**A) What is NOSQL and Why No SQL**

* **Discussing what is SQL and data model used for SQL(RDBMS)**

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 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. As discussed in previous modules, structured data can be easily stored using the relational databases because data is mostly in tabular format with well-defined columns. Most importantly, the data contained in individual columns will be of the uniform data type. Databases supporting SQL are used for storing transactional data which are mostly structured in nature such as bank transactions, hotel reservations etc. **(00.35 Databases like RDBMS - 2:10 hotel reservations etc.)**

* **Why popularity of SQL started to decline or for which use cases SQL databases were incompetent**

In the introduction module, you understood why traditional systems are considered unpopular when it comes to storing and processing big data. The 4 V's of Big data, i.e. volume, variety, velocity and veracity gave you a clarity regarding how storing and processing big data using traditional systems like RDBMS is challenging. 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.

**(In the Introduction module 2:24 - 3:20 such massive volumes of data.) ( Moreover 3:48 - 4:25 and fault tolerant)**

* **Hadoop/HDFS solved some of the challenges faced in SQL. But introduced new challenges**

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. As already discussed HDFS is used to store data in a distributed manner across a cluster of machines. HDFS is a robust file system which can store structured, unstructured and semi-structured data. Apart from this HDFS uses commodity hardware to store data and is horizontally scalable. So, with the advent of HDFS, the problem of storing massive non-structured data was solved, but at this point, the job is half done. The stored data is of no use if the data is not being churned or processed. So, Hadoop used Map-Reduce for processing the data stored in HDFS. But the biggest drawback of using Map-Reduce for processing big data is it is not suited for all use cases. Map-reduce is well suited for batch processing, where the whole data set is accessed sequentially. In other words, irrespective of the task being performed entire data set is always scanned. So, there was a need for a way to access the data randomly.

**(When Organisations were trying 4:43 - 6:33 the data randomly)**

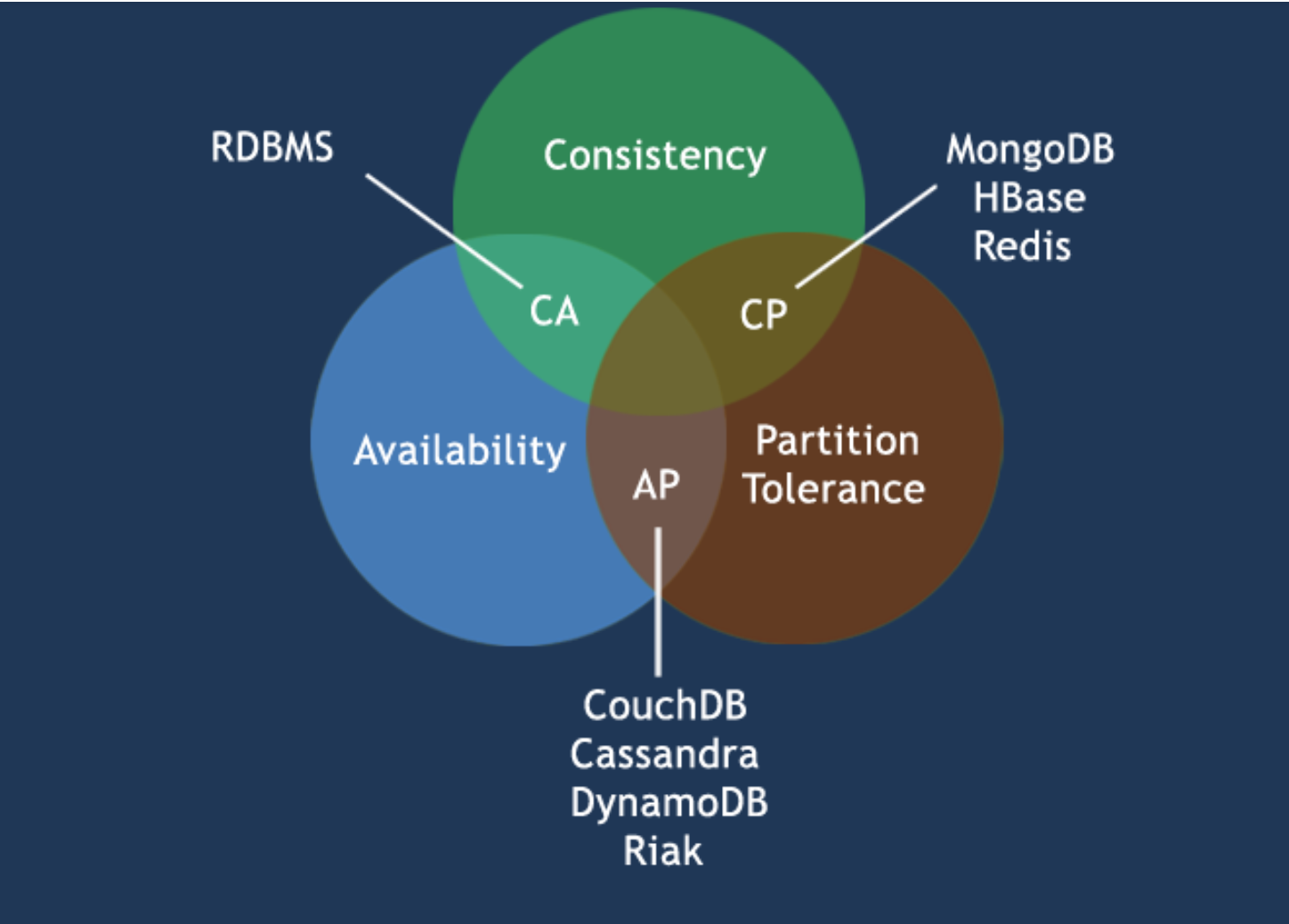
* **What is a NoSQL database and how it solved the SQL and HDFS-related challenges**

We discussed that Hadoop solved the problem of storing huge volumes of data which could be structured or unstructured. But capabilities of map-reduce were only limited to performing batch processing on the data. Like it is not a preferable solution for quick random lookup of specific records from a vast data store. Hence, apart from Hadoop, there was a need for some other solutions which can service use cases such as random lookup. So this led to the inception of NoSQL datastores. Unlike SQL data stores, every NoSQL data stores are designed to solve a particular use case. Which means every data store has its own functionality and which may not be there in other NoSQL datastores. Irrespective of the use cases these NoSQL data stores serve, all NoSQL stores are fundamentally intended to run on a cluster of machines. The name NoSQL is misleading, i.e. NoSQL can be expanded as "Not Only SQL" which means there are other non SQL ways of managing data in the data store, but sometimes SQL can be used for querying data as well. So some of the reasons why there is an increasing popularity of NoSQL databases are firstly 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. Secondly, 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. Thirdly, NoSQL data stores are flexible. They do not restrict themselves to a fixed schema. Hence, NoSQL data stores can adapt to changes in the schema of data dynamically.

**( We Discussed that Hadoop 6:47 -7:56 in other NoSQL datastore) (Irrespective of the use cases 8:18 - 9:15 for storing data) ( Secondly, NoSQL data 9:29 - 9:57 data Dynamically)**

**B) CAP Theorem**

When the data volume increases, and it becomes sufficiently large to be stored and managed by a single machine, the data is partitioned and is distributed across a cluster of machines. The CAP theorem is a tool used to make system designers aware of the trade-offs while designing networked shared-data systems. In CAP, 'C' refers to consistency, 'A' refers to availability, and 'P' refers to partition tolerance. Consistency guarantees that every node in the distributed system returns the same, most successful, recent write. Availability refers to every request receives a response without the guarantee that it contains the most recent write and lastly partition tolerance refers to the system continues to function and upholds its consistency guarantees in spite of network partitions. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals. **(12:11 When the data - 15:20 Garbage Message)**



The CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the three guarantees mentioned. They are Consistency, Availability and Partition tolerance. Systems which guarantees consistency and availability are RDBMS systems which support SQL. As NoSQL systems store data in a distributed manner across a cluster of interconnected machines, they provide network partitioning. Hence, there are two flavours of NoSQL databases, one which guarantees consistency and partition tolerance and the second one guarantees availability and partition tolerance. The first flavour which guarantees consistency over availability will report an error or a time-out if particular information is not latest due to network partitioning. Let's say you are 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 time out. 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 choosing availability over consistency, the system processes the query for sure and tries to return the most recent available version of the information, even if it the data is not latest due to network partitioning. Let's say you are 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. A point to be noted over here is, in the absence of network failure both consistency and availability are satisfied by a distributed data store. **(When choosing 15:49 - 17:12 distributed data store)**

**C) Introduction to HBase**

* History of Hbase.
  + Who developed it
  + Like inspired by Google’s Big Table etc.

HBase is a distributed data store built on top of HDFS. Hence it holds a prominent position in Hadoop ecosystem. This means HBase can leverage all the benefits provided by Hadoop or HDFS. Some of the benefits which HBase has because of HDFS are distributed processing, horizontal scalability and fault tolerance because of replication. But beyond its Hadoop roots, HBase is a powerful database because of some features which are unique to HBase only. In short, 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.

In early 2000's as usage of internet increased drastically, the number of resources available on the web started to grow exponentially. So there was a daunting task in front of the search engine giant Google, i.e. How could it provide timely search results across the entire Internet? Already the number of resources available on the internet was huge but again the number is not static, it has the potential of increasing exponentially. So to solve this problem, Google defined the following technologies: Google File System, BigTable and MapReduce. In 2007, Mike Cafarella released code for an open source BigTable implementation that he called HBase. The first HBase release was integrated with Hadoop 0.15.0 in October 2007. In May 2010, HBase became a top-level Apache project that runs on Facebook, Twitter, and Adobe, just to name a few. **(17:59 Hbase is a distributed - 20:30 Just to name a few)**

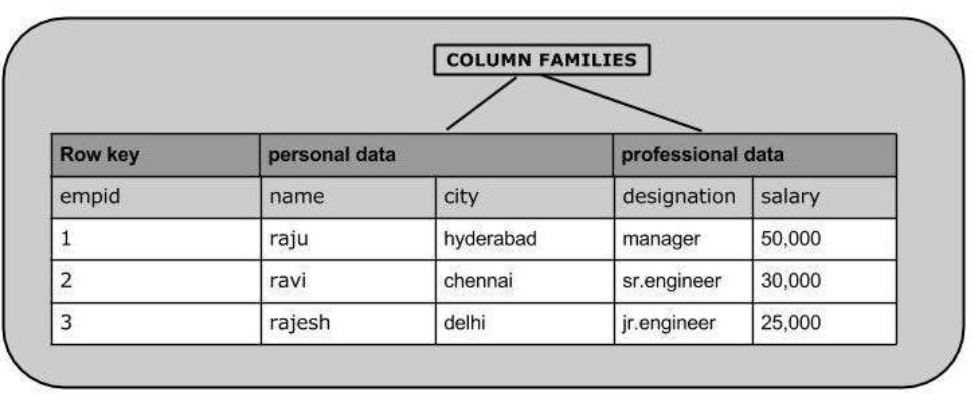
* Features of Hbase

Let's discuss some features of HBase which differentiate it from Hadoop and some other relational or NoSQL data stores. Firstly, HBase is a key-value store which stores data internally as key, value pairs where the rowkey is the key and rest of the data is value. Secondly, the records in HBase are sorted by rowkey. Thirdly, the columns in an HBase do not have a specific data type. All data in HBase are stored in form of bytes. Lastly, HBase tables do not follow a strict schema which means any number of columns can be added dynamically.

**(Let us discuss 20:40 - 21:04 data is value) ( 21:16 Secondly, the records - 21:45 dynamic manner)**

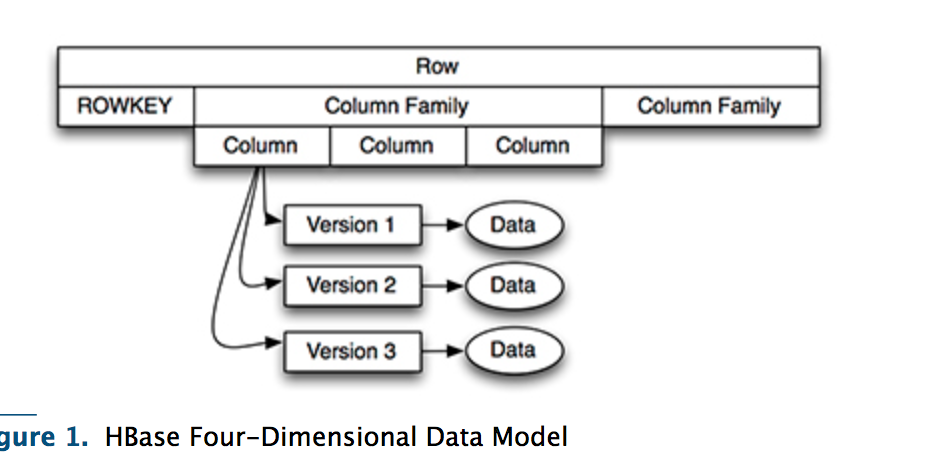
**D) Data Model Used in Hbase**

Along with professor’s voice over will be highlighting relevant sections in the below image:



Like SQL, HBase also stores data in tabular format with some modifications. In HBase tables, instead of columns, there are 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. This feature ensures that the schema of an HBase table is flexible and the table can scale linearly. Every row in an HBase table is associated with a rowkey. Entire data in HBase table is sorted as per the row key. Records in an HBase table are fetched as per the rowkey. Also, each column can have a configurable number of versions and there is a provision for selecting data for a particular version. In HBase, each version is identified by a timestamp.

**(21:58 Like SQL - 4 dimensional model 23:57)**



In this figure, you can see an individual row is reached through its row key and is comprises of one or more column families. Each column family has one or more column qualifiers called columns and each column has one or more versions. To access an individual piece of data, you need to know its row key, column family, column qualifier, and version.

So reiterating what we learnt, 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. Every entry in an HBase table is identified and indexed by a RowKey. A column family is a collection of columns, and for every row key, an unlimited number of columns can be stored.

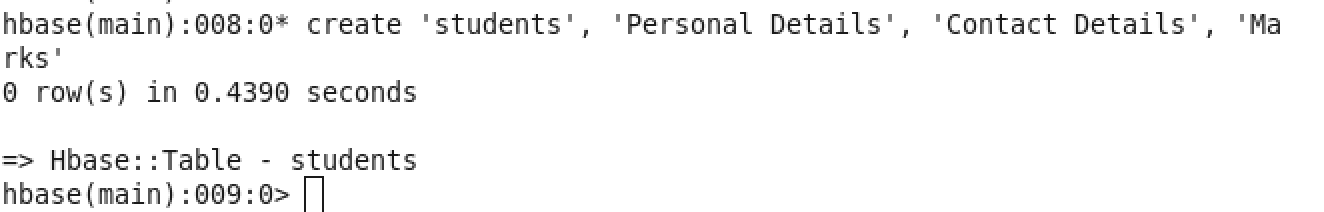
**(So In Summary 24:13 - columns can be stored 24:44)**

**F) Common Operations performed on a HBase table (25% of student time on this section)**

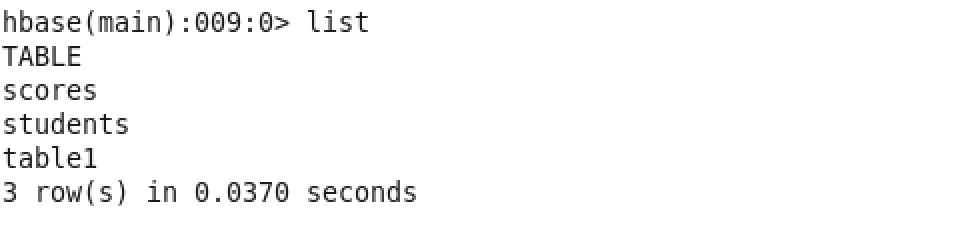
HDFS along with Map Reduce follows write once and read many times paradigm. In other words, data in HDFS is written once, but it can be read an unlimited number of times. Even there is no provision for updating an existing data set or record in HDFS. Let's see how can we perform basic CRUD operations in HBase using HBase shell commands. Please note even if HBase uses HDFS to store data but HBase supports update operations by maintaining multiple versions of same data points.

So let's start by creating a simple HBase table. The command used for creating a table is 'create'. For creating a table you must name the table and define its schema. As a part of the schema, we only need to specify the column families, columns are defined while inserting records into the HBase table. So let's create a table named students with three column families they are Personal Details, Contact Details and Marks. The command for creating a student table in HBase is create table name which is students and list of column families they are Personal Detail, Contact Details, Marks. (HDFS along with 24:56 - 26:40 marks scored in)

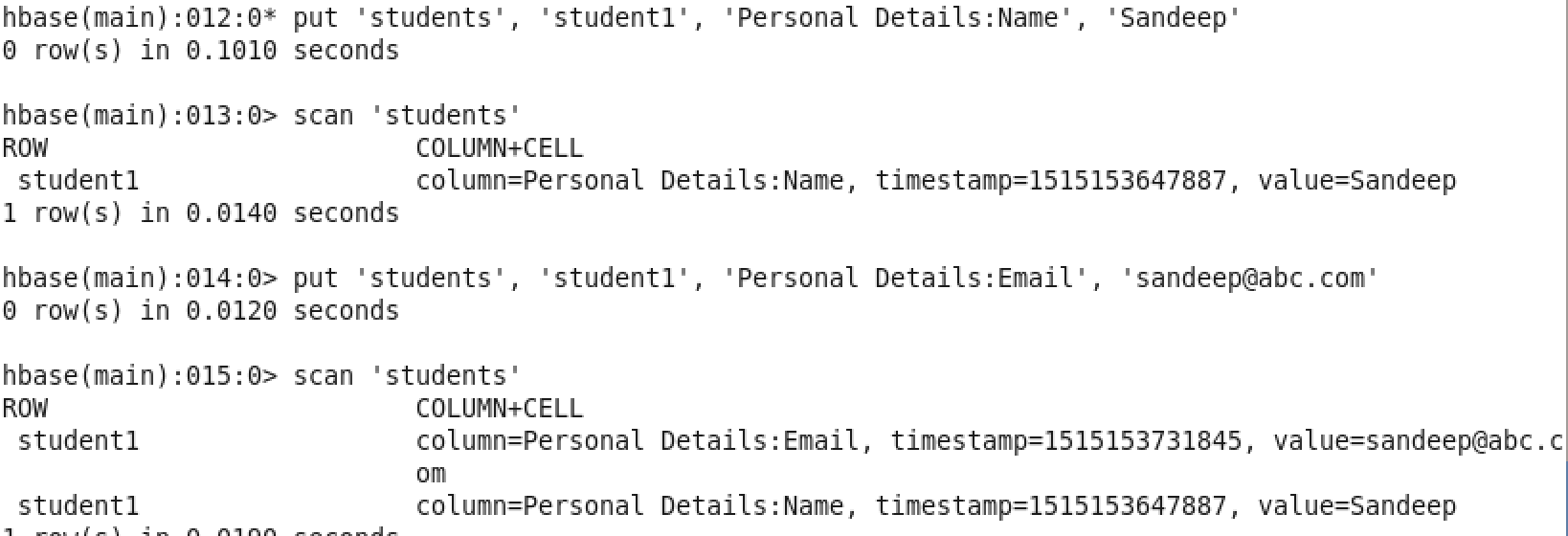
(*26:57* The command - 27:54 put command)



We can check all the tables present in HBase using the list command. We see the table students is created successfully.

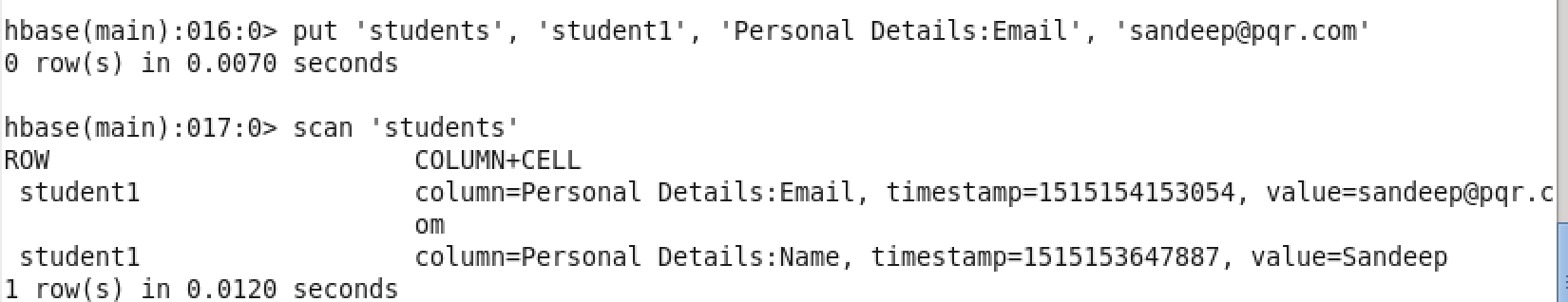


So we created a table named students successfully. Now we will insert some student records into this table. For populating the table cells, we make use of put command. To put a cell value into table ‘t1’ at row ‘r1’ under column ‘c1’ we use the command put ‘t1’, ‘r1’, ‘c1’, ‘value’. Using the same command let's insert information for a student into HBase table. *(27:43)***(To put cell value 29:16-31:50 removed or not)**



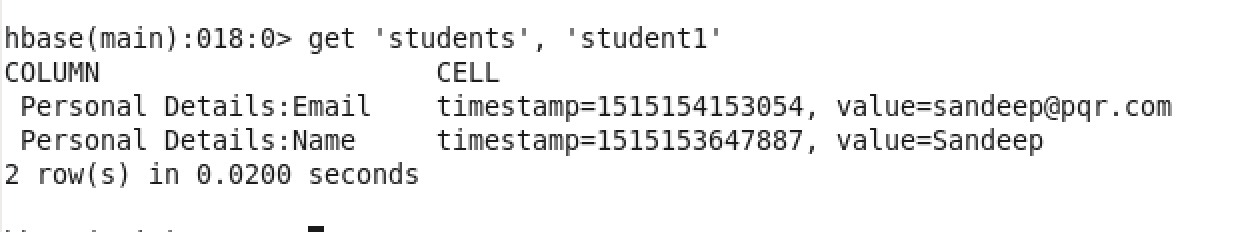
After we insert a data item into the table we can view the table contents using the scan command. scan table name will show all the data items present in the table.

Let's try to update the email id of student1 to sandeep@pqr.com using put command.

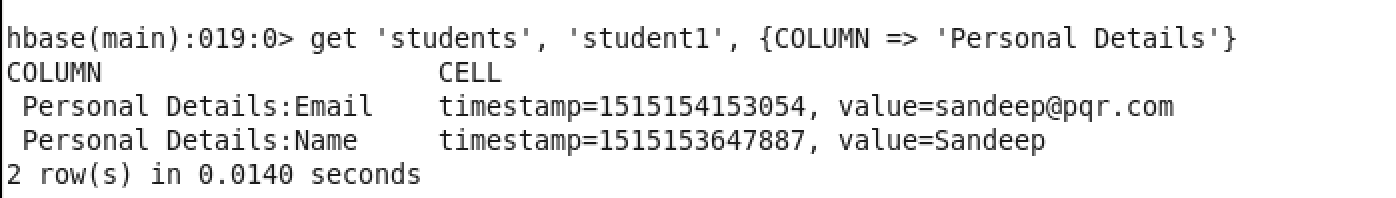


We see that the email id is updated to [sandeep@pqr.com](mailto:sandeep@pqr.com) from [sandeep@abc.com](mailto:sandeep@abc.com)

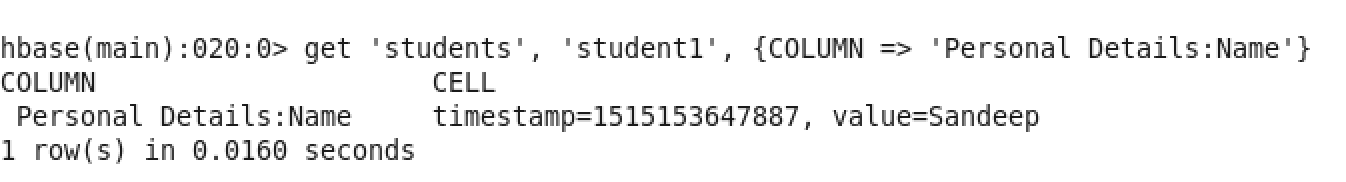
The command which is used to fetch data from HBase is 'get'. There are various ways by which we can use the get command. First of all, we will fetch entire row's data using the get table name and row key command.



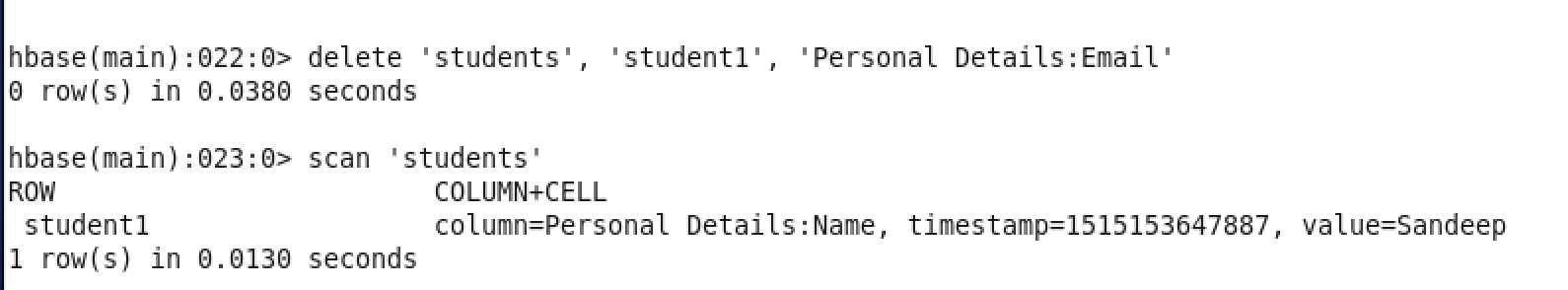
If you want to fetch data for a given column family the command is get table name, row key, keyword COLUMN in the curly braces followed by implies operator and the column family name.



For fetching data for a particular cell, add column family name followed by column name. The column family and column name are separated by a colon(:)



For deleting a cell we will use the delete command. The syntax of delete command is delete, table name i.e students, rowkey i.e. students1, column family name i.e. Personal Details followed by colon and column name i.e. Email



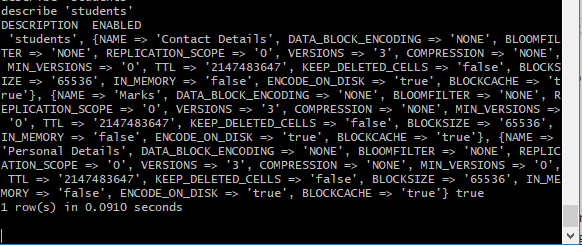
Let’s execute the scan command and see if the data is removed or not.

**Commands to be covered by Divjot**

**DDL:**

**DESCRIBE:**

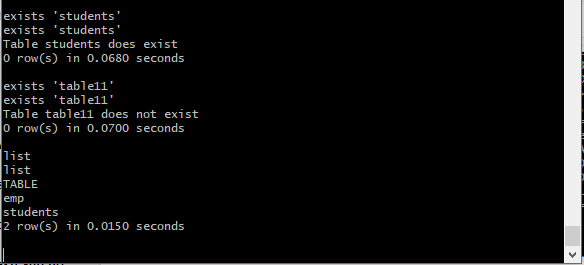
We have already seen how we can create a table using ‘create’ command. Now if we wish to check out the schema of the table we created, we can do so using ‘describe’ command. Let us use the ‘students’ table and describe it.



Here we can see on the screen that the table describes the schema of the various column families of table ‘students’. We have specified three column families for the table namely ‘Contact Details’, ‘Personal Details’ and ‘Marks’. The command displays the comma separated key value pairs for each column family. The key value pairs contain the information about the specific column family like ‘REPLICATION\_SCOPE’, ‘VERSIONS’ etc. So basically the ‘describe’ command prints the schema of the table on the screen.

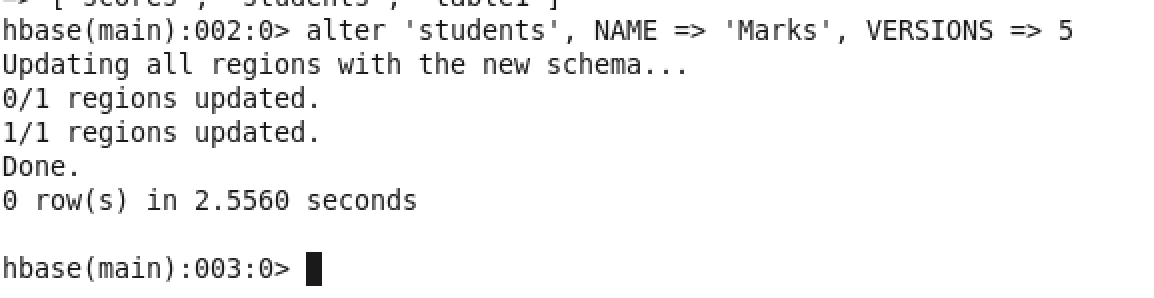
**EXISTS:**

The exists command shows whether or not the given table is present in the hbase or not. Here we can see that because we have already created a table named ‘students’, ‘exists’ command on that table prints that the table is present. While when we try to use this operator on a table that is not present, it prints that the table is not present.

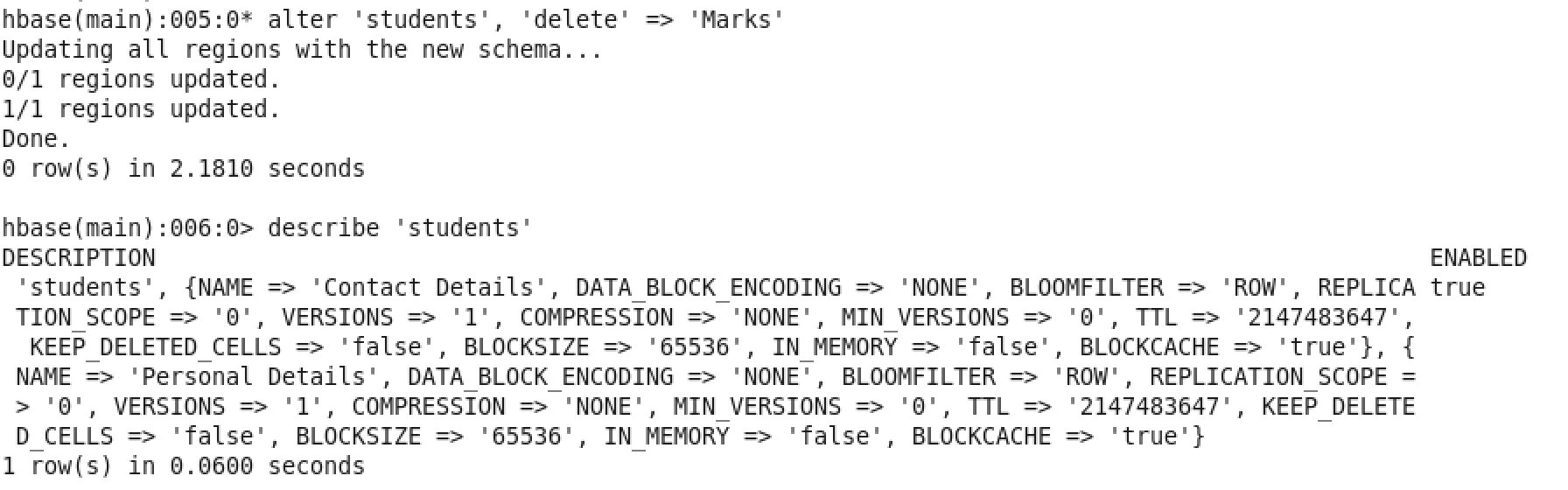


**ALTER:**

Alter command is basically used to modify the existing table. Like if we wanted to modify the number of versions of ‘Marks’ column family to 5, we could use this command.

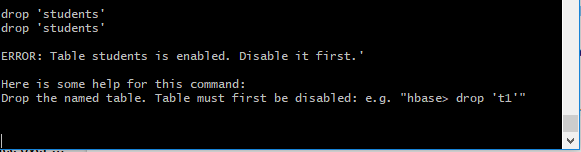


Other than VERSIONS, we can alter many other properties of the table. We can also delete the column family by applying delete method on it. Like this:

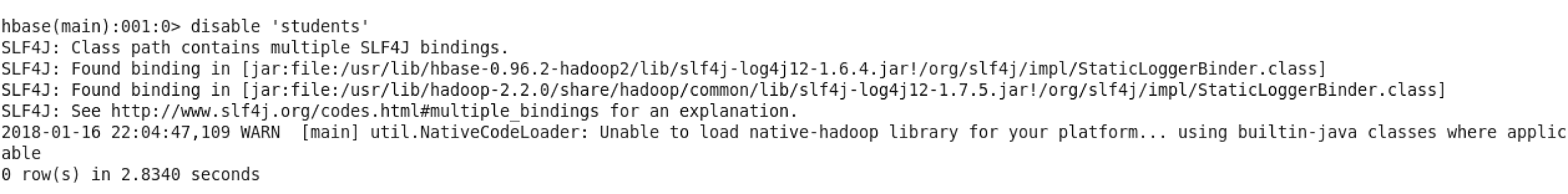


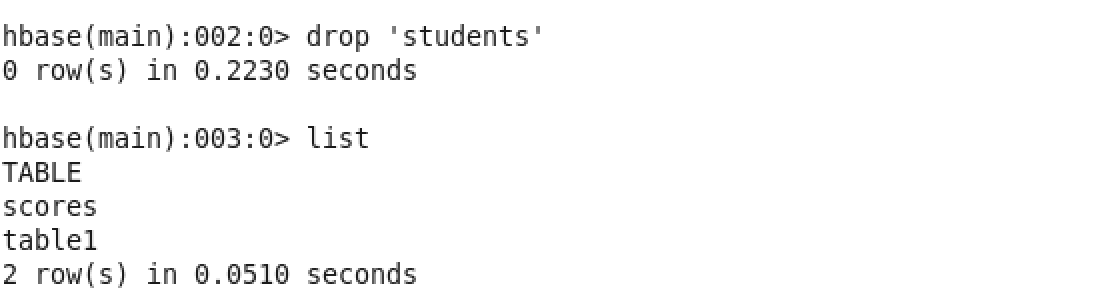
**DROP:**

Drop table is an operator that is used to delete the tables. But this operator cannot be applied directly to the table. Instead, the table is first disabled. And then it is dropped. Like this

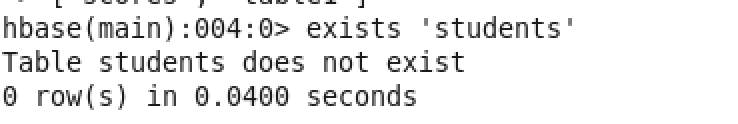
****

Now if we use disable first and then drop. The table is deleted.



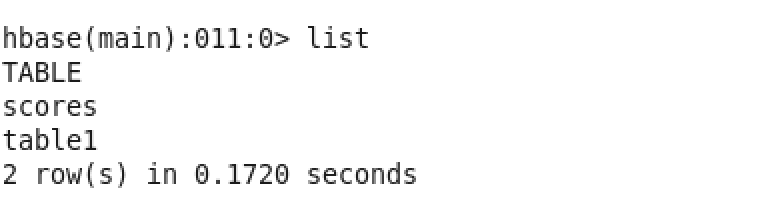


Now let us verify if the table is actually deleted or not using the list command and the exists command..

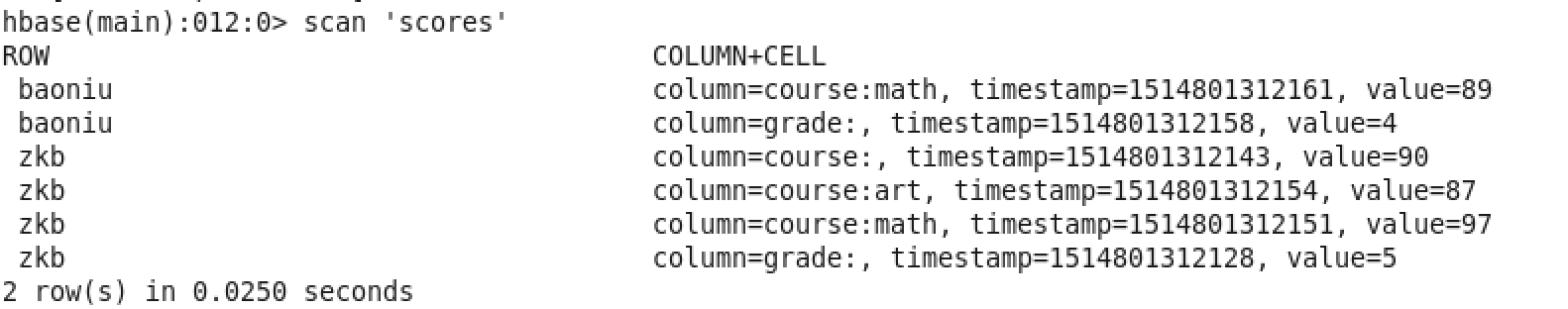
****

**Truncate:**

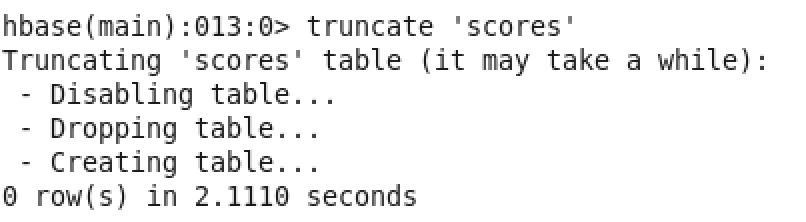
This command is basically used to remove all the data from the table. Note that we do not intend to delete the table, just the data. Internally, this command disables the table, drops it and then again recreates it but for us, the end result is that the table’s data has been removed. Have a look. Let me fire a list command and check what all tables are present :



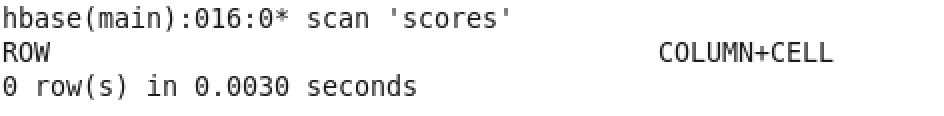
Here we see there are two tables scores and table1. Using the scan command let’s check if there are any records present in scores table. If not discussed before, scan() command is used to display all the rows present in the table.



We see there are two rows. Let’s execute the truncate command on this table and check the outcome.



As discussed the truncate command first disabled the table, then dropped the table and recreated the same table for us. Now, ideally the table scores should be empty. Let’s find out by executing the scan command on this table:



Indeed there are 0 rows now in the table.

**JAVA Implementations:**

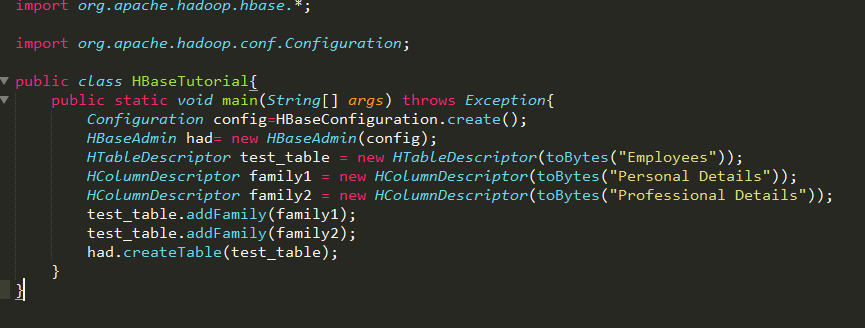
Instead of using the command shell, we can also use the JAVA API’s to perform similar operations. For that, you need to import apache.hadoop.hbase package and apache.hadoop.conf package in your code.

Now let us create a table using the JAVA API’s like this.

Here we first need to create HBase configuration using HBaseConfiguration. Now create an instance of HBase admin passing this configuration to it. Now we can perform other operations using this hbase admin instance.

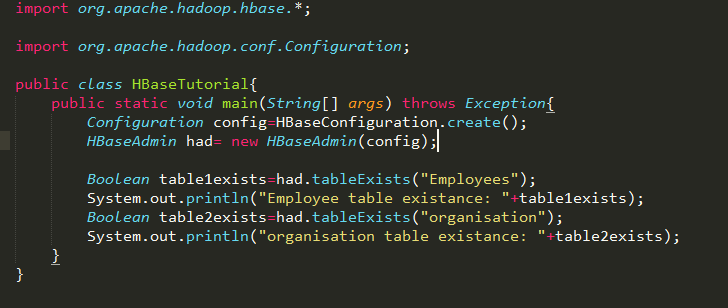
Now to create a table we first need to create a table descriptor that takes the parameter as the name of the table. After it, we add the column families to that descriptor using addFamily method of the table descriptor instance. The column families are an instance of HColumnDescriptor class.

After adding the families, we create the table using createTable method on the hbase admin instance.



**To check if a table Exists in HBase:**

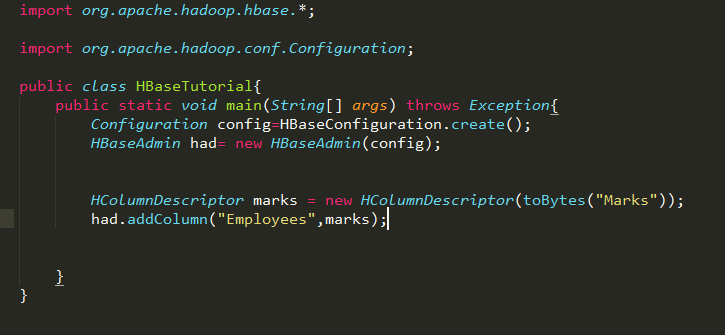
To check if a table exists or not, we use the method tableExists on the hbase admin like this:

****

Here, it takes the parameter as the table name and returns a boolean value which is true if the table exists and false if it doesn't.

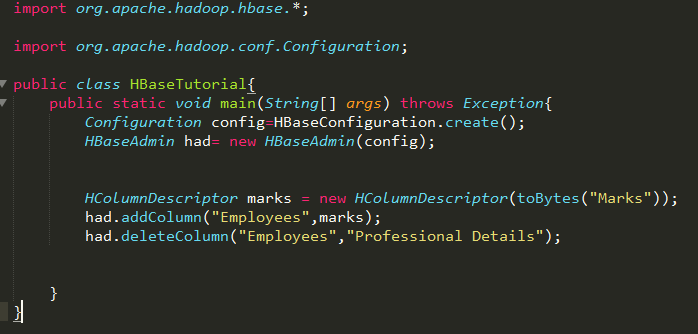
**Adding a column in a table:**

To add a column, we use addColumn method on the hbase admin instance. This method takes the parameters as the table name and the column descriptor. Have a look:



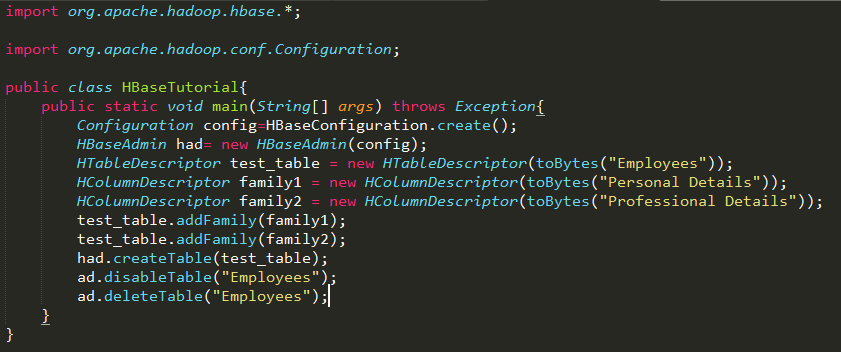
**Deleting a column in a table:**

Similarly, to delete the column, we use the deleteColumn method on the hbase admin instance and this method takes the parameters as the table name and the column family name to be deleted. Have a look:



**Dropping a table:**

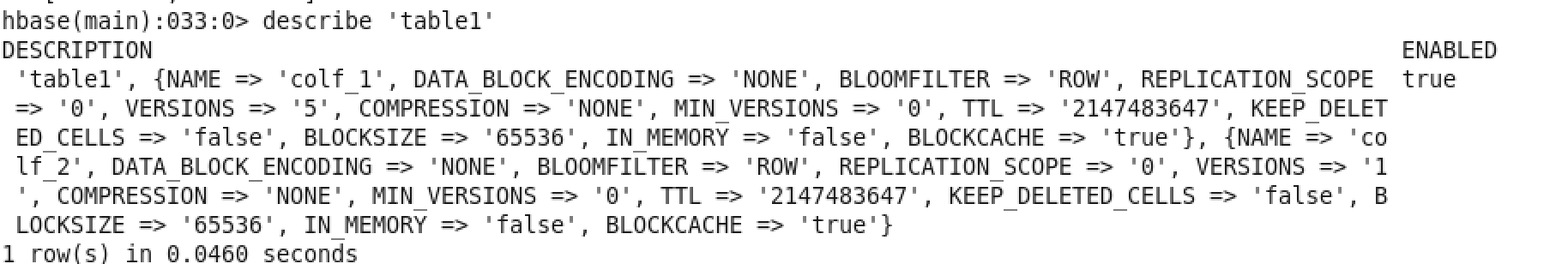
To drop the table, as we did in shell, it is mandatory to disable the table first. We do this by using the disableTable method of the HBaseAdmin class and then we use the delete table method of the same class. Both the methods take the parameter as the name of the table to be dropped. Have a look.



**DML:**

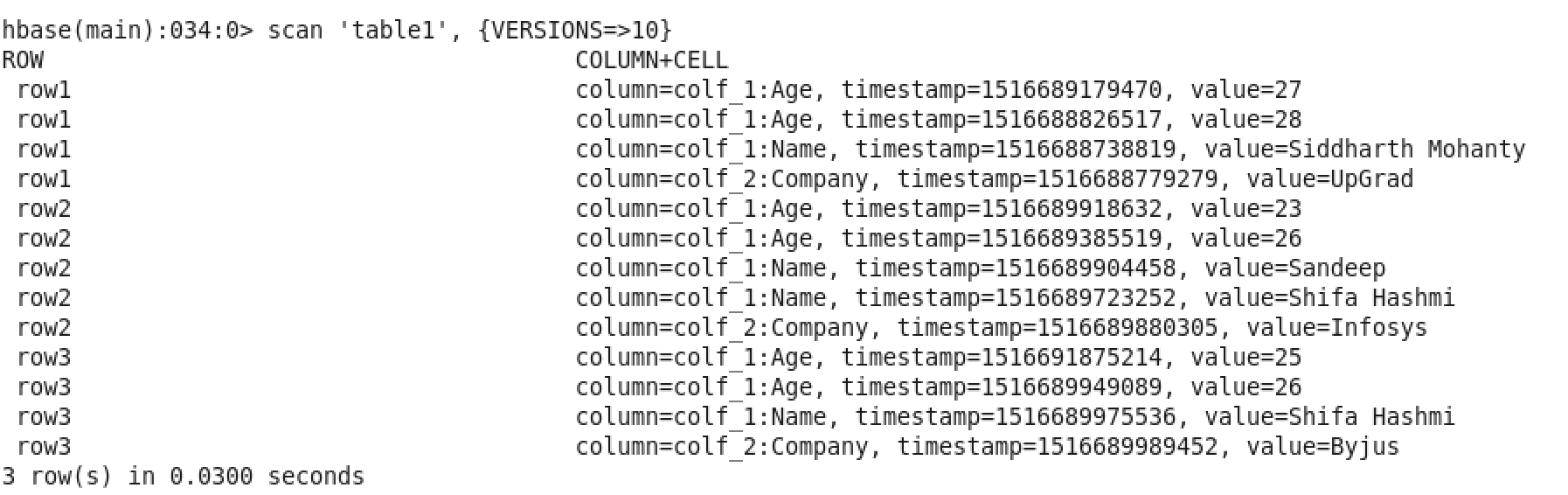
**Multiple put on the same cell. Demonstrating how hbase maintains multiple versions of data by timestamp**

In the previous sessions, you learnt that HBase is capable of storing multiple versions of the data present in a single cell. Let's try and see how that is possible in HBase. By default, column families in an HBase table do not support storing multiple versions of data. We can confirm this by checking the table schema by firing a describe command in the console. We already have a table named 'table1' . Let's describe it and see its properties.



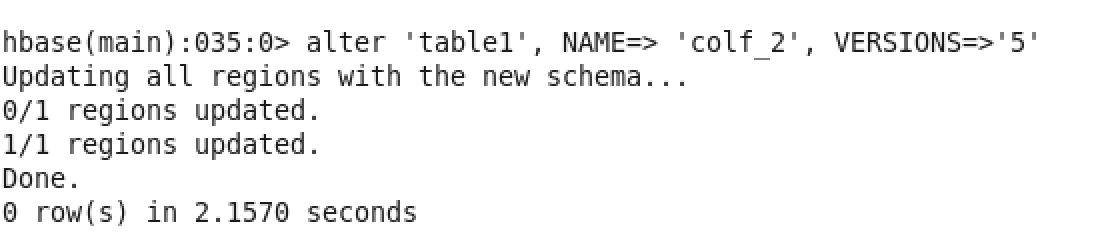
Here we see that the version count for colf\_1 is 5 and version count for colf\_2 is 1 by default. The version count for colf\_1 is 5 because I had changed this manually before.

Previously we saw how scan command is used to retrieve all the rows present in the table. To see all the versions of a cell, we will execute a different variant of scan command i.e. scan 'table1',{VERSIONS => 10}. VERSIONS => 10 specifies the maximum number of versions which has to be displayed.

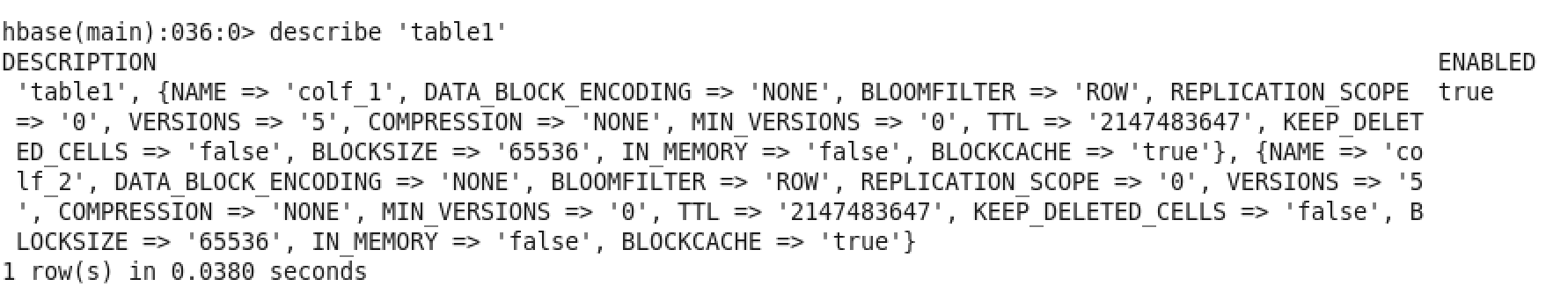


As checked before multiple versioning was allowed for colf\_1. Hence, for all the three rows we see multiple versions in age column. For row2, we even saw multiple versions in name column as well. As checked before multiple versioning was allowed for colf\_1.

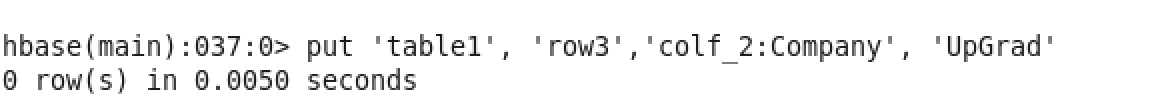
Now, let’s enable multiple version feature for colf\_2 as well. For that we need to fire the following alter command.



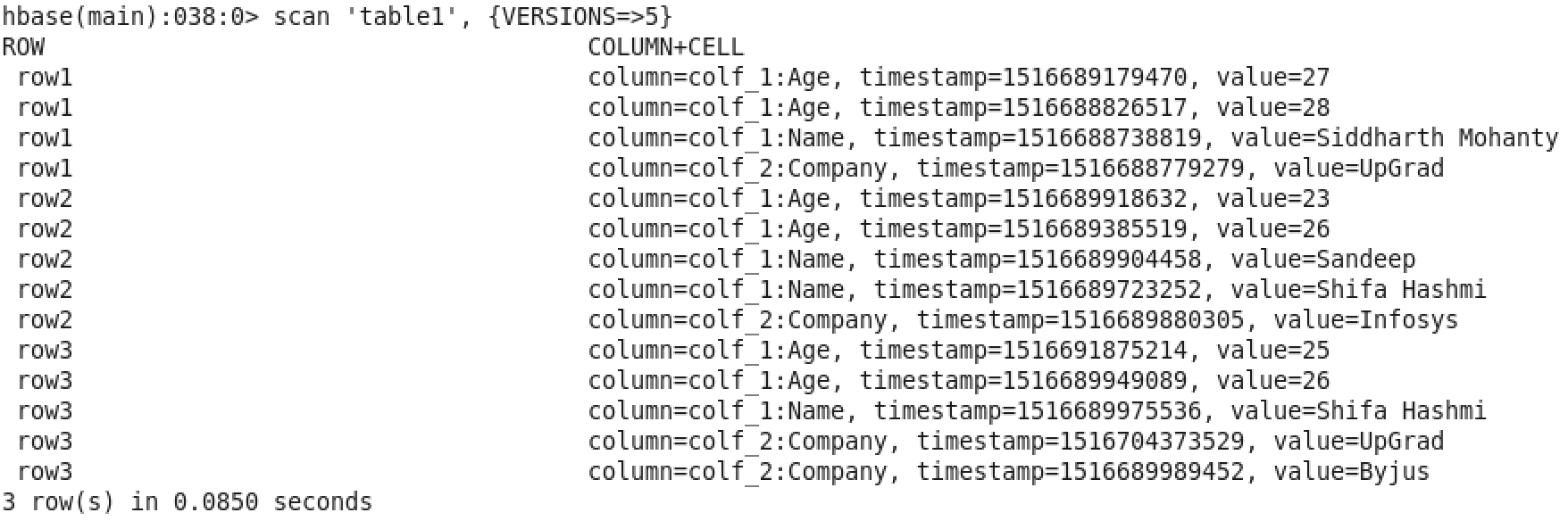
Let’s verify if the version number has changed for colf\_2 or not by executing the describe command:



We see for colf\_2 the version count have changed to 5. Let’s try to update the company of row3 using put command.



Now let’s execute the scan ‘table1’, {VERSIONS=>5} command to check all the versions of the data present in the table.

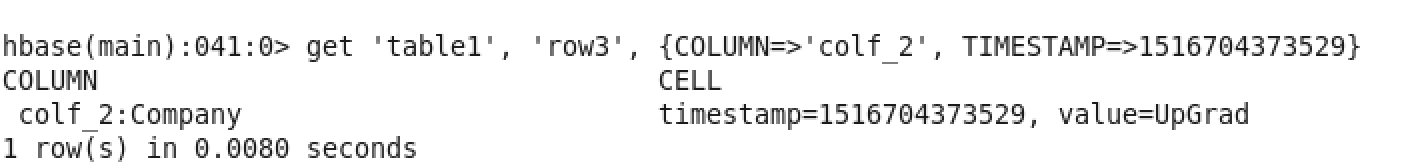


Indeed we can see for row3 and column company there are two value UpGrad and Byjus.

**GET:**

* **Get data based on timestamp**

In HBase, if you fire a scan() command then only latest records are displayed. Past versions can be retrieved based on timestamp using the command get ‘table1’ which is the table name, ‘row3’ the rowkey and in curly braces additional parameters such as column family name and the exact timestamp is specified.

****

* **Get data based on a filter condition**

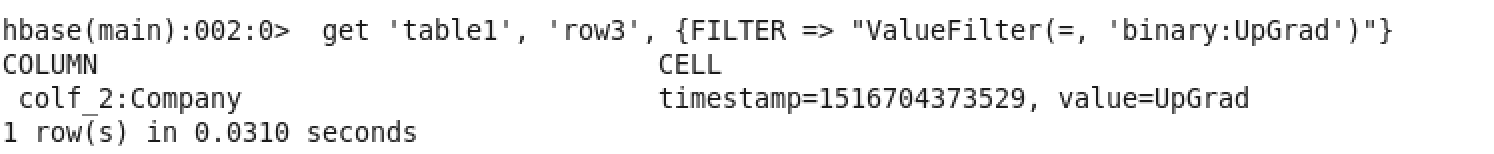
In HBase, fetching data based on a filtering condition is achieved by using Filters. In HBase, filters are like java methods which take two input parameters they are a logical operator and a comparator. The logical operator specifies the type of the test i.e. equals, less than, etc. The type of filter and comparator together specify the left and right-hand side of the logical expression. Some commonly used filter functions are :

1. ValueFilter
2. QualifierFilter
3. FamilyFilter

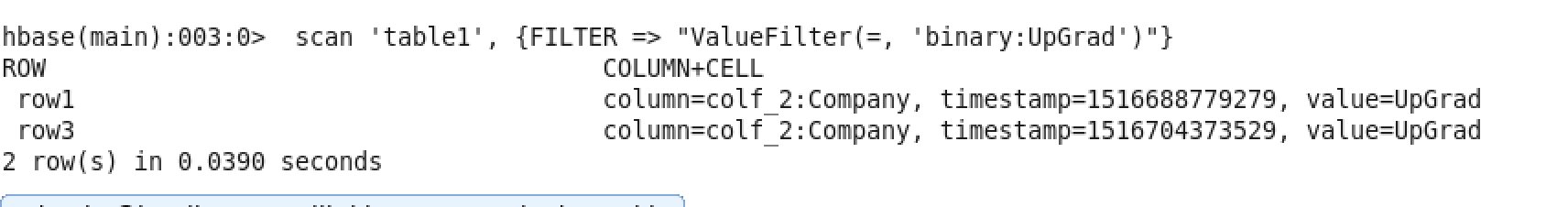
Let’s check how a ValueFilter works.

A ValueFilter takes a comparison operator and comparator as the parameter. It compares each value with the comparator using the comparison operator. If the check is true then result is displayed in the console. To test this lets fire this command

get 'table1', 'row3', {FILTER => "ValueFilter(=, 'binary:UpGrad')"}



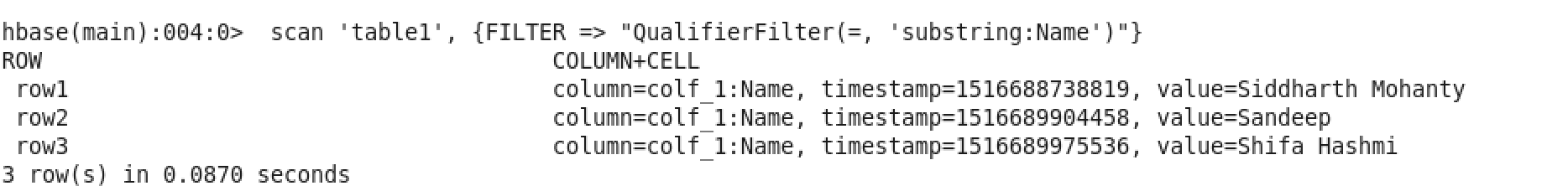
Let’s fire a scan command using ValueFilter



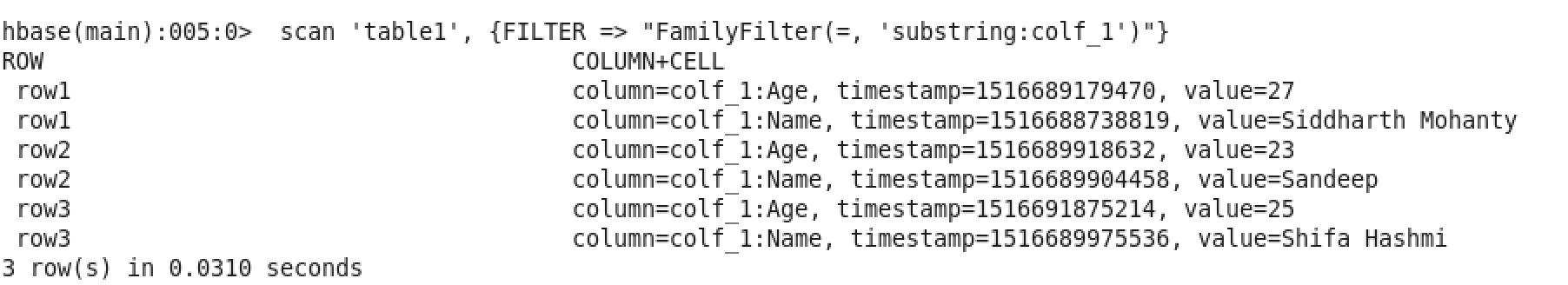
We can see company UpGrad exists for both row1 and row3.

Like ValueFilter, 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.

This column qualifier can be used to fetch an entire column’s data. Let’s see how we can fetch all the data present in column “Name”. We can do this by firing scan command with the QualifierFilter. The qualifier filter takes ‘=’ operator as the operator and “substring:Name” as comparator where “Name” is the qualifier name.

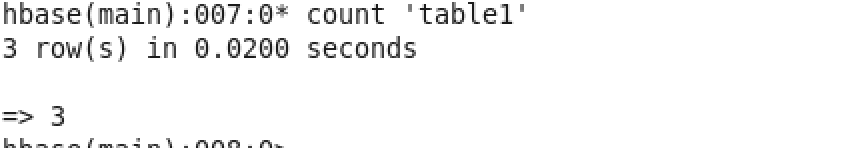


Similarly, familyfilter is used to fetch key-values for a specified column family. Let’s check that as well



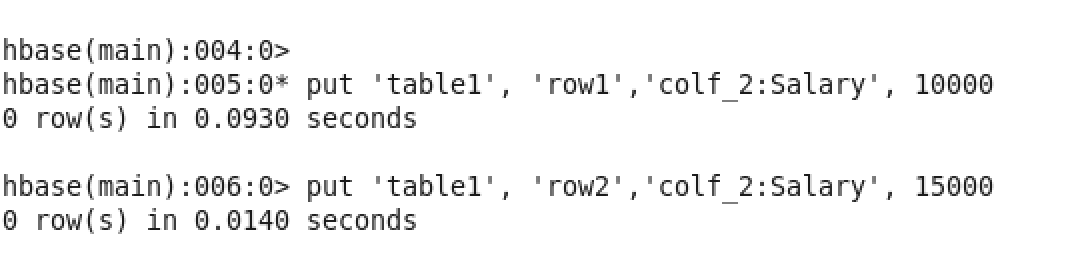
**COUNT:**

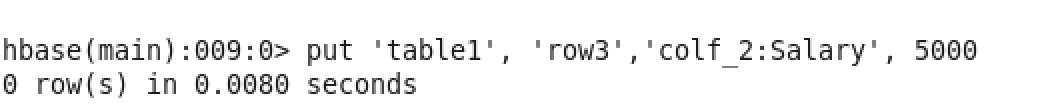
Count command is used to count the number of rows present in the table. It can be executed using count ‘table1’



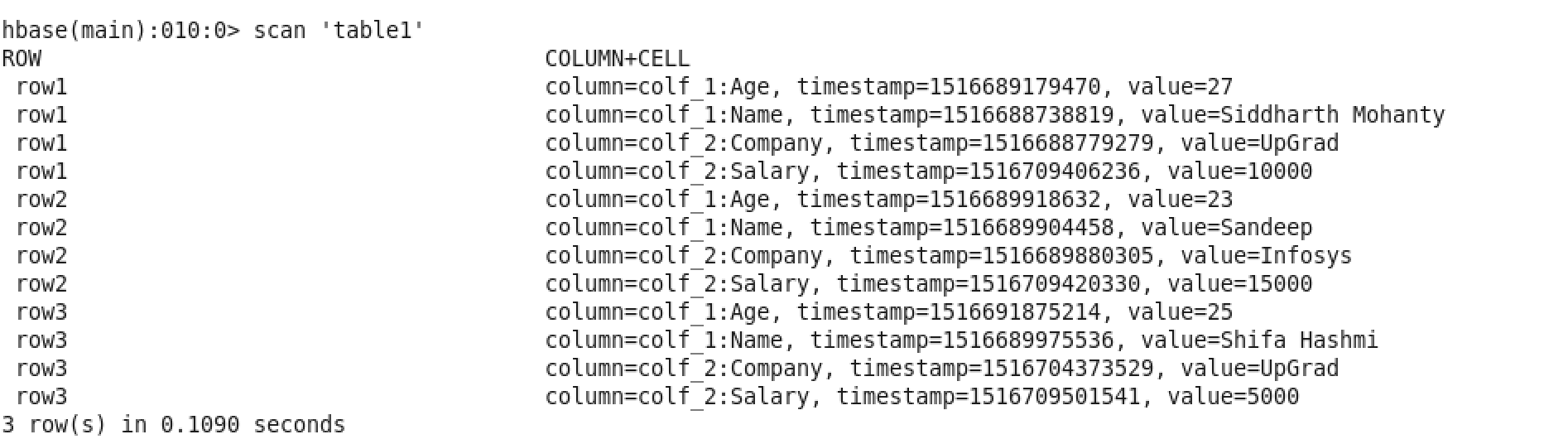
**INCR:**

INCR Is used to increment the value of a cell. In our table ‘table1’ we have an age field. Let’s add a new column salary for all the three rows in table1.





Let’s validate the same using scan command:



We could clearly see that column Salary is successfully added for all the three rows.

Let’s try to increase the salary for each row by 5000 using INCR command. This can be done by firing the command INCR ‘table1’, ‘row1’, ‘colf\_2:Salary’, 5000

**G) Java representation of all operations (25% of student time on this section)**

HBase like Hadoop is written in java. So we can also perform the operations discussed in the previous discussion using Java. Before going through the code, let me introduce some important classes along with their relevance. The first class is "HBaseConfiguration". This class is used to create a configuration object. The static method "create" defined in HBaseConfiguration class is used to return a configuration object. This configuration object will be used for creating tables. The configuration object points the code to that cluster where they are supposed to be executed. **( Hbase like Hadoop See 32:13 - 33:40 in Hbase Table)**

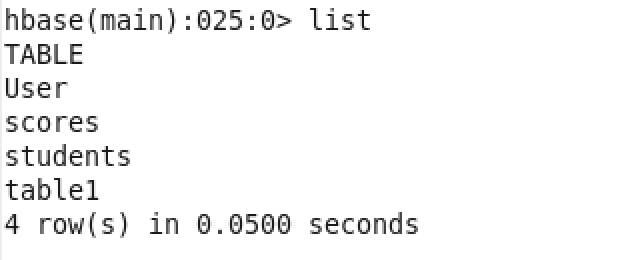
The next class is HBaseAdmin. This class is used to administer or manage the HBase cluster. Table creation and deletion are taken care by HBaseAdmin class. The class HTable is used to operate on a table. **(A put object 33:50 - for a rowkey 34:16)**

Objects of Put class are used for entering data into the HBase table. Each Put puts a single cell value in the Hbase table. A put object is created for a particular rowkey. Using the add method which takes the columnFamily, qualified and value a value is added in the HBase table. Objects of the get class are used to fetch an entire row from the table. A get object is created for a rowkey.

First, let's see how we will be creating a table named User with column families as ID and Name.

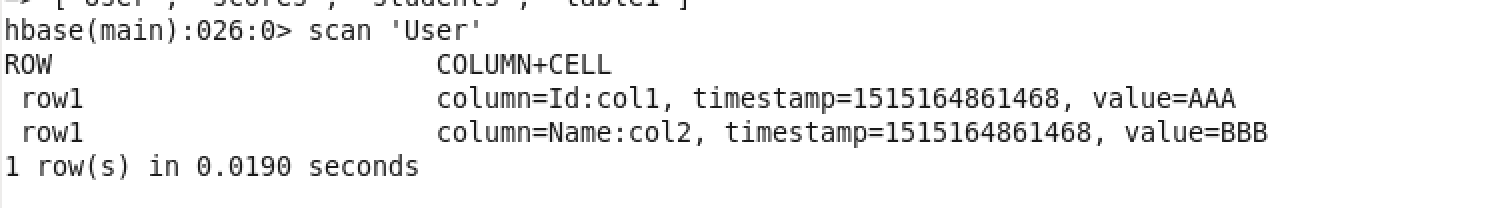
**(34:41 First Let’s See - 34:53 ID and Name)**

**<Profesor to go through the first code and highlight HBaseAdmin table which is used to create a HBase table. In the shell he can run the list command and highlight that the user table is created>**



Now as we have successfully created an HBase table we will put some values into the table using HTable and Put classes. Let's add the value AAA under column family ID and column qualifier col1. Similarly, lets add BBB under column family Name and column qualified col2.

**<Professor should do a code walk through line by line only for the put action. Comment the get part of the code and execute only the put section which inserts the required data.>**



Now as we have already inserted the data into the table we shall see how to fetch the data from a table. Typically, there are two ways of doing it. Firstly, we can use get method to retrieve row-wise data and secondly we can use a scanner to retrieve all the records in one go. A scanner is implemented using the ResultScanner class in java. Let's see how we are doing it programmatically in java.

**<Professor should do a code walk through line by line only for the get action. Comment the put part of the code and execute only the get section which fetches the rows>**

**(Now let us see 35:26 - 43:34 files are there are not)**

**(Let us now give list 43:49 - 44:49 column 2 Value is BBB)**

**(You can see values being displayed here 44:34 to 44:49): use Screengrab from (10:23 Scan “TableDemo4” to 10:38)**

**(Now what we saw 45:15 - 46:01 No sql Database)**