1. **What is NoSQl database?**

Ans. The database technology can be divided into two types in terms of how they are built-

a. Relational Database
b. Non-Relational Database

Relational databases are structured, like phone books that store phone numbers and addresses. Non-relational databases are document-oriented and distributed, like file folders that hold everything from a person's address and phone number to their Facebook likes and online shopping preferences.

We call them SQL and NoSQL, referring to whether or not they're written solely in structured query language (SQL).

NOSQL DATABASES: NON-RELATIONAL & DISTRIBUTED DATA

If our data requirements aren't clear at the outset or if we're dealing with massive amounts of unstructured data, we may not have the luxury of developing a relational database with clearly defined schema. Unstructured data from the web can include sensor data, social sharing, personal settings, photos, location-based information, online activity, usage metrics, and more. Trying to store, process, and analyze all of this unstructured data led to the development of schema-less alternatives to SQL.

Taken together, these alternatives are referred to as NoSQL, meaning "Not only SQL." Now today NoSQL databases have become the first alternative to relational databases, with scalability, availability, and fault tolerance being key deciding factors. They go well beyond the more widely understood legacy, relational databases (such as Oracle, SQL Server and DB2 databases) in satisfying the needs of today's modern business applications. A very flexible and schema-less data model, horizontal scalability, distributed architectures, and the use of languages and interfaces that are "not only" SQL typically characterize this technology.

Types of NoSQL Databases

There are four general types of NoSQL databases, each with their own specific attributes:

- **Graph database** – Based on graph theory, these databases are designed for data whose relations are well represented as a graph and has elements which are interconnected, with an undetermined number of relations between them. Examples include: Neo4j and Titan.
- **Key-Value store** – we start with this type of database because these are some of the least complex NoSQL options. These databases are designed for storing data in a schema-less way. In a key-value store, all of the data within consists of an indexed key and a value, hence the name. Examples of this type of database include: Cassandra, DyanmoDB, Azure Table Storage (ATS), Riak, BerkeleyDB.
- **Column store** – (also known as wide-column stores) instead of storing data in rows, these databases are designed for storing data tables as sections of columns of data, rather than as rows of data. While this simple description sounds like the inverse of a standard database, wide-column stores offer very high performance and a highly scalable architecture. Examples include: HBase, BigTable and HyperTable.
- **Document database** – expands on the basic idea of key-value stores where "documents" contain more complex in that they contain data and each document is assigned a unique key, which is used to retrieve the document. These are designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Examples include: MongoDB and CouchDB.

Popular NoSQL Databases

- **MongoDB**— It is the most popular NoSQL system. A document-oriented database with JSON-like documents in dynamic schemas instead of relational tables that's used on the back end of sites like Craigslist, eBay, Foursquare. It's open-source, so it's free, with good customer service.
- **Apache's CouchDB**—The true DB for the web, it uses the JSON data exchange format to store its documents; JavaScript for indexing, combining and transforming documents; and, HTTP for its API.
- **HBase**—The Apache project, developed as a part of Hadoop, this open-source, non-relational "column store" NoSQL DB is written in Java, and provides BigTable-like capabilities.

- **Oracle NoSQL**—This is Oracle's entry into the NoSQL category.
- **Apache's Cassandra DB**—It was developed at Facebook, Cassandra is a distributed database that's great at handling massive amounts of structured data. Anticipate a growing application? Cassandra is excellent at scaling up. Examples: Instagram, Comcast, Apple, and Spotify.

2. How does data get stored in NoSQl database?

Ans. There are various NoSQL Databases. Each one uses a different method to store data. NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed. More sophisticated NoSQL databases allow validation rules to be applied within the database, allowing users to enforce governance across data, while maintaining the agility benefits of a dynamic schema.

Several different varieties of NoSQL databases have been created to support specific needs and use cases. These fall into four main categories:

- **<u>Key-value data stores:</u>** Key-value NoSQL databases emphasize simplicity and are very useful in accelerating an application to support high-speed read and write processing of non-transactional data. Stored values can be any type of binary object (text, video, JSON document, etc.) and are accessed via a key. The application has complete control over what is stored in the value, making this the most flexible NoSQL model. Data is partitioned and replicated across a cluster to get scalability and availability. For this reason, key value stores often do not support transactions. However, they are highly effective at scaling applications that deal with high-velocity, non-transactional data.
- **<u>Document stores:</u>** Document databases typically store self-describing JSON, XML, and BSON documents. They are similar to key-value stores, but in this case, a value is a single document that stores all data related to a specific key. Popular fields in the document can be indexed to provide fast retrieval without knowing the key. Each document can have the same or a different structure.
- **<u>Wide-column stores:</u>** Wide-column NoSQL databases store data in tables with rows and columns similar to RDBMS, but names and formats of columns can vary from row to row across the table. Wide-column databases group columns of related data together. A query can retrieve related data in a single operation because only the columns associated with the query are retrieved. In an RDBMS, the data would be in different rows stored in different places on disk, requiring multiple disk operations for retrieval.
- **<u>Graph stores:</u>** A graph database uses graph structures to store, map, and query relationships. They provide index-free adjacency, so that adjacent elements are linked together without using an index.

Because of the way they are structured, relational databases usually scale vertically – a single server has to host the entire database to ensure acceptable performance for cross- table joins and transactions. This gets expensive quickly, places limits on scale, and creates a relatively small number of failure points for database infrastructure. The solution to support rapidly growing applications is to scale horizontally, by adding servers instead of concentrating more capacity in a single server.

'Sharding' a database across many server instances can be achieved with SQL databases, but usually is accomplished through SANs and other complex arrangements for making hardware act as a single server. Because the database does not provide this ability natively, development teams take on the work of deploying multiple relational databases across a number of machines. Data is stored in each database instance autonomously. Application code is developed to distribute the data, distribute queries, and aggregate the results of data across all of the database instances. Additional code must be developed to handle resource failures, to perform joins across the different databases, for data rebalancing, replication, and other requirements. Furthermore, many benefits of the relational database, such as transactional integrity, are compromised or eliminated when employing manual sharding.

NoSQL databases, on the other hand, usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool. Data and query load are automatically balanced across servers, and when a server goes down, it can be quickly and transparently replaced with no application disruption.

3. **What is column family in HBase?**

Ans. In HBase row columns are grouped into column families. All column families have a common prefix, so, for below example, the table shows two column families: CustomerName and ContactInfo. When creating a table in HBase, the developer or administrator is required to define one or more column families using printable characters. The qualifying tail, the column family *qualifier*, can be made of any arbitrary bytes.

Generally, column families remain fixed throughout the lifetime of an HBase table but new column families can be added by using administrative commands. The official recommendation for the number of column families per table is three or less.

Physically, all column family members are stored together on the filesystem. Because tunings and storage specifications are done at the column family level, it is advised that all column family members have the same general access pattern and size characteristics.

Row Key	Column Family: {Column Qualifier:Version:Value}
00001	CustomerName: {'FN': 1383859182496:'John', 'LN': 1383859182858:'Smith', 'MN': 1383859183001:'Timothy', 'MN': 1383859182915:'T'} ContactInfo: {'EA': 1383859183030:'John.Smith@xyz.com', 'SA': 1383859183073:'1 Hadoop Lane, NY 11111'}
0002	CustomerName: {'FN': 1383859183103:'Jane', 'LN': 1383859183163:'Doe', ContactInfo: { 'SA': 1383859185577:'7 HBase Ave, CA 22222'}

4. **How much maximum number of columns can be added to HBase table?**

Ans. Hbase does not have any such limit to define maximum number of column families in a table but it is officially recommended to use a maximum of 3 column families or below them as much as possible. Currently, flushing and compactions are done on a per Region basis so if one column family is carrying the bulk of the data bringing on flushes, the adjacent families will also be flushed even though the amount of data they carry is small. When many column families exist the flushing and compaction interaction can make for a bunch of needless i/o (To be addressed by changing flushing and compaction to work on a per column family basis).

Also it is highly suggested to make do with one column family if in our schemas. Only introduce a second and third column family in the case where data access is usually column scoped; i.e. when we query one column family or the other but usually not both at the one time.

However we can add as much column as we want to a column family. There is no specific limit on the number of columns in a column family. Actually you can have millions of columns in the single column family.

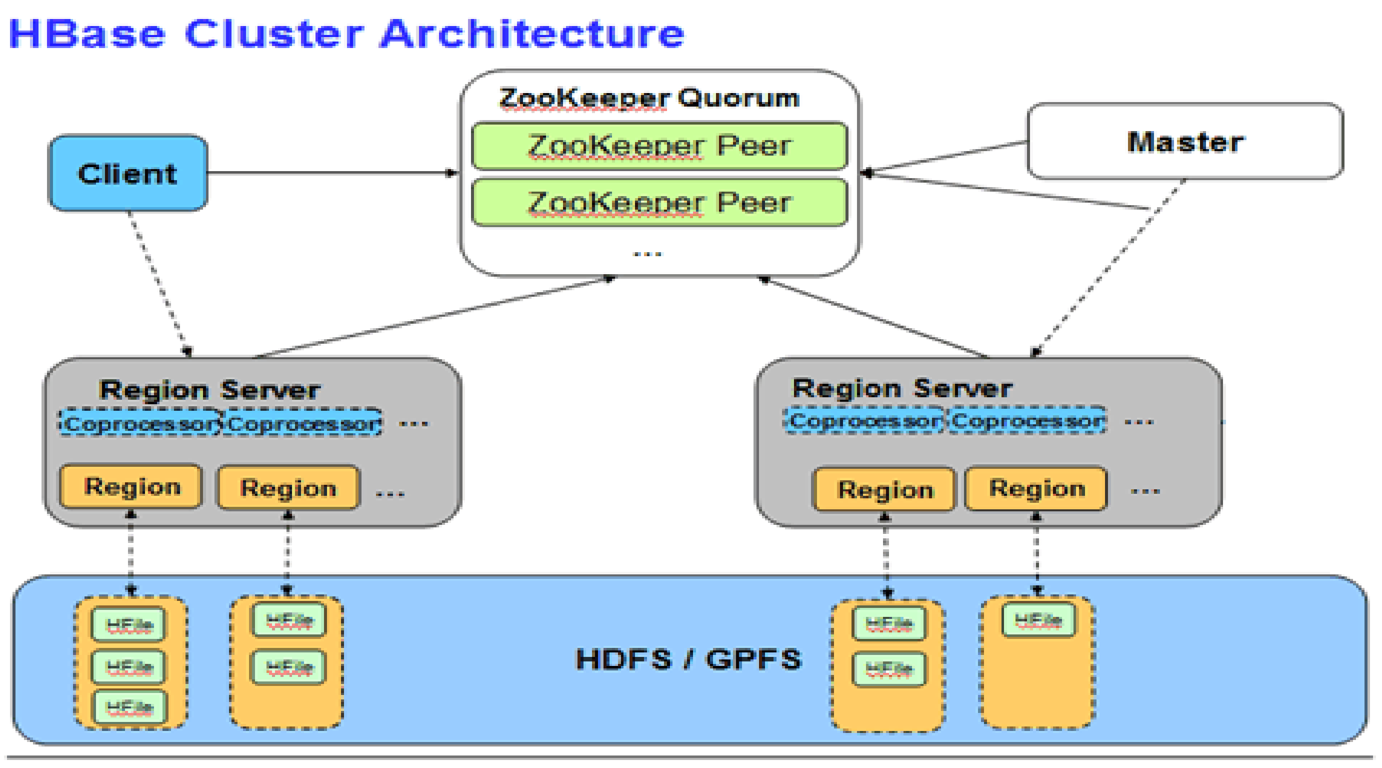
5. **Why columns are not defined at the time of table creation in HBase?**

Ans. Columns are basically the values for one column families. We can compare them to rows in a RDBMS. So while creating table we don't define rows, we insert them, similarly while creating table in HBase we don't define columns, we define column families and then we add columns on the fly while the table is running and up.

6. How does data get managed in HBase?

Ans. HBase combines the scalability of Hadoop by running on the Hadoop Distributed File System (HDFS), with real-time data access as a **key/value** store and deep analytic capabilities of Map Reduce. HBase is built on top of the distributed file system (DFS), which can store large files. HBase provides fast record lookups and updates for large tables.

Below is the Architecture of HBase Cluster-



The ZooKeeper cluster acts as a coordination service for the entire HBase cluster.

HBase contains two primary services:

- **<u>Master server</u>** The master server co-ordinates the cluster and performs administrative operations, such as assigning regions and balancing the loads.

- **<u>Region server</u>** The region servers do the real work. A subset of the data of each table is handled by each region server. Clients talk to region servers to access data in HBase.

  <u>Regions</u> Region servers manage a set of regions. An HBase table is made up of a set of regions. Regions are the basic unit of work in HBase. It is what is used as a split by the map reduce framework. The region contains store objects that correspond to column families. There is one store instance for each column family. Store objects create one or more StoreFiles, which are wrappers around the actual storage file called the HFile. The region also contains a MemStore, which is in-memory storage and is used as a write cache. Rows are written to the MemStore. The data in the MemStore is ordered. If the MemStore becomes full, it is persisted to an HFile on disk.
  To improve performance, it is important to get an even distribution of data among regions, which ensures the best parallelism in map tasks.

  <u>HFiles</u>- HFiles are the physical representation of data in HBase. Clients do not read HFiles directly but go through region servers to get to the data. HBase internally puts the data in indexed StoreFiles that exist on HDFS for high-speed lookups.

  An HBase table contains *column families*, which are the logical and physical grouping of columns. There are *column qualifiers* inside of a column family, which are the columns. Column families contain columns with time stamped versions. Columns only exist when they are inserted, which makes HBase a sparse database. All column members of the same column family have the same column family prefix. Each column value is identified by a key. The *row key* is the implicit primary key. Rows are sorted by the *row key*.

7. What happens internally when new data gets inserted into HBase table?

Ans. One way of inserting data into HBase table is using PUT operation. A PUT operation writes data to HBase. When you put data into HBase, a timestamp is required. The timestamp can be generated automatically by the RegionServer or can be supplied by us. The timestamp must be unique per version of a given cell, because the timestamp identifies the version. To modify a previous version of a cell, for instance, we would issue a Put with a different value for the data itself, but the same timestamp.

The idea behind having a timestamp is to maintain the versioning of the table. Since we know that the data is stored in HDFS and it is a Write Once and Read Many times facility, so in order to keep functionality of a database we maintain the versioning.

Now there is one more operation which happens internally when we issue PUT command-

WAL(Write Ahead Log)

Each of these modifications is wrapped into a Key-Value object instance and sent over the wire using RPC calls. The calls are (ideally batched) to the HRegion Server that serves the affected regions. Once it arrives the payload, the said Key-Value, is routed to the HRegion that is responsible for the affected row. The data is written to the WAL and then put into the MemStore of the actual Store that holds the record. And that also pretty much describes the write-path of HBase.

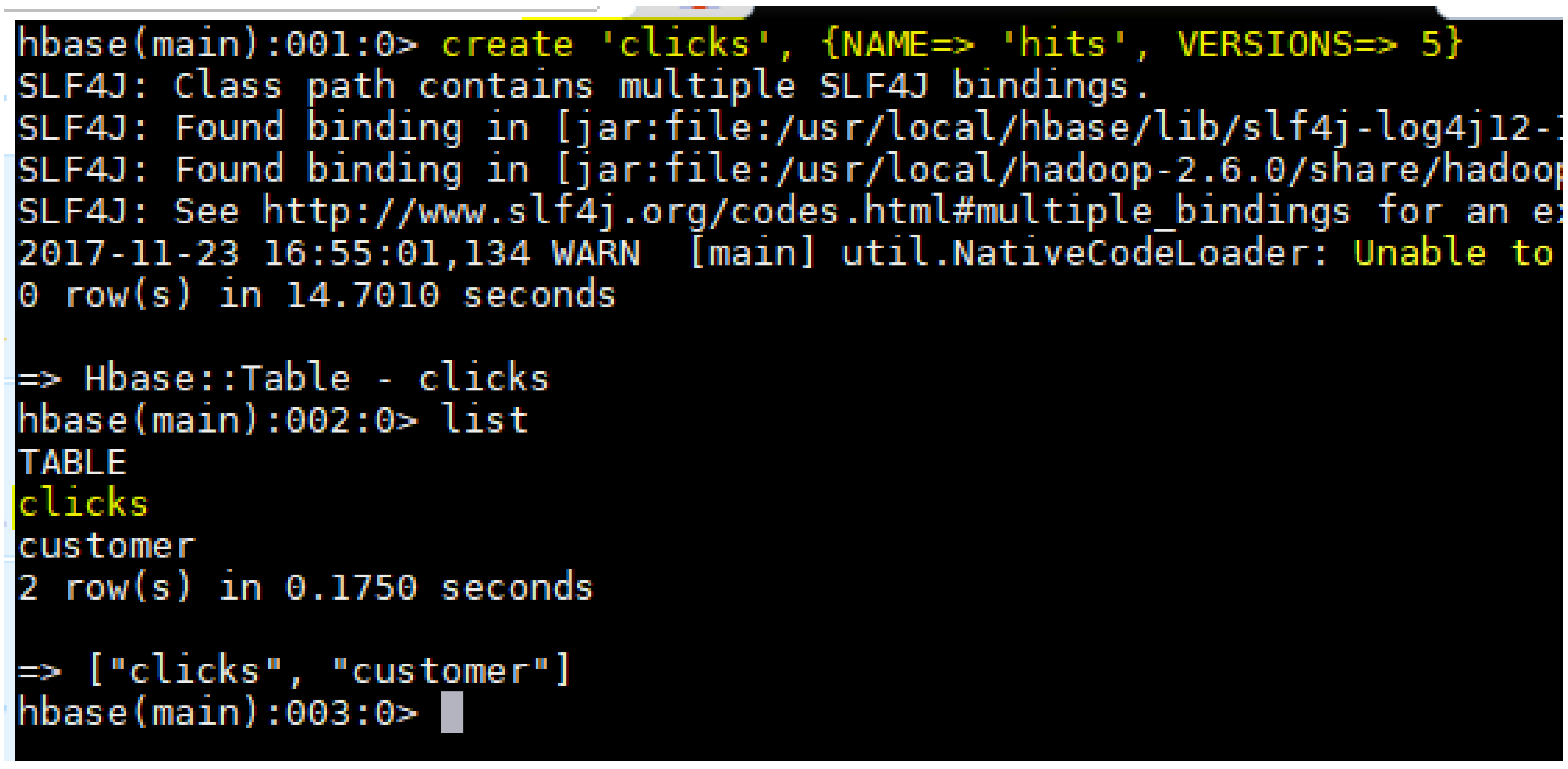
Problem 1:

Create an HBase table named 'clicks' with a column family 'hits' such that it should be able to store last 5 values of qualifiers inside 'hits' column family.

Solution-

We will create a HBase table first with name “clicks” and column family “hits” and will specify the maximum version of its columns to 5 using below command-

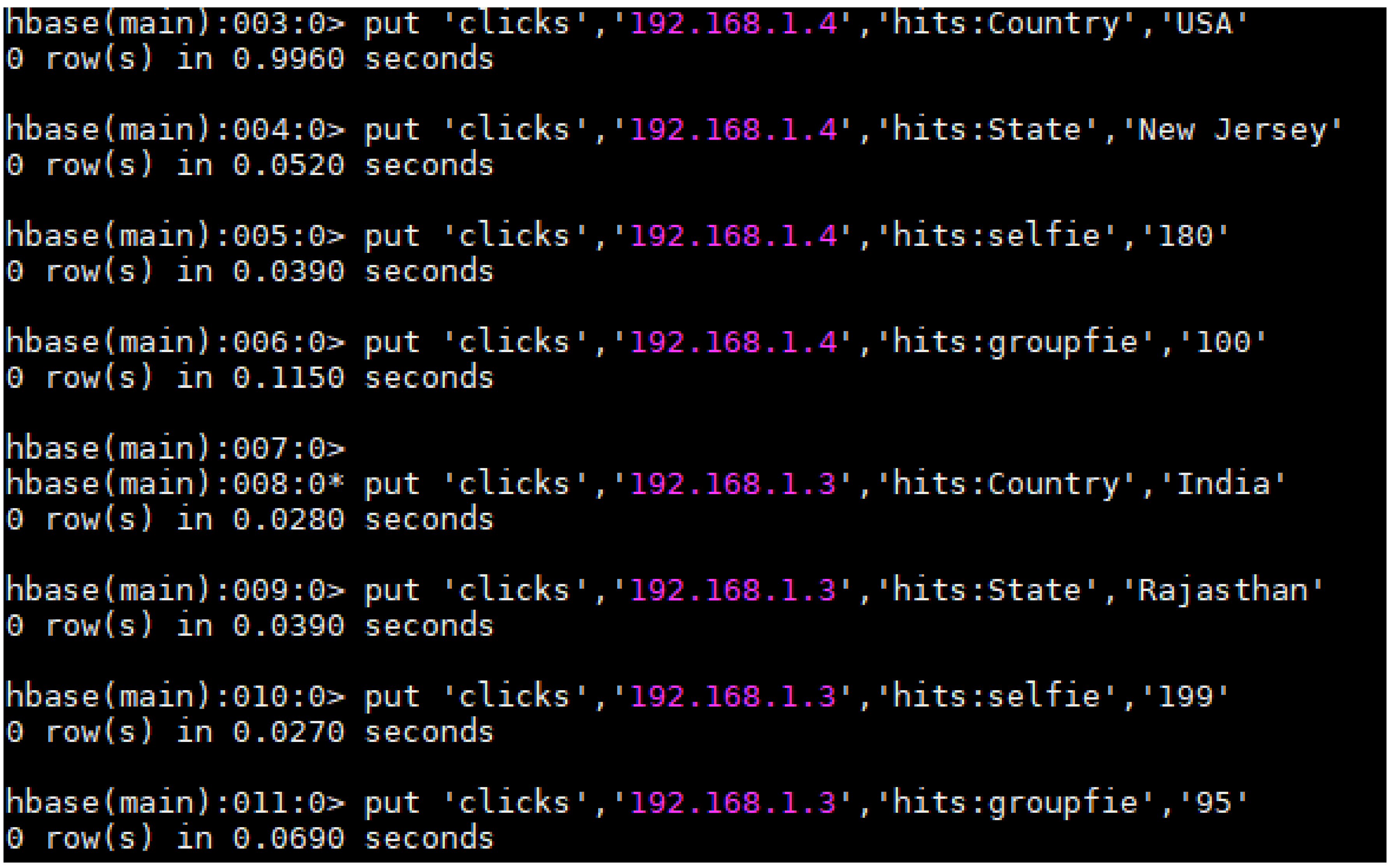
- Create ‘clicks’, {NAME=> ‘hits’, VERSIONS=>5}



```
hbase(main):001:0> create 'clicks', {NAME=> 'hits', VERSIONS=> 5}
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/hbase/lib/slf4j-log4j12-
SLF4J: Found binding in [jar:file:/usr/local/hadoop-2.6.0/share/hadoo
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an e
2017-11-23 16:55:01,134 WARN  [main] util.NativeCodeLoader: Unable to
0 row(s) in 14.7010 seconds

=> Hbase::Table - clicks
hbase(main):002:0> list
TABLE
clicks
customer
2 row(s) in 0.1750 seconds

=> ["clicks", "customer"]
hbase(main):003:0>
```

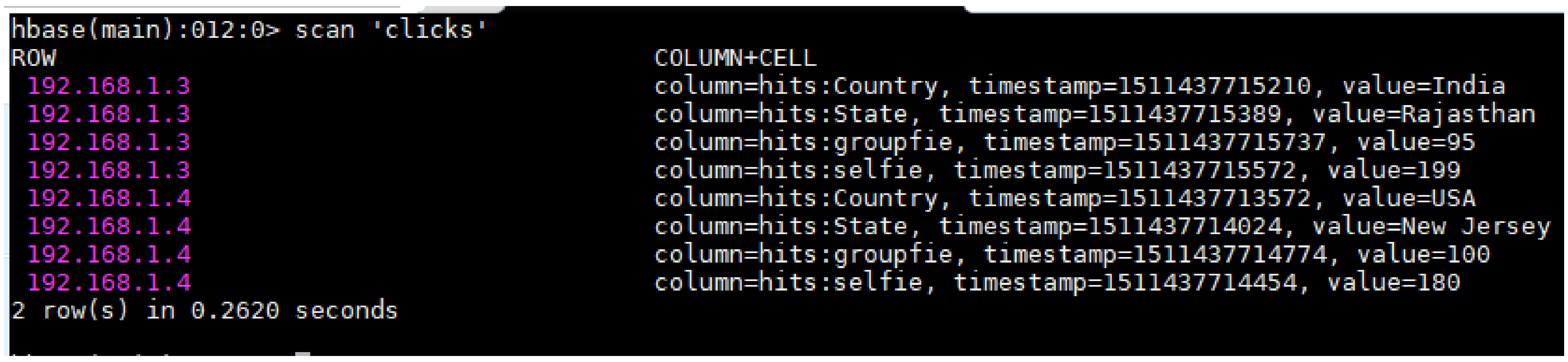
Problem 2:
Add few records in the table and update some of them. Use IP Address as row-key. Scan the table to view if all the previous versions are getting displayed.

Now we will insert data to the table using put with IP addresses as ROW KEYS.
Here we have used 2 IP addresses- **192.168.1.4** and **192.168.1.3** as ROW KEYS for column family **'hits'**.
We are inserting columns Country, State, selfie and groupfie for this column family and also their corresponding value-



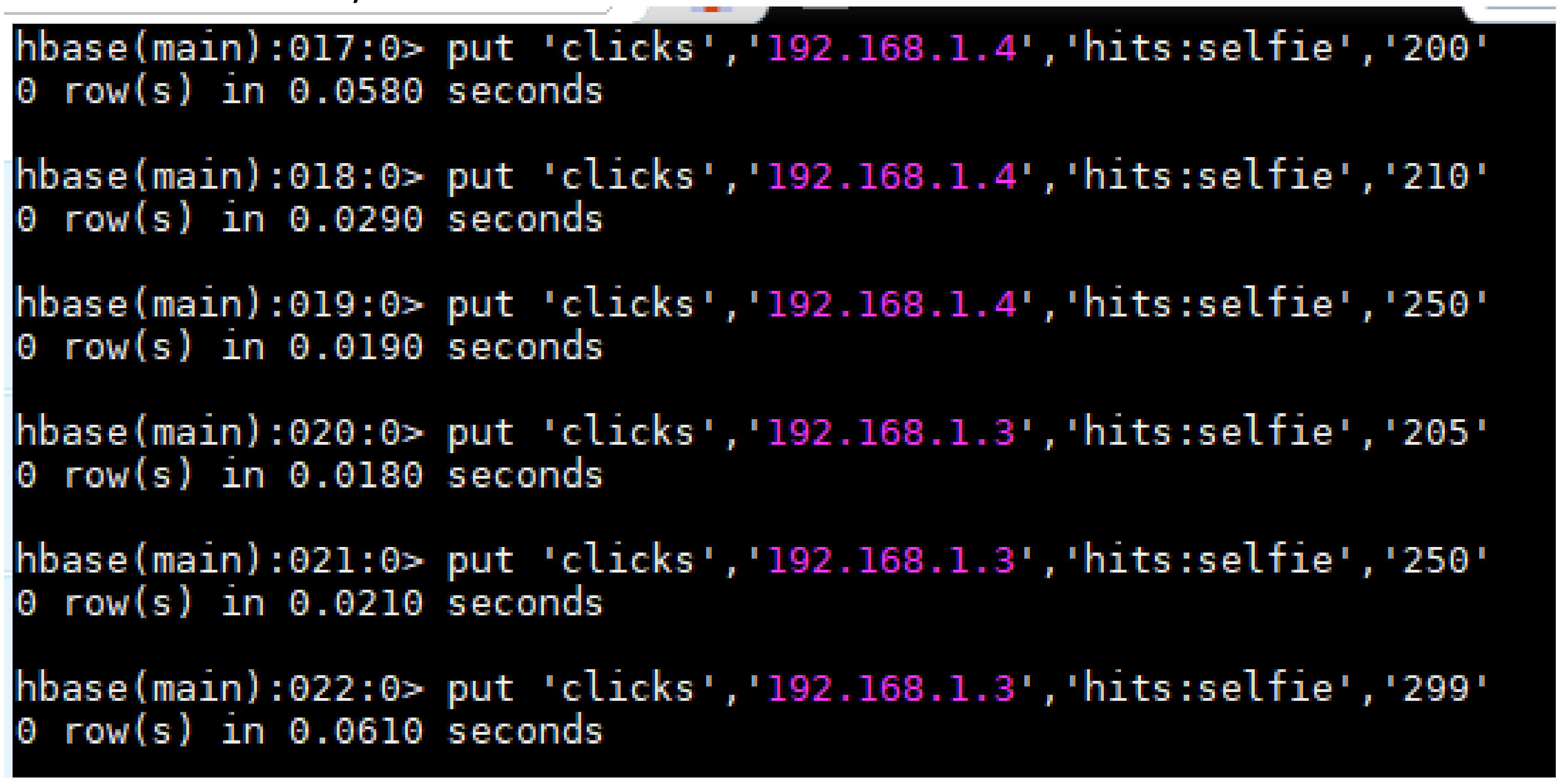
```
hbase(main):003:0> put 'clicks','192.168.1.4','hits:Country','USA'
0 row(s) in 0.9960 seconds

hbase(main):004:0> put 'clicks','192.168.1.4','hits:State','New Jersey'
0 row(s) in 0.0520 seconds

hbase(main):005:0> put 'clicks','192.168.1.4','hits:selfie','180'
0 row(s) in 0.0390 seconds

hbase(main):006:0> put 'clicks','192.168.1.4','hits:groupfie','100'
0 row(s) in 0.1150 seconds

hbase(main):007:0>
hbase(main):008:0* put 'clicks','192.168.1.3','hits:Country','India'
0 row(s) in 0.0280 seconds

hbase(main):009:0> put 'clicks','192.168.1.3','hits:State','Rajasthan'
0 row(s) in 0.0390 seconds

hbase(main):010:0> put 'clicks','192.168.1.3','hits:selfie','199'
0 row(s) in 0.0270 seconds

hbase(main):011:0> put 'clicks','192.168.1.3','hits:groupfie','95'
0 row(s) in 0.0690 seconds
```

Below is the result after inserting data-



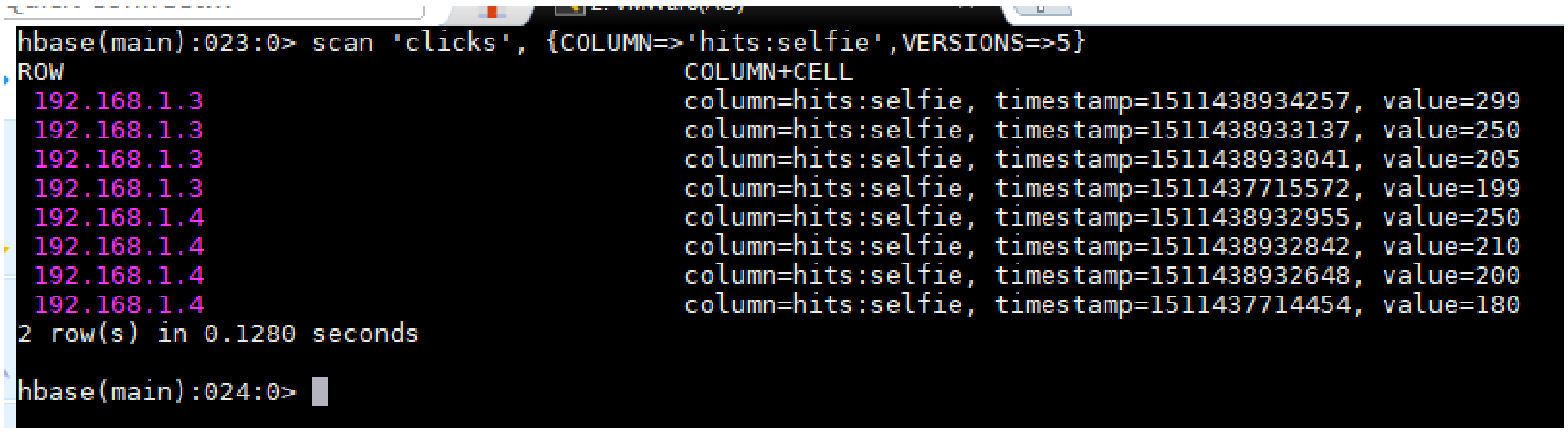
```
hbase(main):012:0> scan 'clicks'
ROW                                        COLUMN+CELL
 192.168.1.3                               column=hits:Country, timestamp=1511437715210, value=India
 192.168.1.3                               column=hits:State, timestamp=1511437715389, value=Rajasthan
 192.168.1.3                               column=hits:groupfie, timestamp=1511437715737, value=95
 192.168.1.3                               column=hits:selfie, timestamp=1511437715572, value=199
 192.168.1.4                               column=hits:Country, timestamp=1511437713572, value=USA
 192.168.1.4                               column=hits:State, timestamp=1511437714024, value=New Jersey
 192.168.1.4                               column=hits:groupfie, timestamp=1511437714774, value=100
 192.168.1.4                               column=hits:selfie, timestamp=1511437714454, value=180
2 row(s) in 0.2620 seconds
```

Now we will upsert 3 more data in the table with same row keys and will try to update column for selfie for both row keys-



```
hbase(main):017:0> put 'clicks','192.168.1.4','hits:selfie','200'
0 row(s) in 0.0580 seconds

hbase(main):018:0> put 'clicks','192.168.1.4','hits:selfie','210'
0 row(s) in 0.0290 seconds

hbase(main):019:0> put 'clicks','192.168.1.4','hits:selfie','250'
0 row(s) in 0.0190 seconds

hbase(main):020:0> put 'clicks','192.168.1.3','hits:selfie','205'
0 row(s) in 0.0180 seconds

hbase(main):021:0> put 'clicks','192.168.1.3','hits:selfie','250'
0 row(s) in 0.0210 seconds

hbase(main):022:0> put 'clicks','192.168.1.3','hits:selfie','299'
0 row(s) in 0.0610 seconds
```

In below screenshot using **scan** command we can see all the previous versions of column “selfie” which we upserted. Currently it has 4 versions per ROW KEY

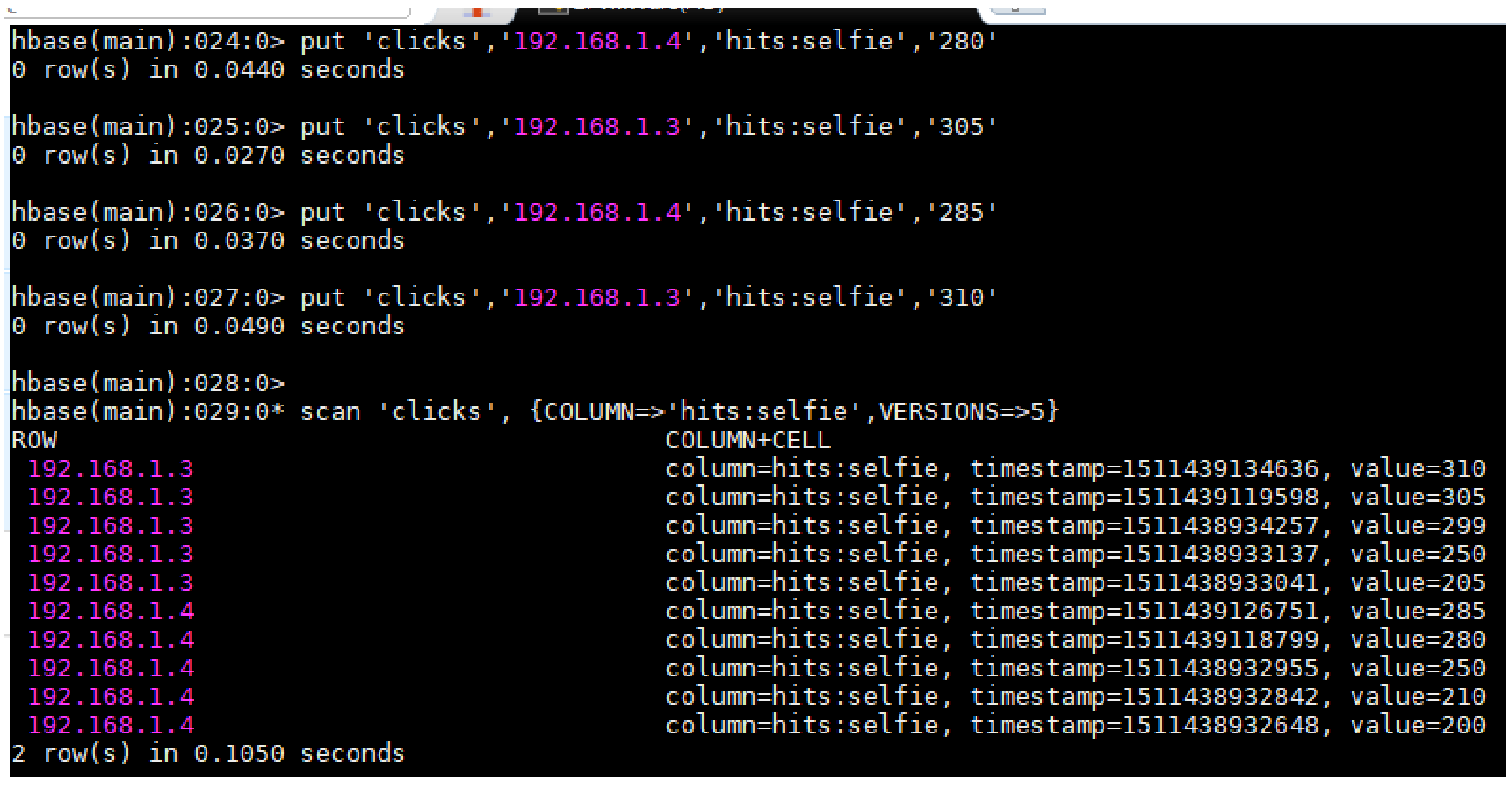


```
hbase(main):023:0> scan 'clicks', {COLUMN=>'hits:selfie',VERSIONS=>5}
ROW                                        COLUMN+CELL
 192.168.1.3                               column=hits:selfie, timestamp=1511438934257, value=299
 192.168.1.3                               column=hits:selfie, timestamp=1511438933137, value=250
 192.168.1.3                               column=hits:selfie, timestamp=1511438933041, value=205
 192.168.1.3                               column=hits:selfie, timestamp=1511437715572, value=199
 192.168.1.4                               column=hits:selfie, timestamp=1511438932955, value=250
 192.168.1.4                               column=hits:selfie, timestamp=1511438932842, value=210
 192.168.1.4                               column=hits:selfie, timestamp=1511438932648, value=200
 192.168.1.4                               column=hits:selfie, timestamp=1511437714454, value=180
2 row(s) in 0.1280 seconds

hbase(main):024:0>
```

Now again we are upserting some data for column “selfie” and thus making a total of 6 versions for column selfie.

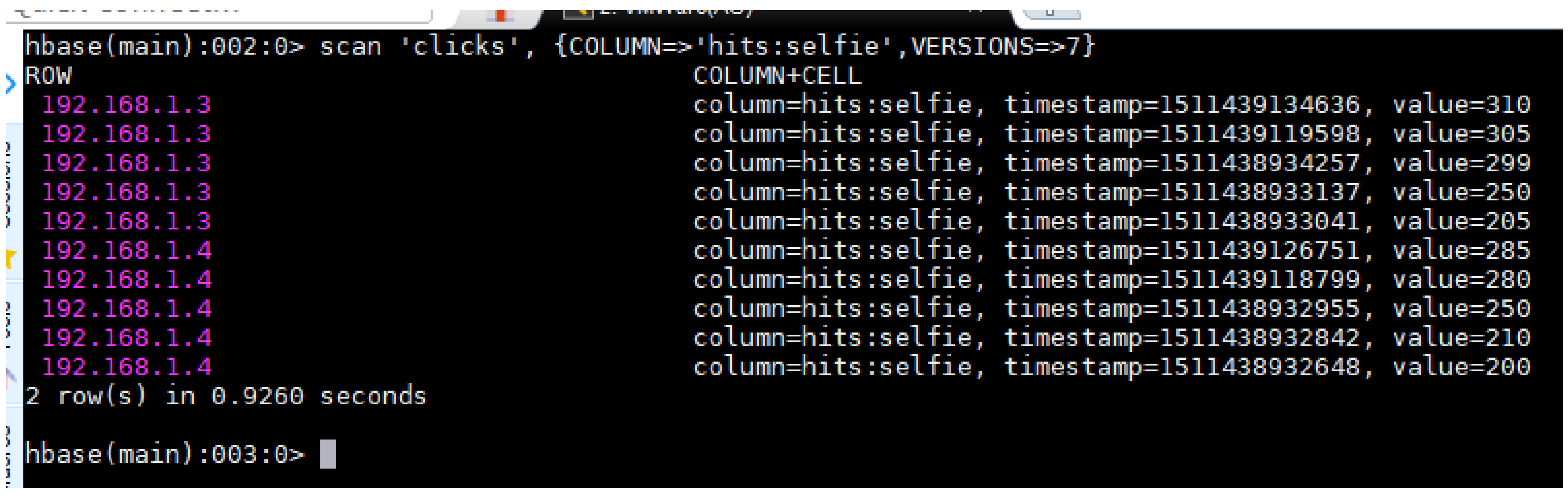
Then we are doing a scan on the table for column “selfie” and we can see that there are only 5 version available for column selfie.

The earliest one version is not being displayed.



```
hbase(main):024:0> put 'clicks','192.168.1.4','hits:selfie','280'
0 row(s) in 0.0440 seconds

hbase(main):025:0> put 'clicks','192.168.1.3','hits:selfie','305'
0 row(s) in 0.0270 seconds

hbase(main):026:0> put 'clicks','192.168.1.4','hits:selfie','285'
0 row(s) in 0.0370 seconds

hbase(main):027:0> put 'clicks','192.168.1.3','hits:selfie','310'
0 row(s) in 0.0490 seconds

hbase(main):028:0>
hbase(main):029:0* scan 'clicks', {COLUMN=>'hits:selfie',VERSIONS=>5}
ROW                                      COLUMN+CELL
 192.168.1.3                             column=hits:selfie, timestamp=1511439134636, value=310
 192.168.1.3                             column=hits:selfie, timestamp=1511439119598, value=305
 192.168.1.3                             column=hits:selfie, timestamp=1511438934257, value=299
 192.168.1.3                             column=hits:selfie, timestamp=1511438933137, value=250
 192.168.1.3                             column=hits:selfie, timestamp=1511438933041, value=205
 192.168.1.4                             column=hits:selfie, timestamp=1511439126751, value=285
 192.168.1.4                             column=hits:selfie, timestamp=1511439118799, value=280
 192.168.1.4                             column=hits:selfie, timestamp=1511438932955, value=250
 192.168.1.4                             column=hits:selfie, timestamp=1511438932842, value=210
 192.168.1.4                             column=hits:selfie, timestamp=1511438932648, value=200
2 row(s) in 0.1050 seconds
```

Even if we try scanning table with VERSIONS=>7 we will get only 5 VERSIONS as table has been restricted to store only 5 versions



```
hbase(main):002:0> scan 'clicks', {COLUMN=>'hits:selfie',VERSIONS=>7}
ROW                                      COLUMN+CELL
 192.168.1.3                             column=hits:selfie, timestamp=1511439134636, value=310
 192.168.1.3                             column=hits:selfie, timestamp=1511439119598, value=305
 192.168.1.3                             column=hits:selfie, timestamp=1511438934257, value=299
 192.168.1.3                             column=hits:selfie, timestamp=1511438933137, value=250
 192.168.1.3                             column=hits:selfie, timestamp=1511438933041, value=205
 192.168.1.4                             column=hits:selfie, timestamp=1511439126751, value=285
 192.168.1.4                             column=hits:selfie, timestamp=1511439118799, value=280
 192.168.1.4                             column=hits:selfie, timestamp=1511438932955, value=250
 192.168.1.4                             column=hits:selfie, timestamp=1511438932842, value=210
 192.168.1.4                             column=hits:selfie, timestamp=1511438932648, value=200
2 row(s) in 0.9260 seconds

hbase(main):003:0>
```