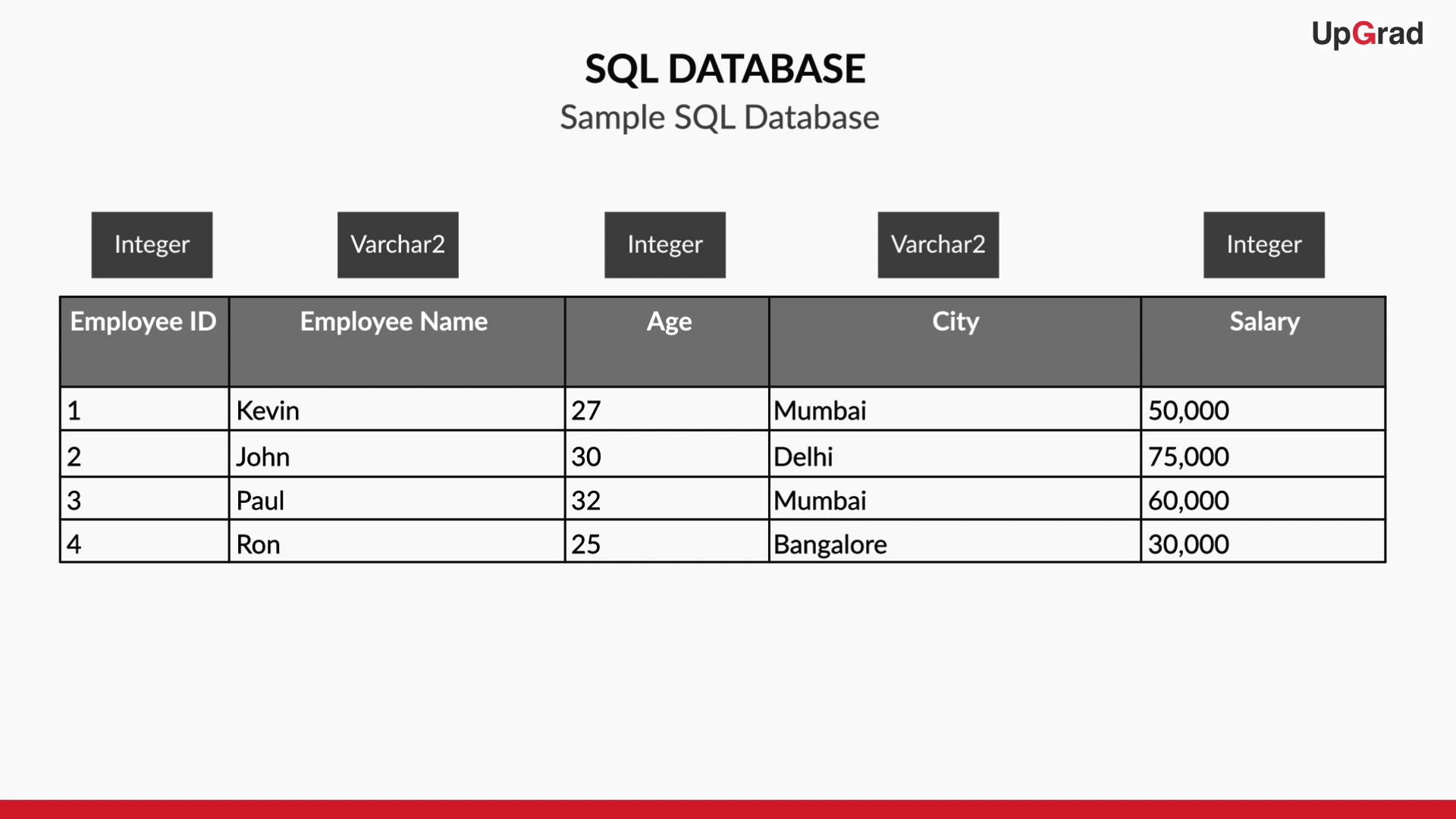


In this module, you were introduced to HBase - NoSQL database component of Hadoop Ecosystem and the features provided by it that differentiate it from Hadoop and other relational or NoSQL data stores. You started with understanding the limitations of SQL and HDFS and then got introduced to NoSQL databases. After looking at the reasons for their increasing popularity, you saw the CAP theorem and its consequences, in particular, the tradeoffs between consistency, availability and partition tolerance in the design of networked shared-data systems. Then, you also looked at how data is stored in HBase and understood its data model.



Databases like RDBMS which supports SQL, stores data in tables divided into rows and columns. Here each row represents a data record which comprises of a collection of various columns or attributes. A fixed schema governs the tables, which means a data type is associated with each column. In case of data type violations, while inserting data into the table, the insert operation fails, and there is no flexibility in the data model to accommodate such discrepancies in data. A record is added successfully to a SQL table only when the type of all the individual data elements in the record is strictly adhering to the destination table’s schema. Because of such strict schema oriented regulations, SQL databases are used for storing and processing structured data.



**Figure 1: SQL Database**

With technological advancements such as ease in availability of internet at affordable prices, increased usage of IoT devices, usage of social networking websites, etc. the rate of generation of digital data grew at an alarming pace. Hence, traditional systems were not scalable to accommodate such massive volumes of data. Moreover, the digital data was not structured anymore, and organisations were not interested in discarding this data because nowadays non-structured data comprises almost 80% of the total generated data and 80% is a significant proportion of the total data. So to store and process humongous non-structured data, organisations started to migrate from traditional SQL supporting systems to other data processing tools which are scalable and fault tolerant.

When organisations were trying to find out a way for storing and processing Big Data, Hadoop came to their rescue. Hadoop uses HDFS as its storage layer and is used to store data in a distributed manner across a cluster of commodity machines. HDFS is a robust file system which can store structured, unstructured and semi-structured data and is horizontally scalable.

Though Hadoop overcame some of the challenges faced by SQL, it introduced new challenges. MapReduce of Hadoop (used for processing the data stored in HDFS) is well suited for batch processing, where the whole data is accessed sequentially. But the drawback of using MapReduce for processing big data is that it is not suited for all use cases like performing random lookups on data. Hence, apart from Hadoop, there was a need for some solution for these use cases. These Limitations of Hadoop led to the inception of NoSQL datastores.

Some of the reasons for increasing popularity of NoSQL databases are:

* NoSQL datastores are efficient in storing and handling Big Data. Based on the targeted use cases, every NoSQL database has its data model for storing data.
* NoSQL data stores provide scalability, i.e. in case of space crunch extra space can be easily created by just adding additional nodes to the cluster.
* NoSQL data stores are flexible and do not restrict themselves to a fixed schema. Hence, NoSQL data stores can adapt to changes in the schema of data dynamically.

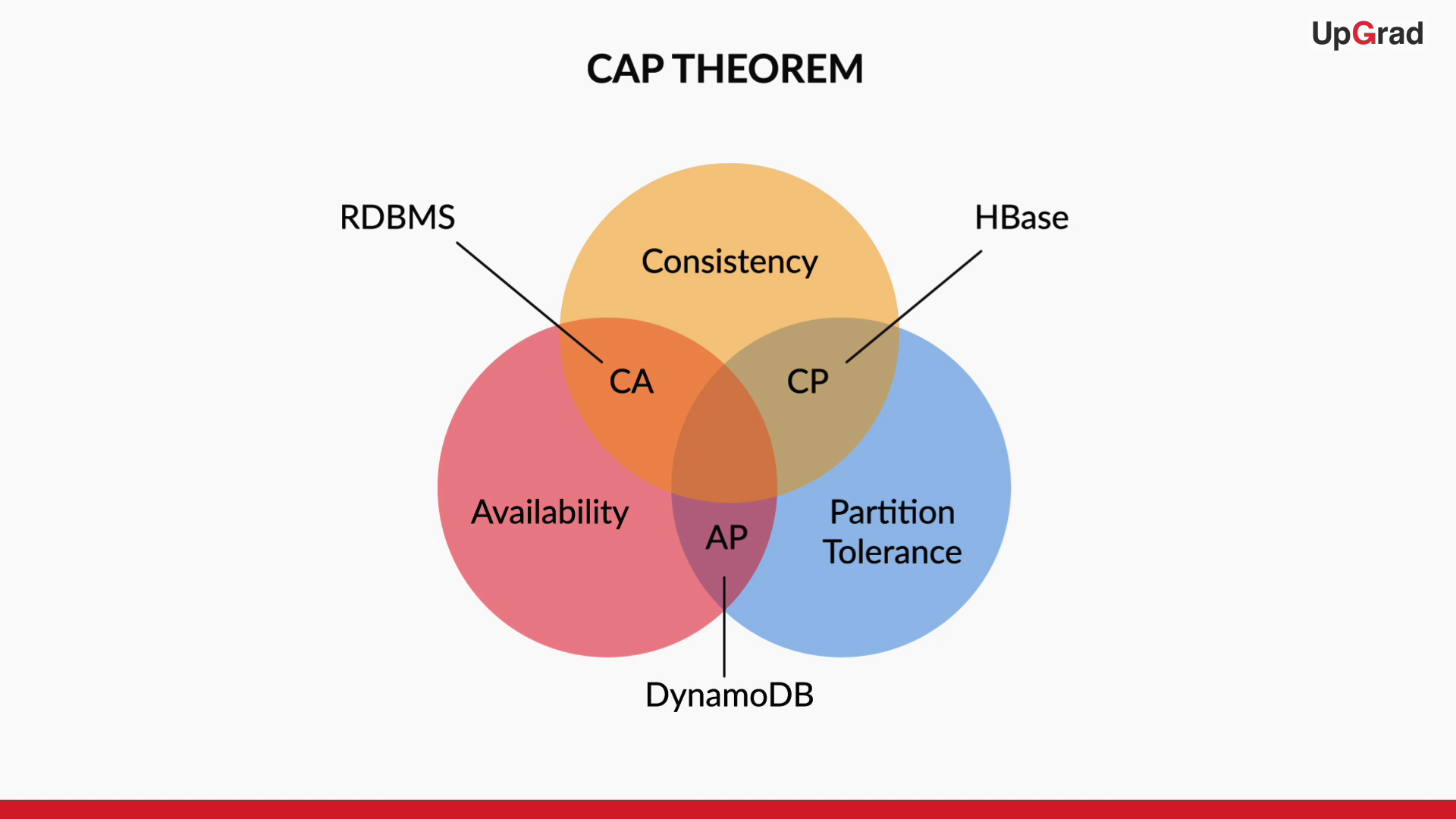


The CAP theorem relates to the distributed systems that store data. The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the three guarantees - Consistency, Availability and Partition tolerance. Consistency guarantees that every node in the distributed system returns the same, most successful, recent write. Availability is when every request receives a response without the guarantee that it contains the most recent write. Partition tolerance is when the system continues to function and upholds its guarantees in spite of network partitions.

Some of the known systems and the guarantees provided by them are mentioned below:

* **RDBMS Systems (support SQL):** Consistency and Availability.
* **NoSQL Systems:** They store data in a distributed manner across a cluster of interconnected machines and provide network partitioning. Hence, there are two flavours of NoSQL databases which provide a different set of guarantees:

1. Consistency and Partition tolerance(Ex - HBase, MongoDB).
2. Availability and Partition tolerance(Ex - Cassandra, DynamoDB).



**Figure 2: CAP Theorem**

Depending upon the type of application, while designing networked shared-data systems, the system designers should make the trade-off between consistency, availability and partition tolerance accordingly.

If one chooses consistency over availability, the system will report an error or a time-out if particular information is not latest due to network partitioning. Consider an example of using a chat application and you sent an instant message to your friend. If during this process there is a network partition then ideally the message will not get delivered and there will be a timeout. Once the system is up, the exact message which you sent will be delivered to your friend's message box. So, in a messaging application consistency was given preference over availability. Just to ensure instantaneous delivery of message the system should not deliver a garbage message.

When one chooses availability over consistency, the system processes the query and returns the newest version of information available, even if it the data is not latest due to network partitioning. Consider another example of using a travel portal like MakeMyTrip to check for available hotel rooms. So, due to network partition, if the portal is unable to fetch latest prices it’s perfectly fine because the customer may not be concerned about prices, he/she is concerned about other aspects like location of the hotel, amenities, etc. So, in such scenarios using a highly available data store is must because travel portals and e-commerce websites cannot afford website timeout for showing consistent results.



HBase is a distributed data store built on top of HDFS and can leverage all the benefits provided by Hadoop or HDFS. HBase has the ability which allows a user to query for individual records as well as derive aggregate analytic reports across a massive amount of data. HBase was first released as a code for an open source BigTable implementation.

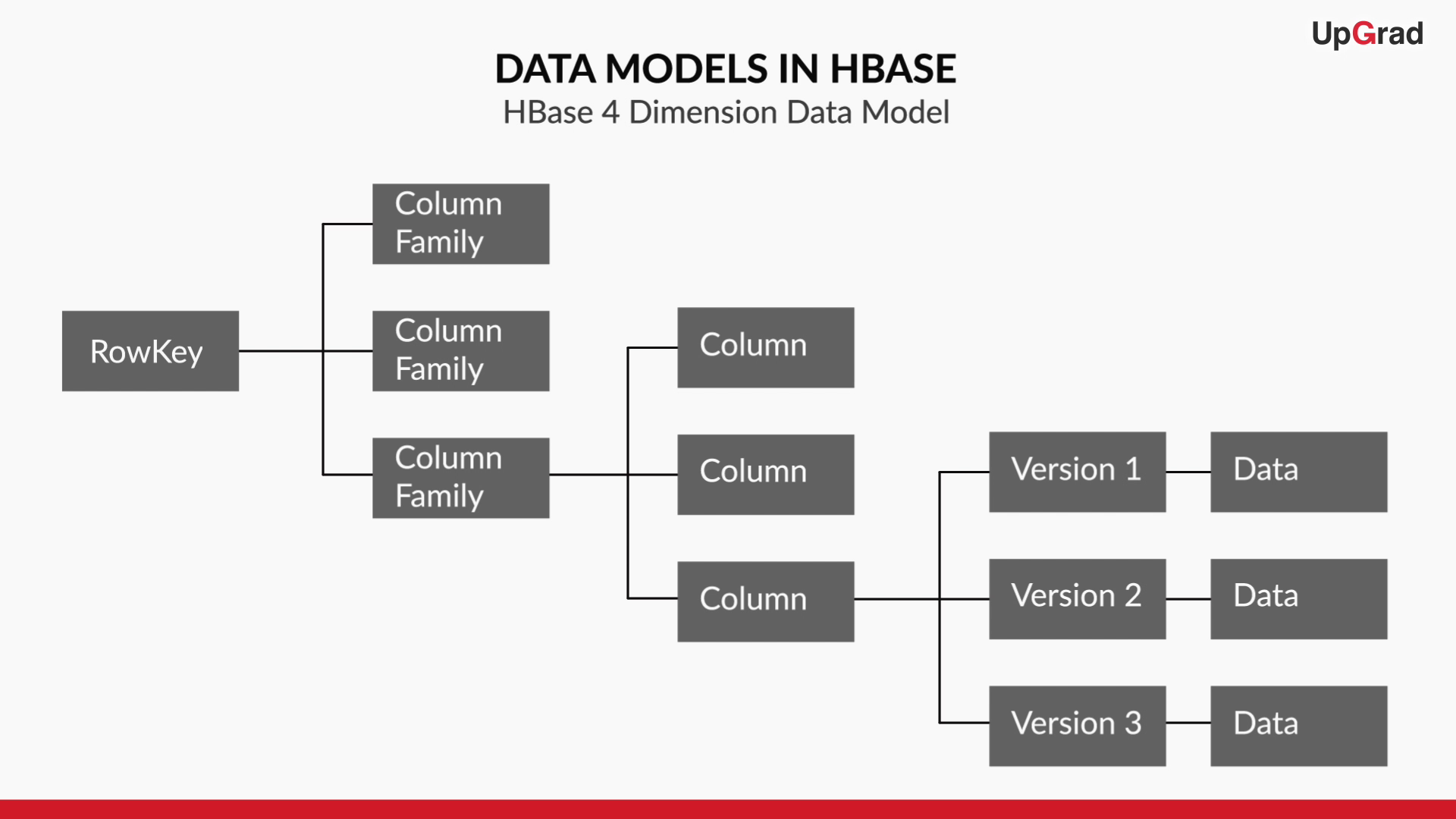
Some of the prominent features of HBase that distinguishes it from Hadoop and other relational or NoSQL data stores are:

* HBase stores data internally as key-value pairs where the rowkey is the key and the rest of the data is value.
* HBase records are sorted by RowKey.
* HBase Columns do not have any specific data type and all the data in HBase is stored in the form of bytes.
* HBase do not follow a strict schema (Schema-less and dynamic architecture) which means any number of columns can be added dynamically.



Like SQL, HBase also stores data in tabular format with some modifications and the data model of HBase can be summarised as follows:

* In HBase, data is stored in tables which are nothing but a collection of rows. In an HBase table, a row is a collection of column families. A column family is a collection of related columns known as column qualifiers, and there can be any number of columns in a single column family.
* Every entry in an HBase table is identified and indexed by a RowKey and for every RowKey, an unlimited number of columns can be stored. This feature ensures that the schema of an HBase table is flexible and the table can scale linearly.
* Each column can have a configurable number of versions and there is a provision for selecting data from a particular version. In HBase, each version is identified by a timestamp and each column has one or more versions.



**Figure 3: Data Models in HBase**

To access an individual piece of data, one needs to know its row key, column family, column qualifier, and version which makes it a four dimensional model.



HDFS along with Map Reduce follows write once and read many times paradigm. In other words, data in HDFS is written once, but it can be read an unlimited number of times. Even there is no provision for updating an existing data set or record in HDFS. Even though HBase uses HDFS to store data but HBase supports update operations by maintaining multiple versions of same data points.

Shell commands used for performing CRUD (Create, Read, Update and Deletion) in HBase are:

* **List:** This command is used to check all the tables present in your HBase instance.

Syntax: **hbase> list**

* **Create:** For creating a table in HBase, this command is used. To create a table, you must name the table and define its schema. As a part of the schema, you are required to specify the column families. However, columns are defined later while inserting records into the HBase table.

Syntax: **hbase> Create ‘<table\_name>’, ‘<column\_family\_name>’**

Example: Create a table named Students with three column families named Personal Details, Contact Details and Marks.

Command: Create ‘Students’, ‘Personal Details’, ‘Contact Details’, ‘Marks’

* **Scan:** This command is used to view the contents of a table created. The optional parameters in syntax include TIMERANGE, FILTER, TIMESTAMP, LIMIT, MAXLENGTH, COLUMNS, CACHE, STARTROW and STOPROW.

Syntax: **hbase> Scan ‘<table\_name>’ {Optional parameters}**

* **Put:** This command is used to add or update a cell value in the mentioned table. Each cell in a table is identified by a row id and a column name.

Syntax: **hbase> put ‘<table\_name>’, ‘<row\_key>’, ‘<column\_value>’, ‘<value>’**

Example: Insert a “Name” record (student name “Sandeep”) into the “Students” table in a row ‘students1’ and in the column ‘Personal Details’ that was created earlier. After insertion, update the earlier record by adding the “email id” of Student1.

Command: Put ‘Students’, ‘Student1’, ‘Personal Details:Name’, ‘Sandeep’

Update Command:

Put ‘Students’, ‘Student1’, ‘Personal Details:Email’, ‘Sandeep@pqr.com’

* **get:** This command is used to fetch data from HBase. There are various ways to make use of the get command.

Syntax: **hbase> get ‘<table\_name>’, ‘<row\_key>’, {‘<Additional Parameters>’}**

Example: Fetch data from “Students” table created in three ways as mentioned below:

1. Fetch entire “Student1” row data
2. Fetch data from the column family “Personal Details”
3. Fetch data from the particular column “Name”

Commands:

1. get ‘Students’, ‘Student1’
2. get ‘Students’, ‘Student1’, {COLUMN => ‘Personal Details’}
3. get ‘Students’, ‘Student1’, {COLUMN => ‘Personal Details:Name’}

* **delete:** To delete a cell in HBase table this command is used.

Syntax: **hbase> delete ‘<table\_name>’, ‘<row\_key>’, ‘<column\_value>’, ‘<value>’**

Example: Delete the “email id” updated to the “Student1” record.

Command: delete ‘Students’, ‘Student1’, ‘Personal Details:Email’

**Note:**

1. If we observe the “Put command”, we can understand that columns are defined while inserting records into the HBase table. (Column “Name” is defined while inserting the name “Sandeep”)
2. “<Additional Parameters>” mentioned in the syntax include TIMERANGE, TIMESTAMP, VERSIONS and FILTERS.
3. “scan” command helps to verify whether the contents of a table are deleted or not.



Shell commands that are used to modify and view the specifications of the HBase table are:

* **describe:** This command is used to check the schema of the HBase tables.

Syntax: **hbase> describe <table\_name>**

* **exists:**  This command is used to verify whether a given table is present in the HBase storage or not.

Syntax:  **hbase> exists <table\_name>**

* **alter:** This command is used to modify the specifications of a HBase table like adding column families, updating the version number of column families. It can also be used to delete the column family by applying delete method to it.

Syntax (Modify the number of versions):

**hbase> alter ‘<table\_name>’, Name=>’<column\_family\_name>’, VERSIONS => <new\_version\_number>**

Syntax (Modify the number of versions for multiple column families):

**hbase> alter ‘<table\_name>’, {Name=>’<column\_family\_name>’, VERSIONS => <new\_version\_number>}, {Name=>’<column\_family\_name>’, VERSIONS => <new\_version\_number>}**

Syntax (To delete):

**hbase> alter ‘<table\_name>’, Name=>’<column\_family\_name>’, METHOD => <’delete’> or hbase> alter ‘<table\_name>’, ‘delete’ => ‘<column\_family\_name>’**

* **drop:**  This command is used to delete a cell in HBase table. But this operator cannot be applied directly to the table. Instead, the table is first disabled. And then it is dropped.

Syntax:

Step 1: **hbase> disable ‘<table\_name>’**

Step 2: **hbase> drop ‘<table\_name>’**

* **truncate:**  This command is used to remove all the data from the table (Note that we do not intend to delete the table, just the data that is stored in the table). Internally, this command disables the table, drops it and then again recreates it but for us, the end result is that the table’s data has been removed.

Syntax: **hbase> truncate ‘<table\_name>’**

HBase stores multiple versions of the data present in a single cell and that data stored in HBase tables can be filtered using shell commands.

* **Get data based on Timestamp:** The“get” command can also be used to retrieve past versions of records based on timestamp.
* Syntax: **hbase> get ‘<table\_name>’, <row\_key>, {COLUMN => ‘<column\_family\_name>’, TIMESTAMP => value}**
* **Get data based on filter condition:** In HBase, fetching data based on a filtering condition is achieved by using Filters. In HBase, filters are like java methods which take two input parameters that are, a logical operator and a comparator. The logical operator specifies the type of the test i.e. equals, less than, etc. The comparator is the number/value against which you wish to compare your record.

Some commonly used filter functions are:

1. **ValueFilter:** A ValueFilter takes a comparison operator and a comparator as the parameter. It compares each value with the comparator using the comparison operator. If the check is true then result is displayed on the console.

Syntax: **ValueFilter (<compareOp>, ‘<value\_comparator>’)**

Example of using ValueFilter with Scan command:

Let us consider an example of an HBase table named “Companies” maintained by the placement cell of a college , which contains the details of all the companies that visit the college every year. Check whether the company “UpGrad” exists in that table or not.

Command: hbase> scan ‘Companies’, {FILTER => “ValueFilter(=, ‘binary:UpGrad’)”}

1. **QualifierFilter:** A QualifierFilter also takes two parameters they are comparison operator and comparator. Each qualifier name is compared with the comparator using the compare operator and if the comparison is true, it returns the key-values in that column.

Syntax: **QualifierFilter (<compareOp>, ‘<qualifier\_comparator>’)**

Example of using QualifierFilter with Scan command:

Let us consider an example of an HBase table named “Companies” maintained by the placement cell of a college , which contains the details of all the companies that visit the college every year. Do etch all the names of the companies present in the table (In the given HBase table,the names of the companies are present in the column named as “Name”).

**Command**: hbase> scan ‘Companies’, {FILTER => “QualifierFilter(=, ‘substring:Name’)”}

1. **FamilyFilter:** A FamilyFilter is used to fetch key-values for a specified column family.

Syntax: **FamilyFilter (<compareOp>, ‘<family\_comparator>’)**

Example of using FamilyFilter with Scan command:

Let us consider an example of an HBase table named “Companies” maintained by the placement cell of a college, which contains the details of all the companies that visit the college every year. Fetch the contact details (which include the mobile number, Email ID etc) of all the companies present in the table(In the given HBase table,the names of the companies are present in the column family named as “Contact Details”).

**Command**:hbase> scan ‘<table\_name>’, {FILTER => “FamilyFilter(=, substring:Contact Details’)”}

* **count:**  This command is used to count the number of rows present in the table.

**Syntax:** hbase> count ‘<table\_name>’