CSC561 NoSQL Databases
Lab 4: Column Family Database / Hbase Part 1

(If needed you can exit the shell by entering exit, then stop hbase by entering stop-hbase.sh)

Column Family Databases:

Column family databases are best known because of Google's BigTable implementation. They are very similar on the surface to relational database, but they have critical conceptual differences. You will not be able to apply the same sort of solutions that you used in a relational database to a column database.

That is because column databases are not relational; they do not have what a RDBMS would recognize as a table.

In column-family databases, each row consist of a collection of columns/value pairs. A collection of similar rows then makes up a column family. (In a relational databases, this would be equivalent to a collection of rows making up a table.) The main difference is that in a column-family database, rows do not have to contain the same columns.

RDBMS impose high cost of schema change for low cost of query change. Column families impose little to no cost in schema change, for slightly more cost in query change.

Why HBase:

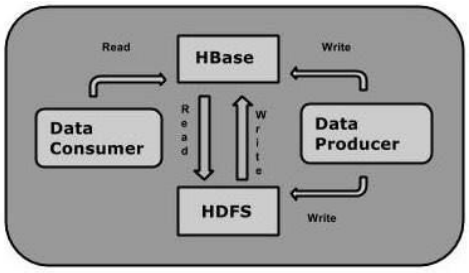
HBase is a column-oriented database that prides itself on consistency and scaling out. It is based on BigTable, a high-performance, proprietary database developed by Google and described in the 2006 white paper "Bigtable: A Distributed Storage System for Structured Data."[1] Initially created for natural language processing, HBase started life as a contrib package for Apache Hadoop. Since then, it has become a top-level Apache project.

On the architecture front, HBase is designed to be fault tolerant. Hardware failures may be uncommon for individual machines, but in a large cluster, node failure is the norm. By using write-ahead logging and distributed configuration, HBase can quickly recover from individual server failures.

Additionally, HBase lives in the Hadoop ecosystem that has its own complementary benefits. HBase is built on Hadoop which provides random real-time read/write access to data in the Hadoop File System. It is a sturdy, scalable computing platform that provides a distributed file system and mapreduce capabilities.

You can store the data in HDFS either directly or through HBase. Data consumers read/access the data in HDFS randomly using HBase; HBase sits on top of the Hadoop File System.

[1] http://static.usenix.org/events/osdi06/tech/chang/chang_html/?em_x=22



It is actively used and developed by a number of high-profile companies for their "Big Data" problems. Notably, Facebook chose HBase as a principal component of its messaging infrastructure in November 2010. Twitter uses HBase extensively, ranging from data generation (for applications such as people search) to storing monitoring/performance data. Other companies using HBase also include eBay, Adobe, Mahalo, and Yahoo!.

With all of this activity, new versions of HBase are constantly being released.

Starting HBase:

A fully operational, production-quality HBase cluster should really consist of no fewer than five nodes (to allow for fault-tolerance, replication, and backup). Such a setup would be overkill for our needs. Fortunately, HBase supports three running modes:

- Stand-alone mode is a single machine acting alone. (uses local file system)
- Pseudo-distributed mode is a single node pretending to be a cluster. (uses HDFS)
- Fully distributed mode is a cluster of nodes working together.

For most of this lab, we'll be running HBase in stand-alone mode.

To get the local instance of HBase running, open a terminal (command prompt), navigate to our HBase directory, and run the start command:

cd /usr/lib/hbase/hbase-1.1.3/bin
start-hbase.sh
(To shut down HBase, use the stop-hbase.sh command in the same directory.)

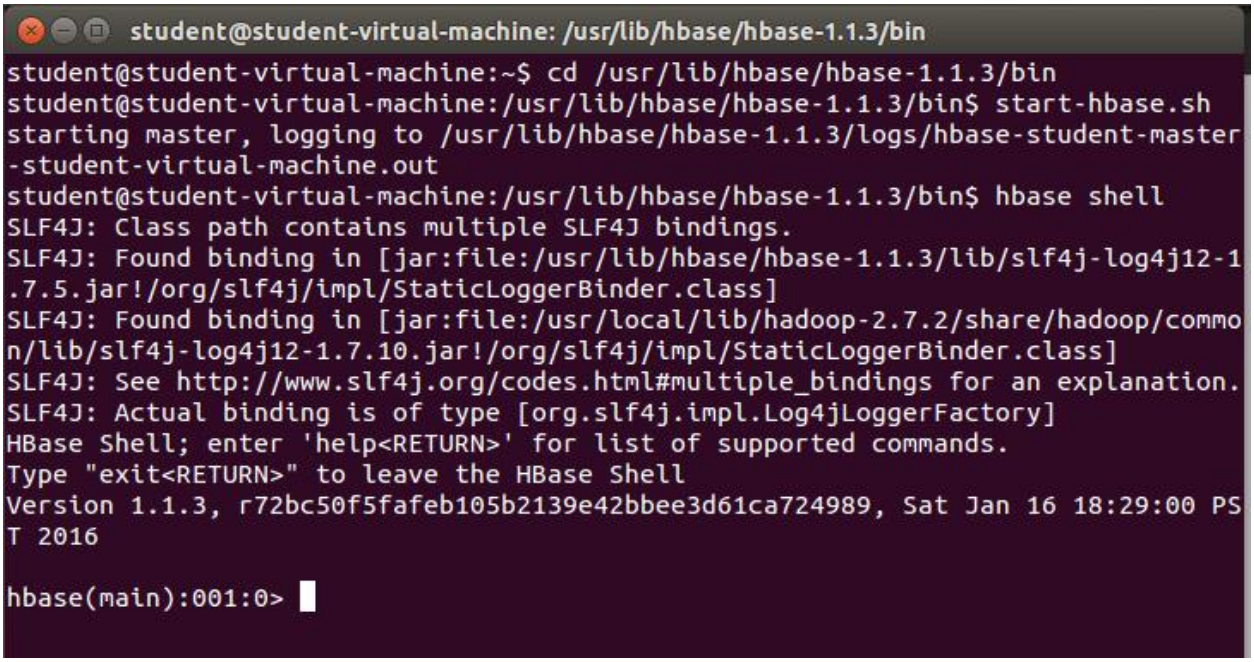
```
student@student-virtual-machine:~$ cd /usr/lib/hbase/hbase-1.1.3/bin
student@student-virtual-machine:/usr/lib/hbase/hbase-1.1.3/bin$ start-hbase.sh
starting master, logging to /usr/lib/hbase/hbase-1.1.3/logs/hbase-student-master
-student-virtual-machine.out
student@student-virtual-machine:/usr/lib/hbase/hbase-1.1.3/bin$
```

The Hbase Shell

The HBase shell is a JRuby-based command-line program you can use to interact with HBase. In the shell, you can add and remove tables, alter table schema, add or delete data, and do a bunch of other tasks. Later we'll explore other means of connecting to HBase, but for now the shell will be our home.

So now that Hbase is running, let's start the HBase shell by entering:

hbase shell



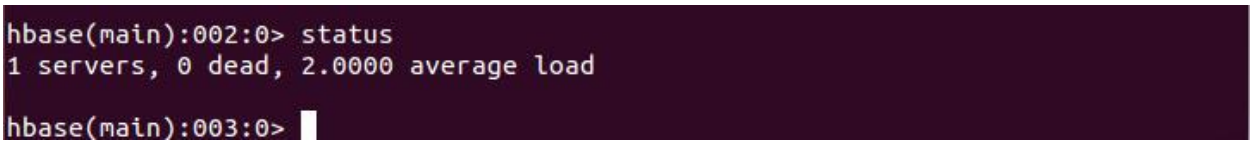
To confirm that it's working properly, try asking it for version information at the shell prompt (hbase(main):001:0>).

version

You can enter help at any time to see a list of available commands or to get usage information about a particular command.

Next, execute the status command to see how your HBase server is holding up:

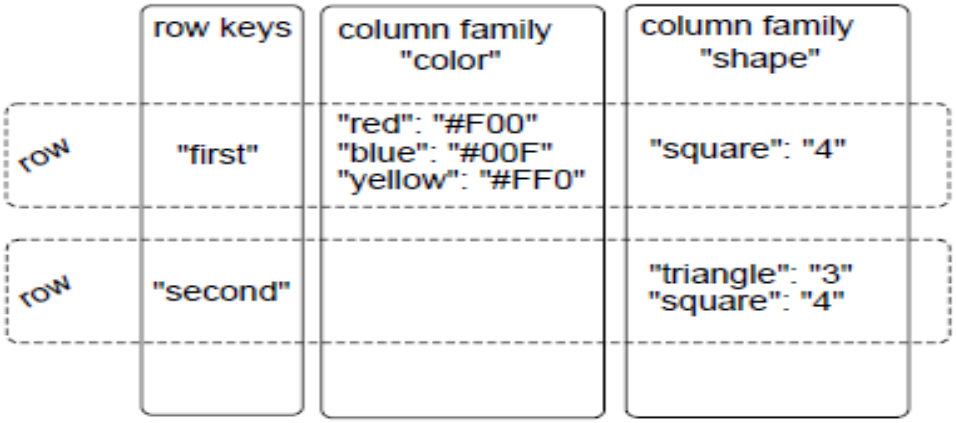
Status



HBase CRUD

A map is a key-value pair, like a hash in Ruby or a hashmap in Java. A table in HBase is basically a big map (more accurately, it's a map of maps). In an HBase table, keys are arbitrary strings that each map to a row of data. A row is itself a map, in which keys are called columns and values are un-interpreted arrays of bytes. Columns are grouped into column families, so a column's full name consists of two

parts: the column family name and the column qualifier. Often these are concatenated together using a colon (for example, 'family:qualifier'). HBase tables consist of rows ,keys, column families, columns, and values:



In the figure above, we have a hypothetical table with two column families: “color” and “shape”. The table has two rows—denoted by dashed boxes—identified by their row keys: “first” and “second”. Looking at just the first row, we see that it has three columns in the color column family (with qualifiers red, blue, and yellow) and one column in the shape column family (square). The combination of row key and column name (including both family and qualifier) creates an address for locating data. In this example, the tuple first/color:red points us to the value '#F00'.

Table Creation

To illustrate this table structure, let’s create a wiki. A wiki contains pages, each of which has a unique title string and contains some article text. Our wiki will be called “wiki” and our column family will be called “text”

Use the *create* command to make our wiki table:

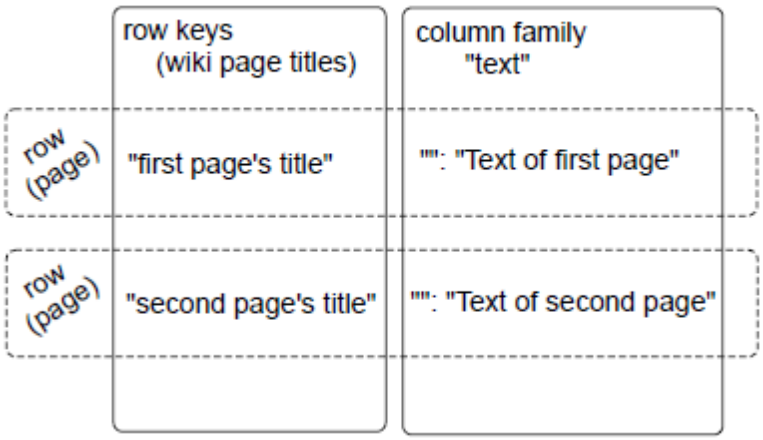
create ‘wiki’, ‘text’

```
hbase(main):003:0> create 'wiki', 'text'
0 row(s) in 1.7320 seconds

=> Hbase::Table - wiki
hbase(main):004:0>
```

The table is currently empty; it has no rows and thus no columns. Unlike a relational database, in HBase a column is specific to the row that contains it. It is only when adding rows that columns are created to store data.

Visualizing our table architecture, we have:



The wiki table has one column family. We will create each row to have exactly one column within the "text" family, qualified by the empty string (''). So, the full column name containing the text of a page will be 'text:'. (Notice we do not have a qualifier after the family name, it is just 'text':)

Inserting and Reading Data: put and get commands

First we'll create the entry for our wiki homepage. To add data to an Hbase table, use the *put* command:

put 'wiki', 'Home', 'text:', 'Welcome to My HBase Wiki!'

```
=> Hbase::Table - wiki
hbase(main):004:0> put 'wiki', 'Home', 'text', 'Welcome to My HBase Wiki!'
0 row(s) in 0.6580 seconds

hbase(main):005:0>
```

This command inserts a new row into the wiki table with the key 'Home', adding 'Welcome to My Hbase Wiki!' to the column called 'text:'.

We can query the data for the 'Home' row using *get*, which requires two parameters: the table name and the row key. (You can optionally specify a list of columns to return.)

get 'wiki', 'Home', 'text:'

```
hbase(main):005:0> get 'wiki', 'Home', 'text:'
COLUMN                CELL
 text:                timestamp=1459097656140, value=Welcome to My HBase Wiki!
1 row(s) in 0.3020 seconds

hbase(main):006:0>
```

Notice the timestamp field in the output. HBase stores an integer timestamp for all data values, representing time in milliseconds since the epoch (00:00:00UTC on January 1, 1970). When a new value is written to the same cell, the old value will be retained, indexed by its timestamp. This is a ***great***

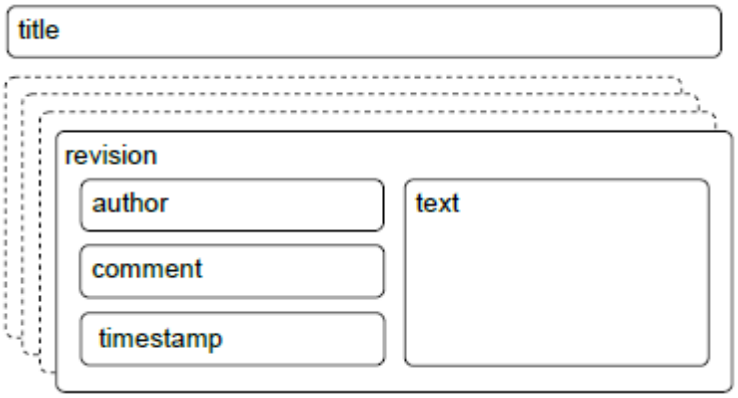
feature. Most databases require you to specifically handle historical data explicitly, but in HBase, versioning is automatic. If you do not want the timestamp to be in MSE, the put and get commands allow you to specify a timestamp explicitly by inserting an integer value of your choice. This gives you an extra dimension to work with if you need it. If you don't explicitly specify a timestamp, HBase will use the current time when inserting, and it will return the most recent version when reading.

Altering Tables

Currently our wiki schema has pages with titles, text, and an integrated version history but nothing else. Let's expand our requirements to include the following:

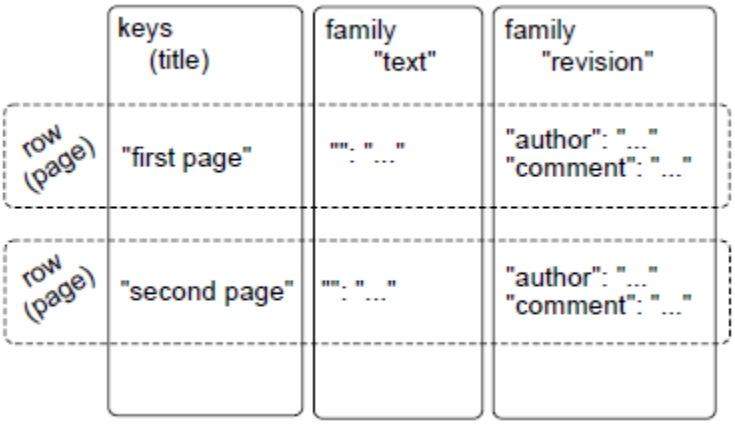
- In our wiki, a page is uniquely identified by its title.
- A page can have unlimited revisions.
- A revision is identified by its timestamp.
- A revision contains text and optionally a commit comment.
- A revision was made by an author, identified by name.

Visually, what we want to create is the following structure:



In this abstract representation of requirements for a page, we see that each revision has an author, a commit comment, some article text, and a timestamp. The title of a page is not part of a revision, because it's the identifier we use to denote revisions belonging to the same page.

Mapping this abstraction to an HBase table creates a different form:



Our wiki table uses the title as the row key and will group other page data into two column families called “text” and “revision”. The text column family is the same as before; we expect each row to have exactly one column, qualified by the empty string (''), to hold the article contents. The “revision” column family is to hold other revision-specific data, such as the author and commit comment.

A Note on Defaults

We created the wiki table with no special options, so all the HBase default values were used. One such default value is to keep only three VERSIONS of column values. For our requirements to be met, we will have to increase that.

To make any schema changes to an Hbase table, we first have to take the table offline with the *disable* command:

disable 'wiki'

```
hbase(main):006:0> disable 'wiki'
0 row(s) in 2.4470 seconds

hbase(main):007:0>
```

Now we can modify column family characteristics using the *alter* command:

alter 'wiki', { NAME => 'text', VERSIONS => org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }

```
hbase(main):011:0> alter 'wiki', {NAME => 'text', VERSIONS => org.apache.hadoop.
hbase.HConstants::ALL_VERSIONS}
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 2.3260 seconds

hbase(main):012:0>
```

We just instructed HBase to alter the text column family’s VERSIONS attribute. (There are a number of other attributes that can be set that we will explore later in the lab).

Altering a Table

Operations that alter column family characteristics can be very expensive because HBase has to create a new column family with the chosen specifications and then copy all the data over. In a production system, this may incur significant downtime. For this reason, it is important to determine the necessary column family options in the design phase.

With the wiki table still disabled, we can now add the revision column family using the *alter* command:

alter 'wiki', { NAME => 'revision', VERSIONS => org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }

```
hbase(main):012:0> alter 'wiki', {NAME => 'revision', VERSIONS => org.apache.had
oop.hbase.HConstants::ALL_VERSIONS}
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 2.0640 seconds

hbase(main):013:0>
```

Once our changes are made, we can re-enable our wiki with the *enable* command:

```
hbase(main):013:0> enable 'wiki'
0 row(s) in 1.4810 seconds
```

As with the 'text' column family, this alteration just creates a 'revision' column family in the table schema; it does not add individual columns. Our model expects each row to eventually contain a revision:author and revision:comment, but it will be up to the client to honor this expectation; it's not written into any formal schema. If someone wants to add a revision:foo for a page, HBase won't stop them.

Adding Data Programmatically

Tthe HBase shell is great for tasks such as manipulating tables. Sadly, the shell's data insertion support isn't the best. The *put* command only allows setting one column value at a time, and in our newly updated schema, we need to add multiple column values simultaneously so they all share the same timestamp.

We're going to need to start scripting. The following script can be executed directly in the HBase shell, since the shell is also a JRuby interpreter. When run, it adds a new version of the text for the Home page, setting the author and comment fields at the same time.

JRuby runs on the Java virtual machine (JVM), giving it access to the Hbase Java code. The script below is available for you to cut/paste into your Hbase shell:

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end
```

```
table = HTable.new( @hbase.configuration, "wiki" )
p = Put.new( *jbytes( "Home" ) )
p.add( *jbytes( "text", "", "Hello world" ) )
p.add( *jbytes( "revision", "author", "jimbo" ) )
p.add( *jbytes( "revision", "comment", "my first edit" ) )
table.put( p )
```

#What this code does
#
#The import lines bring references to useful HBase classes into the shell.
#
#The jbytes()function takes any number of arguments and returns an array converted to
#Java byte arrays, as the HBase API methods demand.
#
#create a local variable (table) pointing to our wiki table, usingthe @hbase administration object #for configuration information.
#
#stage a commit operation by creating and preparing a new instance of Put, which takes the row to be #modified. In this instance, we will use the Home page we've been working with thus far.
#
#p.add() properties to our Put instance and then call on the table object to execute the put #operation we've prepared. The add() method has several forms; in our case, we used the
#three-argument version: add(column_family, column_qualifier, value).

When you paste this or type it into shell you should see:

```
hbase(main):014:0> import 'org.apache.hadoop.hbase.client.HTable'
=> Java::OrgApacheHadoopHbaseClient::HTable
hbase(main):015:0> import 'org.apache.hadoop.hbase.client.Put'
=> Java::OrgApacheHadoopHbaseClient::Put
hbase(main):016:0> def jbytes( *args )
hbase(main):017:1> args.map { |arg| arg.to_s.to_java_bytes }
hbase(main):018:1> end
hbase(main):019:0> table = HTable.new( @hbase.configuration, "wiki" )
=> #<Java::OrgApacheHadoopHbaseClient::HTable:0xa041404>
hbase(main):020:0> p = Put.new( *jbytes( "Home" ) )
=> #<Java::OrgApacheHadoopHbaseClient::Put:0x3317bba6>
hbase(main):021:0> p.add( *jbytes( "text", "", "Hello world" ) )
=> #<Java::OrgApacheHadoopHbaseClient::Put:0x3317bba6>
hbase(main):022:0> p.add( *jbytes( "revision", "author", "jimbo" ) )
=> #<Java::OrgApacheHadoopHbaseClient::Put:0x3317bba6>
hbase(main):023:0> p.add( *jbytes( "revision", "comment", "my first edit" ) )
=> #<Java::OrgApacheHadoopHbaseClient::Put:0x3317bba6>
hbase(main):024:0> table.put( p )
```

Make certain you hit enter after the final line.

You can verify it worked by running a *get* command:

get ‘wiki’, ‘Home’

```
hbase(main):025:0> get 'wiki', 'Home'
COLUMN                CELL
 revision:author      timestamp=1459101095050, value=jimbo
 revision:comment     timestamp=1459101095050, value=my first edit
 text:                timestamp=1459101095050, value=Hello world
3 row(s) in 0.1270 seconds

hbase(main):026:0>
```

You may be tempted to build your whole structure without column families; you could store all of a row’s data in a single column family. That solution would be simpler to implement. But there are downsides to avoiding column families, namely, missing out on the fine-grained performance tuning possible in Column Family Databases. Each column family’s performance options are configured independently. These settings affect things such as read and write speed and disk space consumption. All operations in HBase are atomic at the row level. No matter how many columns are affected, the operation will have a consistent view of the particular row being accessed or modified.

Our *put* operation script affected several columns and did not explicitly specify a timestamp, so, by default, all column values had the same timestamp (the current time in milliseconds). (As you can see in the information returned by *get*, each column value listed has the same timestamp.)

CONTINUE TO NEXT PAGE FOR WORK TO SUBMIT

WORK TO SUBMIT

Be sure to do a 'git pull' to sync your repo.

The code in the previous pages of this lab has been added for you already to the continuous integration server. This includes creating the table 'wiki' with the correct structure as described throughout the lab.

Your submission will be a script file named lab4/lab4a.rb with the Hbase commands to do the following:

1. The script we just used does not contain any variables. It specifically inserts:

    a. into table 'wiki',
    b. into the row 'Home',
    c. into the 'text' column family : no column qualifier, the value "Hello World!"
    d. into the "revision" column family : "author" column qualifier , the value "jimbo"
    e. into the "revision" column family : "comment" column qualifier, the value "my first edit"

    Create a function called put_many() that creates a Put instance, adds any number of column-value pairs to it, and commits it to a table. The function signature should look like this:

```
def put_many( table_name, row, column_values )
#
# your code here
#
end
```

    You can use this as a template – your code should look like this:

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'

def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end

def put_many( table_name, row, column_values)

***INSERT YOUR CODE******

 end

table.put( p )
end
```

In your code, the replacement of the p.add statements is the trickiest part. You want to parse each column_value from the set {} of key, value pairs the user inputs; Use the .each iterator. There will be a column family and a column qualifier for each key. The delimiter for the column family, column qualifier pair is ":" (Remember that the empty string is a valid column qualifier). This is most easily accomplished using the split() method.

You can then use them in p.add(column_family, column_qualifier, value) statement within your put_many def.

2. Paste and Call your put_many function with the following parameters (insert your own text for the parameters in blue):

```
put_many  'wiki', 'Make up your title', {
"text:" => "Make up some article text",
"revision:author" => "Your name",
"revision:comment" => "Insert some comment" }
```

Example:

```
put_many  'wiki', 'NoSQL', {
"text:" => "What JavaScript framework do you prefer?",
"revision:author" => "Tulio Llosa",
"revision:comment" => "I use Vue.js" }

get 'wiki', 'NoSQL'

COLUMN  CELL
 revision:author timestamp=1541191185157, value=Tulio Llosa
 revision:comment timestamp=1541191185157, value=I use Vue.js
 text: timestamp=1541191185157, value=What JavaScript framework do you prefer?
```

Note: Make sure to include the 'get' command so that I can see the output.