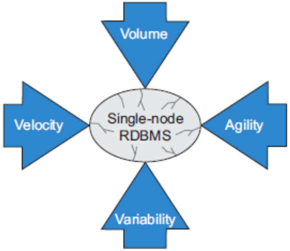
NoSQL

Grigorios Loukides
Email: grigorios.loukides@kcl.ac.uk

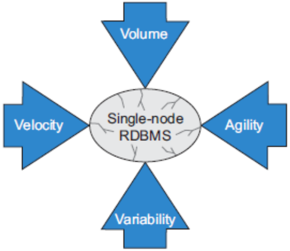
Why NoSQL?

- Traditionally SQL / Relational DBs dominate in usage!
- Since 2009, alternatives to Relational Database started appearing
- BIG Data requirements (how do you query more data that can be held in memory?)
- Relaxing Consistency to achieve High Availability / reduced latency. Important for real-time web systems (e.g. Twitter) !



What is NoSQL?

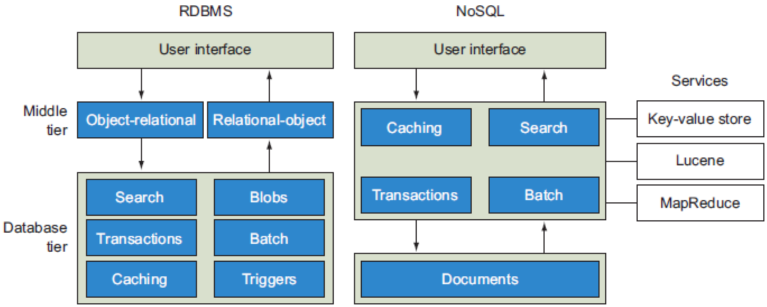
- NoSQL is a set of concepts that allows the rapid and efficient processing of data sets with a focus on performance, reliability, and agility
- NoSQL differentiates a system from a traditional relational database management system



Not only SQL

- Not only rows in tables —NoSQL systems store and retrieve data from **many formats**
- **Join-free** —NoSQL systems allow you to extract your data using simple interfaces without joins
- **Schema-free** —NoSQL systems allow you to drag-and-drop your data into a folder and then query it without creating an entity-relational model.
- **Many processors** —NoSQL systems allow you to store your database on multiple processors and maintain high-speed performance
- NoSQL systems create simple applications that **distribute the required features across the network** (APIs)

NoSQL vs. RDBMS



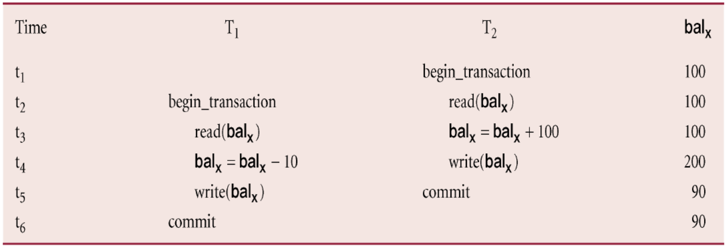
RDBMS: ACID

RDBMSs maintain transaction control by using atomic, consistent, independent, and durable (ACID) properties to insure transactions are reliable:

- *Atomicity.* All-or-nothing transaction.
- *Consistency.* All data must be valid (with respect to constraints, triggers, cascades)
- *Isolation.* Each part of a transaction occurs in isolation from other transactions
- *Durability.* Once a transaction is committed (all its aspects are completed), it is permanent.

While ACID systems focus on high data integrity, they lock resources and block reports and transactions

Transactions Example



Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

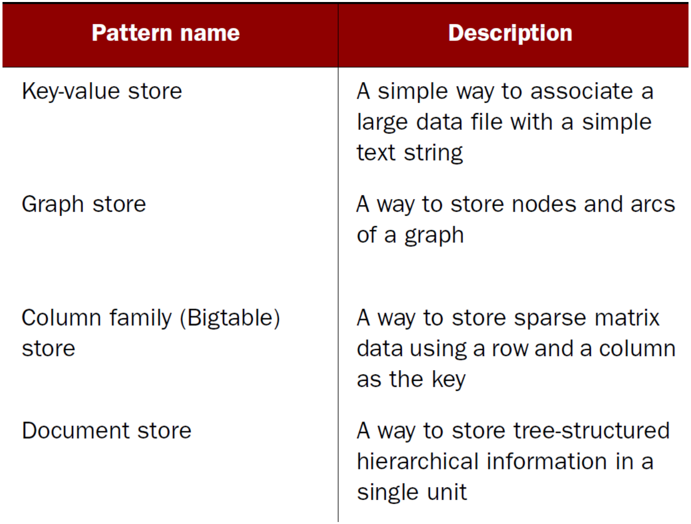
What if blocking one transaction while you wait for another to finish is an unacceptable compromise?

NoSQL: BASE

NoSQL systems are based on availability, they are based on a BASE:

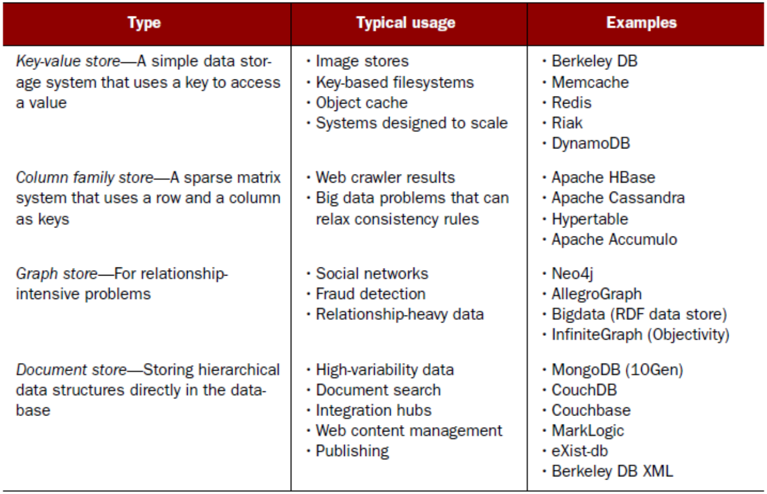
- *Basic* availability: Systems can be temporarily inconsistent, so that transactions are manageable. That is, information and service capability is “basically available”.
- *Soft-state* recognizes that some inaccuracy is temporarily allowed and data may change while being used to reduce the amount of consumed resources.
- *Eventual consistency* means eventually, when all service logic is executed, the system is left in a consistent state.

NoSQL data stores



Pattern name	Description
Key-value store	A simple way to associate a large data file with a simple text string
Graph store	A way to store nodes and arcs of a graph
Column family (Bigtable) store	A way to store sparse matrix data using a row and a column as the key
Document store	A way to store tree-structured hierarchical information in a single unit

Types of NoSQL data stores



Type	Typical usage	Examples
Key-value store—A simple data storage system that uses a key to access a value	· Image stores · Key-based filesystems · Object cache · Systems designed to scale	· Berkeley DB · Memcache · Redis · Riak · DynamoDB
Column family store—A sparse matrix system that uses a row and a column as keys	· Web crawler results · Big data problems that can relax consistency rules	· Apache HBase · Apache Cassandra · Hypertable · Apache Accumulo
Graph store—For relationship-intensive problems	· Social networks · Fraud detection · Relationship-heavy data	· Neo4j · AllegroGraph · Bigdata (RDF data store) · InfiniteGraph (Objectivity)
Document store—Storing hierarchical data structures directly in the database	· High-variability data · Document search · Integration hubs · Web content management · Publishing	· MongoDB (10Gen) · CouchDB · Couchbase · MarkLogic · eXist-db · Berkeley DB XML

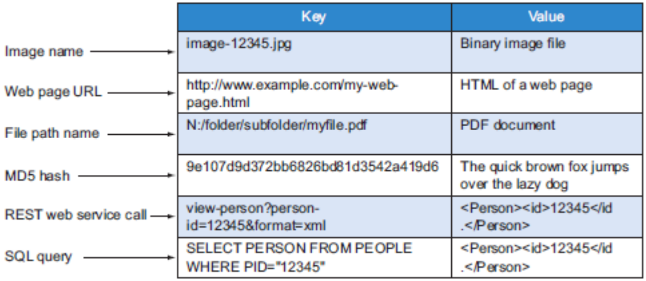
Key-value stores

Key-value stores

- Is a simple database that when presented with a simple string (the key) returns an arbitrary large BLOB of data (the value)
- Has no query language; it provides a way to add and remove key-value pairs
- Is indexed by the key; the key points directly to the value, resulting in rapid retrieval

Key-value stores

- The value has no predefined data type
  - The system will store the information as a BLOB. It's up to the application to determine what type of data is being used
- The key in a key-value store is flexible and can be represented by many formats



	Key	Value
Image name →	image-12345.jpg	Binary image file
Web page URL →	http://www.example.com/my-web-page.html	HTML of a web page
File path name →	N:/folder/subfolder/myfile.pdf	PDF document
MD5 hash →	9e107d9d372bb6826bd81d3542a419d6	The quick brown fox jumps over the lazy dog
REST web service call →	view-person?person-id=12345&format=xml	<Person><id>12345</id.</Person>
SQL query →	SELECT PERSON FROM PEOPLE WHERE PID="12345"	<Person><id>12345</id.</Person>

Using a key-value store

There are three operations performed on a key-value store:

- **put** adds a new key-value pair to the table and will update a value if this key is already present.
- **get** returns the value for any given key, or it may return an error message if there's no key in the key-value store.
- **delete** removes a key and its value from the table, or it may return an error message if there's no key in the key-value store.

Rules

Key-value store has two rules:

- Distinct keys: all the keys in any given key-value store are unique.
- No queries on values: You can't perform queries on the values of the table.

Key-value Store

- Advantages:
    - Maps simple keys to (possibly) more complex values like a huge hashtable (associative array).
    - Relatively simple - primitive to complex data-types
    - Fast.
    - Easily distributed - Horizontally scalable.
- Drawbacks:
    - Lack indexes and good scanning capabilities.
    - Not good for relational data (i.e. doing a JOIN)
    - Only basic CRUD (Create, Read, Update, Delete) operations
    - Lacks more complex queries (i.e. Aggregates and Group By ops)

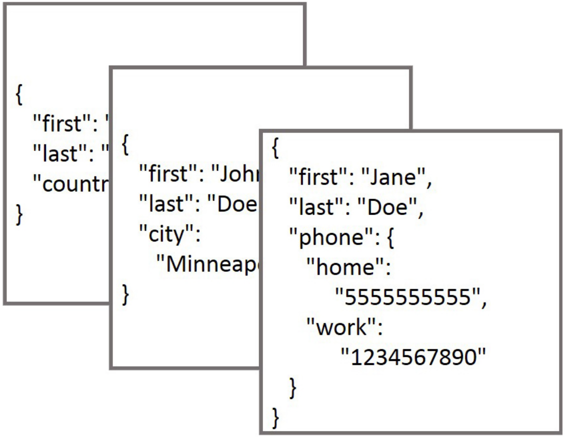
Document stores

Document stores

A document-oriented database (or document store) is a computer program designed for storing, retrieving, and managing document-oriented information

- Document-oriented databases are a subclass of the key-value store
- In a key-value store the data is considered to be inherently opaque
- A document-oriented system relies on internal structure in the document in order to extract metadata that the database engine uses for further optimization
- The document content may be referenced via retrieval or editing methods

Document stores: Example



```
{
  "first": '
  "last": "
  "countr
}
```

```
{
  "first": "Joh
  "last": "Doe
  "city":
    "Minneap
}
```

```
{
  "first": "Jane",
  "last": "Doe",
  "phone": {
    "home":
        "5555555555",
    "work":
        "1234567890"
  }
}
```

Document Stores

- Advantages
  - Document databases allow for any number of fields per data element
  - Assume a standard file-encodings per object: XML, YAML, JSON,...
  - Great for highly variable domains unsure of what the schema will be
- Drawbacks
  - Document should contain everything relevant!
  - Denormalizing the data is common (redundant embedded data)

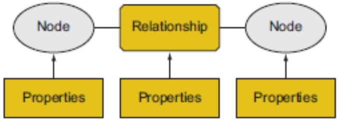
Graph stores

Graph stores

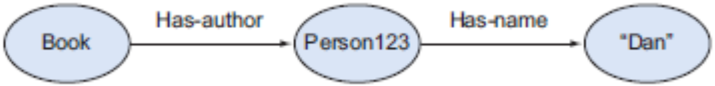
- Is a system that contains a sequence of nodes and relationships that, when combined, create a graph
- Are highly optimized to efficiently store graph nodes and links, and allow you to query these graphs.
- Are useful for any problem that has complex relationships between objects such as social networking

Graph stores

- A graph store has three data fields:
  - nodes, relationships, and properties
- Graph stores allow you to do simple queries that show you the nearest neighboring nodes as well as queries that look deep into networks and quickly find patterns



Graph Store: Example



Graph Stores

- Advantages
  - Queries based on traversing nodes (following edges)
  - Prototypical use cases!: social networking applications, recommendation engines, access control lists, geographic data
- Disadvantages
  - Terrible performance when partitioned in a network!
  - Scales poorly!
  - Not the best choice for tabular analytical data

Column family (Bigtable) stores

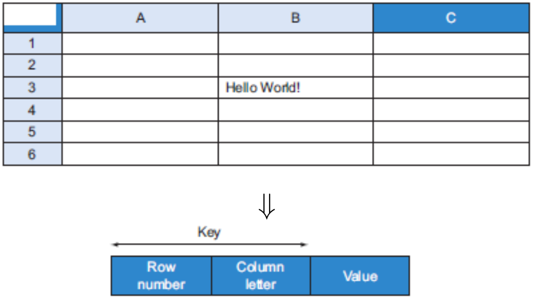
Column family (Bigtable) stores

- Column family stores use row and column identifiers as general purposes keys for data lookup
- They lack features you may expect to find in traditional databases (typed columns, secondary indexes, triggers, and query languages)

Column family and Map Reduce

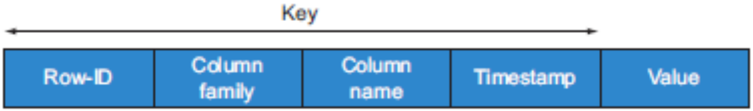
- Column family systems scale to manage large volumes of data
- They're closely tied with many MapReduce systems
  - MapReduce is a framework for performing parallel processing on large datasets across multiple computers (nodes)
  - The map operation breaks up an operation into subparts and distributes each operation to another node for processing
  - The reduce operation collects the results from the other nodes and combines them into the answer to the original problem

Spreadsheet as a Column data store



	A	B	C
1			
2			
3		Hello World!	
4			
5			
6			

Column data store



- a column family is used to group similar column names together
- a timestamp in the key also allows each cell in the table to store multiple versions of a value over time

Benefits of column family systems

- Column family stores are designed to be very big
- Since they don't rely on joins, they tend to scale well on distributed systems
- Column family systems are designed to scale beyond a single processor
- They leverage advanced hashing and indexing to perform probabilistic analysis on large data sets
- Column family implementations are designed to work with distributed filesystems

Column family stores

- Advantages
  - Columnar databases store like data by columns, rather than keeping data together by rows
  - Rows are queried by matching keys
  - Adding Columns are trivial
  - Versioning of data is often built-in
  - Good for Big Data problems:
    - Reading very large number of rows for a small number of columns
- Disadvantages
  - Design of schema based on how the data will be queried
  - Not the best solution for OLTP (Online Transaction Processing), where all columns in a row are often accessed and updated

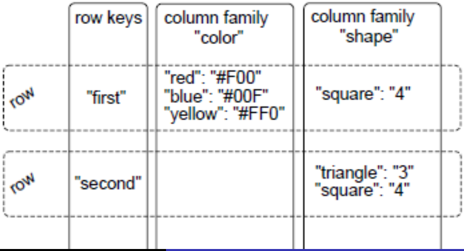
HBase

- HBase is a column-oriented database
- HBase is built on Hadoop —a sturdy, scalable computing platform that provides a distributed file system and map reduce capabilities
- Facebook, Twitter, eBay use HBase

HBase Table

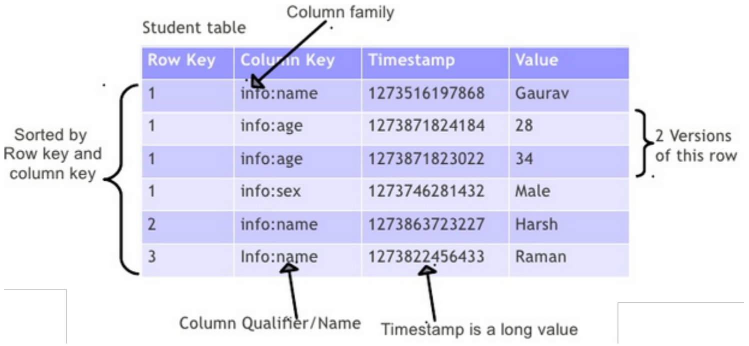
In an HBase table:

- keys are arbitrary strings that each map to a row of data.
- A row is itself a map (a key-value pair):
  - keys are called columns
  - values are uninterpreted 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 ('columnFamily:columnName')



HTable: Timestamps

- HBase stores an integer timestamp for all data values
- When a new value is written to the same cell, the old value hangs around, indexed by its timestamp


Student table
Column family
Sorted by Row key and column key
2 Versions of this row
Column Qualifier/Name
Timestamp is a long value

Row Key	Column Key	Timestamp	Value
1	info:name	1273516197868	Gaurav
1	info:age	1273871824184	28
1	info:age	1273871823022	34
1	info:sex	1273746281432	Male
2	info:name	1273863723227	Harsh
3	Info:name	1273822456433	Raman

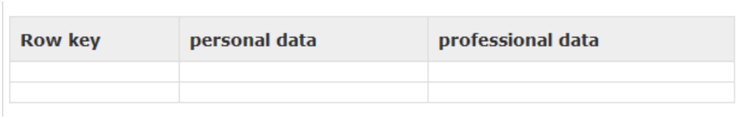
Create

```
create 'tableName', 'columnFamily' [, ...]
```

You can create a table using the create command, here you must specify the table name and the column family names

Create Example

```
hbase> create ’emp’, ’personal data’, ’professional data’
```



Row key	personal data	professional data

Column Family Options

- Number of Versions: Sets how many versions of a cell to keep

```
alter 'emp', { NAME => 'personal data',  VERSIONS => 1 }
```

- Compression:

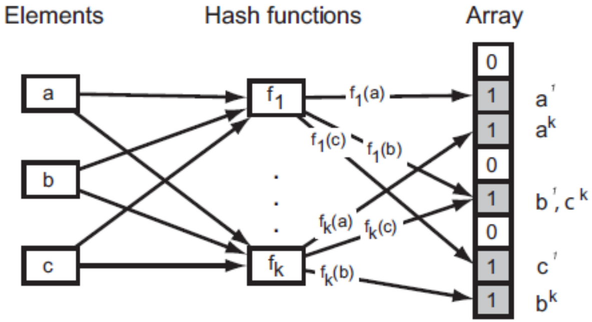
```
alter 'emp', { NAME => 'personal statement', COMPRESSION => 'GZ' }
```

- Bloom filters: to speed up lookups

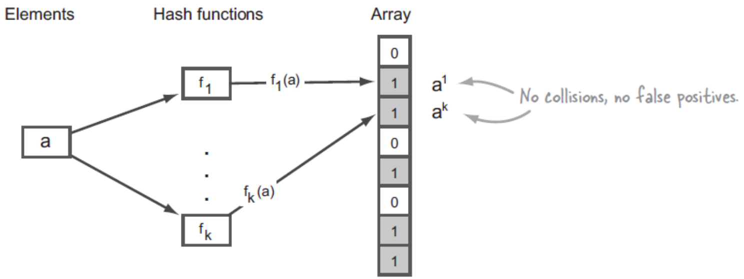
```
alter 'emp', {NAME=>'personal data', BLOOMFILTER=>'ROW'}
```

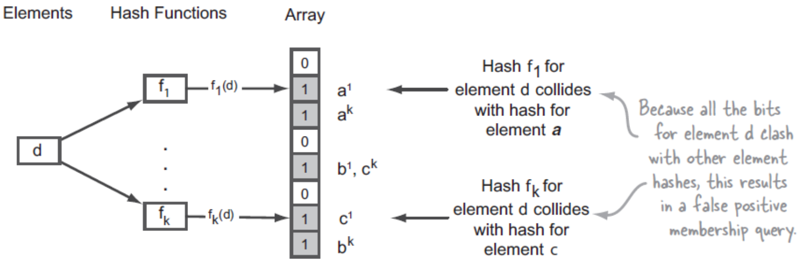
Bloom Filters

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not, a query returns either "possibly in set" or "definitely not in set"



True and False Positives




Elements
Hash Functions
Array
d
f1
fk
f1(d)
fk(d)
0
1
1
0
1
0
1
1
a1
ak
b1, ck
c1
bk
Hash f1 for element d collides with hash for element a
Hash fk for element d collides with hash for element c
Because all the bits for element d clash with other element hashes, this results in a false positive membership query.

Number of hash functions and bits

For a given m (bits) and n (inserted documents), the value of k (the number of hash functions) that minimizes the false positive probability is:

$$k = \frac{m}{n} \ln 2$$

For a given p (false positive rate) and n (inserted documents), the value of m (the required number of bits is):

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

Exercise

Let's create a Bloom filter W with 11 bits and 2 hash functions as follows:

$$h_1(x) = x \ mod \ 11$$

$$h_2(x) = (2x + 1) mod \ 11$$

Initialize W and add three elements (14,20,9) to it. For each of the three operations defined below, write out the state of the Bloom filter after the conclusion of each operation.

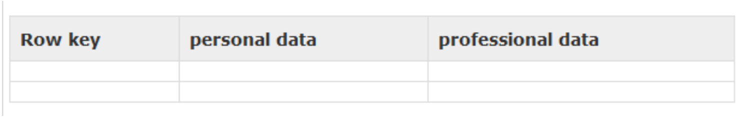
Describe

```
describe 'tableName'
```

Describe the named table

Describe Example

```
hbase> describe 'emp'
Table emp is ENABLED
emp
COLUMN FAMILIES DESCRIPTION
{NAME => 'personal data', BLOOMFILTER => 'ROW', VERSIONS => '1',
 IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'FALSE',
 DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
 COMPRESSION => 'NONE', MIN_VERSIONS => '0',
 BLOCKCACHE => 'true', BLOCKSIZE => '65536',
 REPLICATION_SCOPE => '0'}
{NAME => 'professional data', BLOOMFILTER => 'ROW', VERSIONS => '1',
 IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'FALSE',
 DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER',
 COMPRESSION => 'NONE', MIN_VERSIONS => '0',
 BLOCKCACHE => 'true', BLOCKSIZE => '65536',
 REPLICATION_SCOPE => '0'}
```



Row key	personal data	professional data

List

```
list
```

List all tables in hbase

List Example

```
hbase> list
TABLE
emp
wiki
2 row(s) in 0.0260 seconds

=> ["emp", "wiki"]
```

Enable/Disable

```
disable 'tableName'
enable 'tableName'
```

To delete a table or change its settings, you need to first disable the table using the disable command. You can re-enable it using the enable command

Drop

```
drop 'tableName'
```

Using the drop command, you can delete a table. Before dropping a table, you have to disable it.

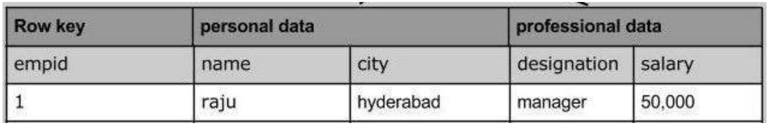
Put

```
put 'tableName','row1','columnFamily:columnName',
    'value'
```

Using put command, you can insert rows into a table

Example Put

```
hbase> put 'emp','1','personal data:name','raju'
hbase> put 'emp','1','personal data:city','hyderabad'
hbase> put 'emp','1','professional data:designation', 'manager'
hbase> put 'emp','1','professional data:salary','50000'
```



Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000

Get

```
get 'tableName','rowid'
```

Using get command, you can get a single row of data at a time

```
get 'tableName', 'rowid',
{COLUMN  => 'columnFamily'}
```

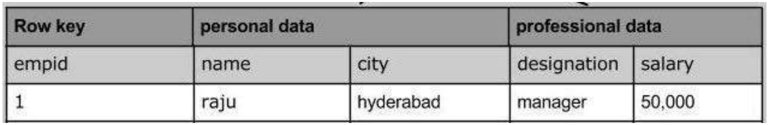
Using get command, you can read a column family

```
get 'tableName', 'rowid',
{COLUMN  => 'columnFamily:columnName'}
```

Using get command, you can read a specific column

Example Get

```
hbase> get 'emp', '1', {COLUMN => 'personal data'}
COLUMN                CELL
personal data:city    timestamp=1480929313676, value=hyderabad
personal data:name    timestamp=1480929305706, value=raju
```



Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000

Get and Bloom filter

There are two types of Bloom filters:

1. `ROW` determines whether a given row key exists at all

   Use this when your query pattern is row key

2. `ROWCOL` determines whether a particular column exists for a given row key

   Use this when your query pattern uses row key and column qualifier

Delete

```
delete 'tableName', 'rowid', 'columnFamily:columnName'
```

Using the delete command, you can delete a specific cell in a table

```
deleteall 'tableName', 'rowid'
```

Deletes all cells in a given row

Example Delete

```
hbase> delete 'emp' , '1', 'professional data:designation'
0 row(s) in 0.0390 seconds

hbase> get 'emp','1'
COLUMN                          CELL
personal data:city             timestamp=1480929313676, value=hyderabad
personal data:name             timestamp=1480929305706, value=raju
professional data:salary       timestamp=1480929339993, value=50000
```

Count

```
count 'tableName'
```

You can count the number of rows of a table using the count command

Truncate

```
truncate 'tableName'
```

This command disables, drops and recreates a table.

Example Truncate

```
hbase> truncate 'emp'
Truncating 'emp' table (it may take a while):
 - Disabling table...
 - Truncating table...
0 row(s) in 3.6330 seconds

hbase> count 'emp'
0 row(s) in 0.1440 seconds

=> 0
```

Scan

```
scan 'tableName'
```

The scan command is used to view the data in HTable. Using the scan command, you can get the table data.
It can include optionally a dictionary of scanner specifications, see more about this here:

`https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/`

Hbase Shell Tips

- You should NOT end queries with ;
- The semicolon defers execution of the current statement until a statement that doesn't end with a semi-colon is entered.
- Solutions:
  1. Enter another query not ended with ;

```
hbase> create 'emp', 'personal data', 'professional data';
hbase:0* list
0 row(s) in 1.2320 seconds

TABLE
emp
wiki
2 row(s) in 0.0130 seconds
=> ["emp", "wiki"]
```

  2. Type anything not ended with ;

```
hbase> create 'emp', 'personal data', 'professional data';
hbase:0* 1
0 row(s) in 1.2380 seconds
```

Conclusion

In this session we have covered:

- What is NoSQL
- Database Patterns
- HBase

Surveys

- End of module feedback survey
  - You have received an email inviting you for answering some simple questions about this module.
  - Please spend 5-10 minutes to complete the survey. Deadline is
  - Your answers are very important for me and equally important for the department and faculty.

Faculty of NMS Diversity & Inclusion Student Survey
https://kings.onlinesurveys.ac.uk/faculty-of-natural-mathematical-sciences-diversity-i-3

- Deadline: ~~Friday 7 December~~ extended to 20 December.
- Important opportunity for you to tell us about your experiences within the department.
- Let's us understand what issues we should be aware of, measure impact of efforts we've been making to improve our culture, and identify whether there's anything we can do to better support our students.
- Need high response rate so we can be confident that views reported are representative.
- More info in email circulated on 13 November.

Lab Session

OPTIONAL: An introduction to HBase and using HBase from Python

You don't need this for the exam or coursework

Suggested video

`https://www.youtube.com/watch?v=qI_g07C_Q5I`

Suggested Readings

- Chapters 1, 2 on the book “Making Sense of NoSQL”
- Chapter 4 on the book “Data Warehousing in the Age of Big Data”
- Chapter 4 on the book “Seven Databases in Seven Weeks”
- Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006). Bigtable: A Distributed Storage System for Structured Data.
- Bloom, B. H. (1970).Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422-426.

Exercise about Bloom filters

Assume a Bloom filter with k hash functions and a bit vector of size n (all bits are set to 0 initially). Further assume that every position in the bit vector can be selected by each hash function with equal probability. Compute the probability that a particular bit b_i in the bit vector is set to 1 after the insertion of x elements

Bloom filters 2

We have 10^9 emails in a white list, and we need to find if a new email is in the white list (i.e., it is not spam):

- How can the problem be solved with a Bloom filter?
- Demonstrate the impact of the number of hash functions on the false positive probability.

Question about NoSQL

Would you recommend the use of a NoSQL database (e.g., a column- oriented database) for storing information about accounts and their transactions at a worldwide bank? Why or why not?

Question about Hbase

- In Hbase, why is it not a good idea to build your table structure without column families?