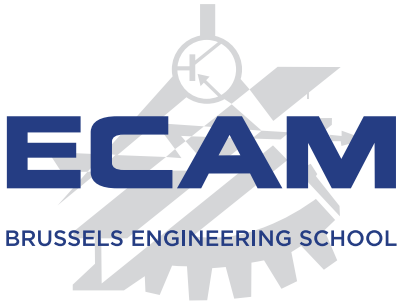


I404B NoSQL

Session 3

Column-Oriented Model: Cassandra, HBase

Sébastien Combéfis

Fall 2019

Objectives

- Column-Oriented model
  - Storing rows or columns on disk
  - The data model
  - Main types of queries
- Examples of column-oriented databases
  - HBase
  - Cassandra

Column-Oriented Model

Column Family (1)

- Column-oriented databases close to relational ones

  Include columns with a given data type

- Follow the BigTable approach brought by Google

  Whose HBase is an open source implementation

- Quick access to data and very good scalability

  In particular with Cassandra and a peer-to-peer distribution

Column Family (2)

- Set of row keys and column families

  Organisation of a database with several tables

- Grouping together data often accessed together

  Each column family is a data map

Row vs. Column (1)

- Disk storage by tuples or by rows

  Initially only a storage issue

- Queries do not often include all columns

  Direct column retrieval from the disk more efficient

ID	**Firstname**	**Class**
16067	Théo	4MIN
15056	Houda	5MIN

Stockage de lignes

ID
16067
15056

Firstname
Théo
Houda

Class
4MIN
5MIN

Stockage de colonnes

Row vs. Column (2)

- Choosing the disk storage to have efficient operations
  - Row storage efficient for writes
  - Row storage efficient for reads
- Reading of a few columns with many rows

  Improve the performances of select queries

	Row storage	**Column storage**
+	Easy to add a record	Only the desired data is read
−	Reading unnecessary data	Writing a tuple requires multiple access

C-Store (1)

- Storing the data in columns in the database

  Created by Brown, Brandeis, MIT and UMass Boston universities

- Based on the relational model and uses SQL

  Does not belong to the NoSQL world, but will inspire it

- Two different storage spaces on the disk

  To better optimise the read and write operations

C-Store (2)

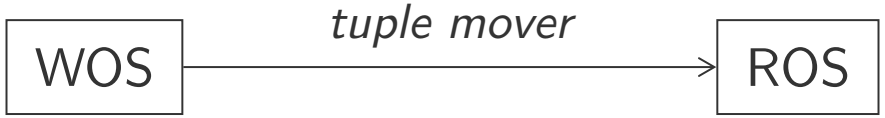
- ROS (*Read Optimized Store*)
  - Storing files containing columns
  - Compressing files depending on the included data types
  - Data sorted by an attribute of the table of the column
- WOS (*Write Optimized Store*)
  - Temporary buffer used for write (INSERT, UPDATE)
  - No compression and vertical partitioning

C-Store (3)

- Regular migration of data from the WOS to the ROS

  Realised by a tuple mover authorised to write in the ROS

- Queries must be able to operate on both stores
  - Insertions directly sent to the WOS
  - Deletions marked in the ROS, then managed by tuple mover
  - Update is a combination of insertions and deletions



Row vs. Column (3)

- No absolute best choice between rows and columns

  It depends on the kind of performed operations

	Rows	**Columns**
Aggregating elements from a column	Slow	Fast
Compression	–	High
Selecting a few columns	Slow (skipping data)	Fast
Insertion/Update	Fast	Slow
Selecting a record	Fast	Slow

Data Model (1)

- A column-oriented base is a two-level map

  Rather than a table structure organised by columns

- A key-value pair identifies a row at the first level

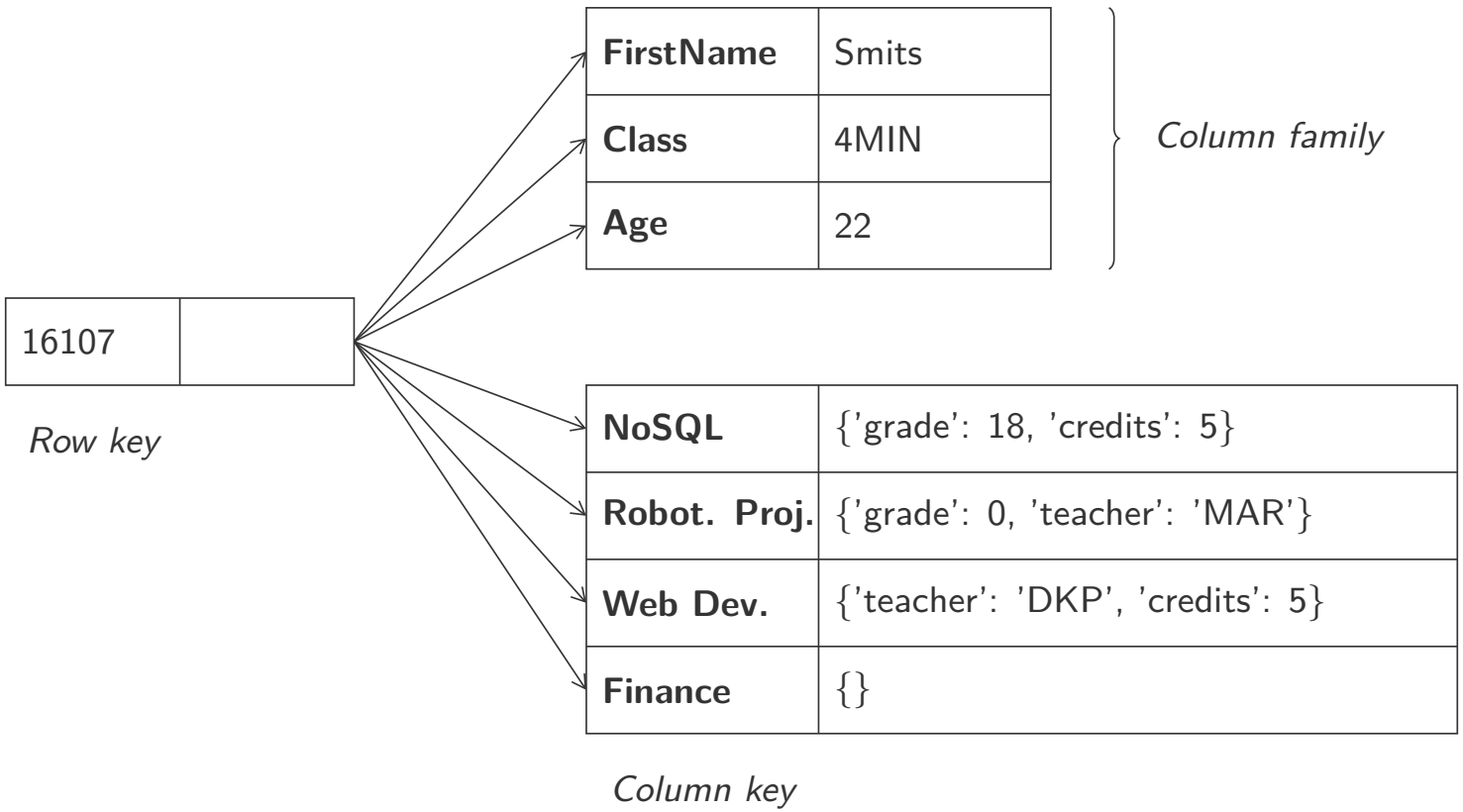
  The key is a row identifier

- A map of columns forming families at the second level
  - Arbitrary number of key-value pairs by row
  - Families for common accesses to columns

Data Model (2)

- Two-level structure combining rows and columns

  Row is the join of records from column families



Data Model (3)

- Column-oriented databases are not really tables
  - Columns can be added to any row
  - Rows can have different column keys
- Defining new column families is rare

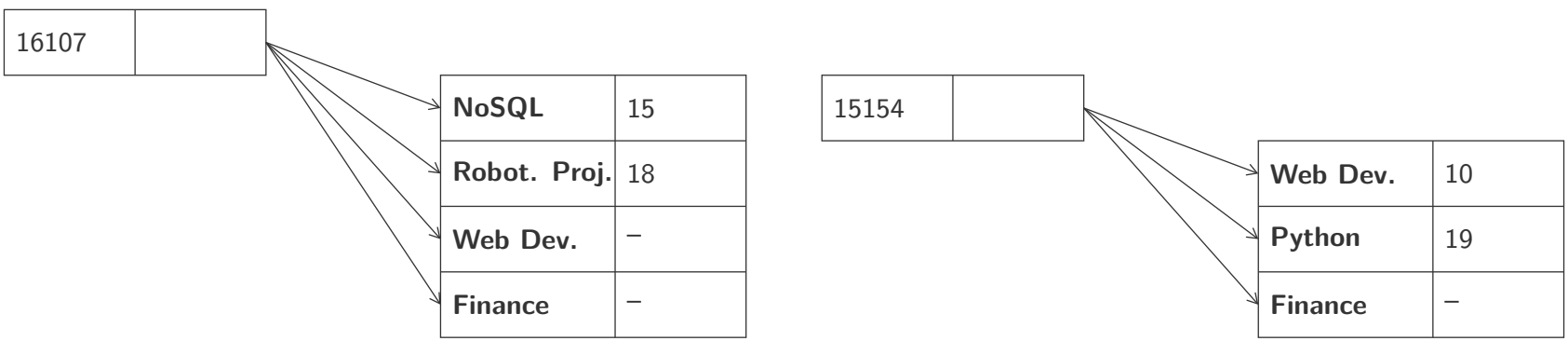
  But adding new column can be done on the fly
- Two kinds of rows depending on the number of columns
  - **Skinny row** few columns and same everywhere (*field-like*)
  - **Wide row** thousands of columns (*list-like*)

Table vs. Column

- Column-oriented databases avoid presence of `NULL`

  Each row only has the columns it should have

Matricule	NoSQL	Robot. Proj.	Web Dev.	Python	Finance
16107	18	0	–	NULL	–
15154	NULL	NULL	10	19	–



Column Advantage

- Efficient read of data only from the necessary columns

  Watch out for tuple reconstruction when reading all

- Better compression rate, but higher CPU usage

  Less entropy since all data from the same domain

- Efficiency of data sorting and indexing

  With redundant storage thanks to space gained by compression

Projection (1)

- Possibility to have physically stored projections

  To improve performances for some query types

Logical table

Region	Customer	Product	Sale
A	G	C	789
B	C	C	743
D	F	D	675
C	C	A	23
A	R	B	654

Super-projection

Region	A	B	D	C	A
Customer	G	C	F	C	R
Product	C	C	D	A	B
Sale	789	743	675	23	654

Projection (2)

- Projections can be sorted on one or several columns

  Improve performance for `SORT` and `GROUP BY` requests

Logical table

Region	Customer	Product	Sale
A	G	C	789
B	C	C	743
D	F	D	675
C	C	A	23
A	R	B	654

Projection 1

Region	A	A	B	C	D
Product	B	C	C	A	D
Sale	654	789	743	23	675

Ease query such as:

```
SELECT Region, Product, SUM(Sale)
GROUP BY Region, Product
```

Projection (3)

- Can be created manually or on the fly

  A bit the same logic than having materialised views

Logical table

Region	Customer	Product	Sale
A	G	C	789
B	C	C	743
D	F	D	675
C	C	A	23
A	R	B	654

Projection 2

Customer	C	C	F	G	R
Sale	743	23	675	789	654

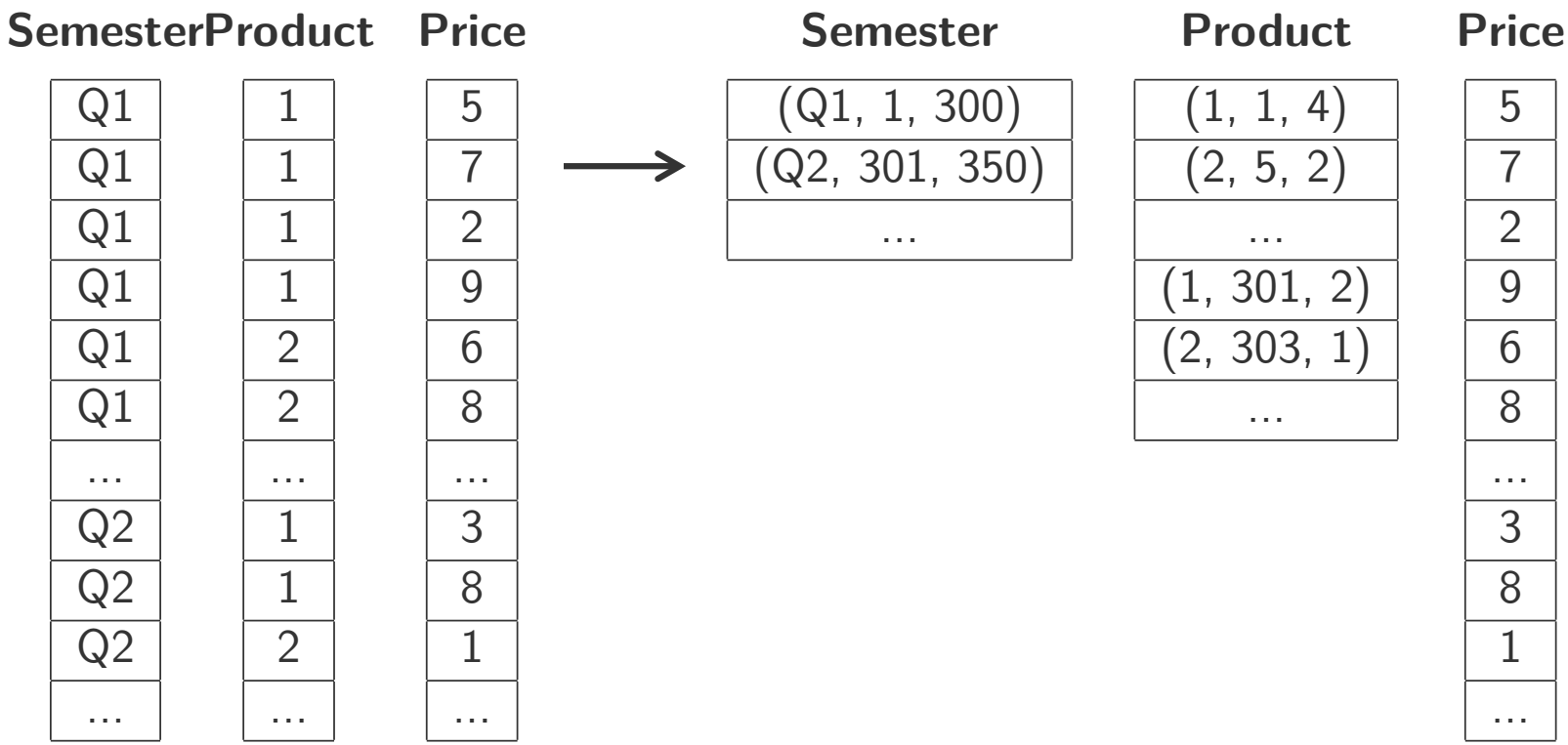
Ease query such as:

```
SELECT Customer, SUM(Sale)
GROUP BY Customer
```

Compression (1)

- Run-Length Encoding on values in the columns

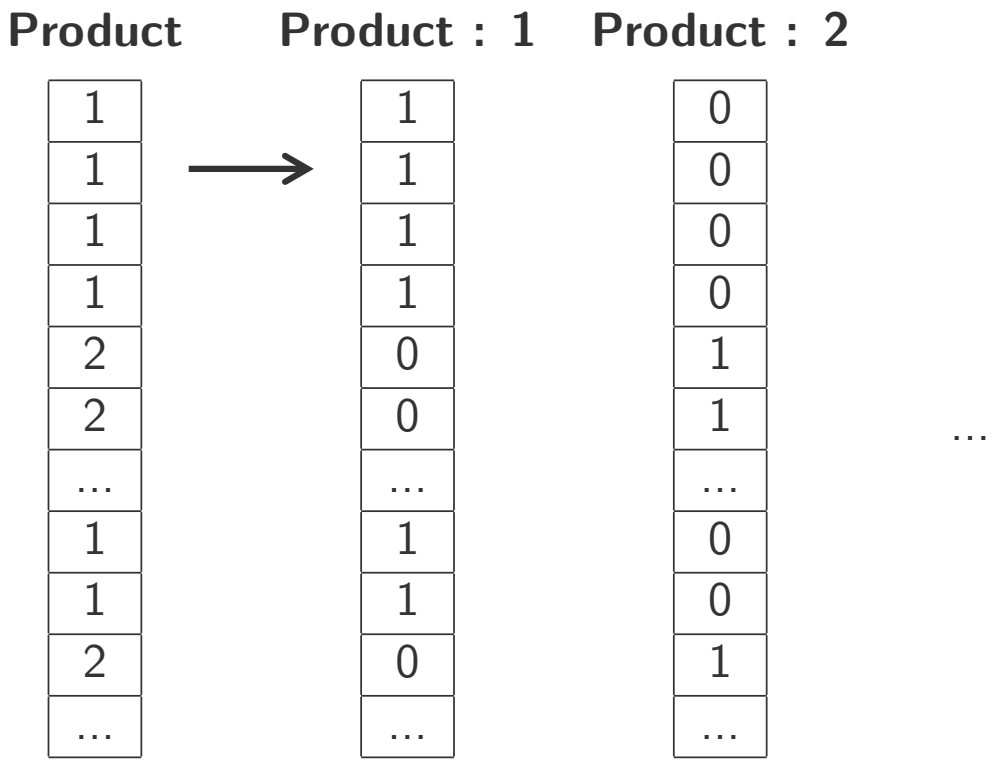
  Convenient when a lot of similar data



Compression (2)

- Bit-Vector Encoding for each unique value of columns

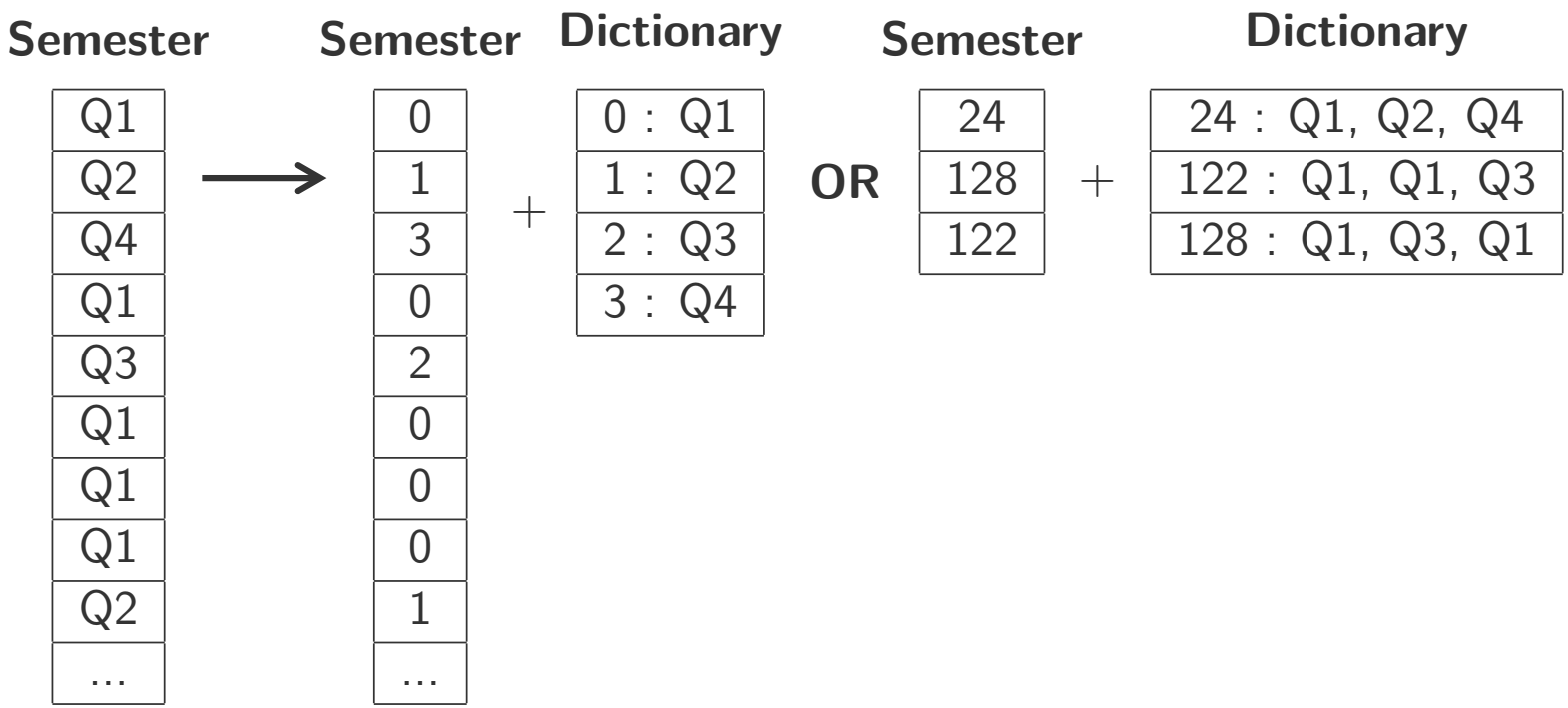
  Convenient when only few unique values, combined with RLE



Compression (3)

- Dictionary for each value or block of values

  Convenient when pattern repetitions



Use Case

- Storing events logs

  State changes or errors found in an application

- Blog posts as part of a CMS

  Tags, categories, links, etc. in different columns of a family

- Count and categorise visitors of a webpage

  Using a particular counter type column

Non-Use Case

- Problems for which ACID must be satisfied for read/write

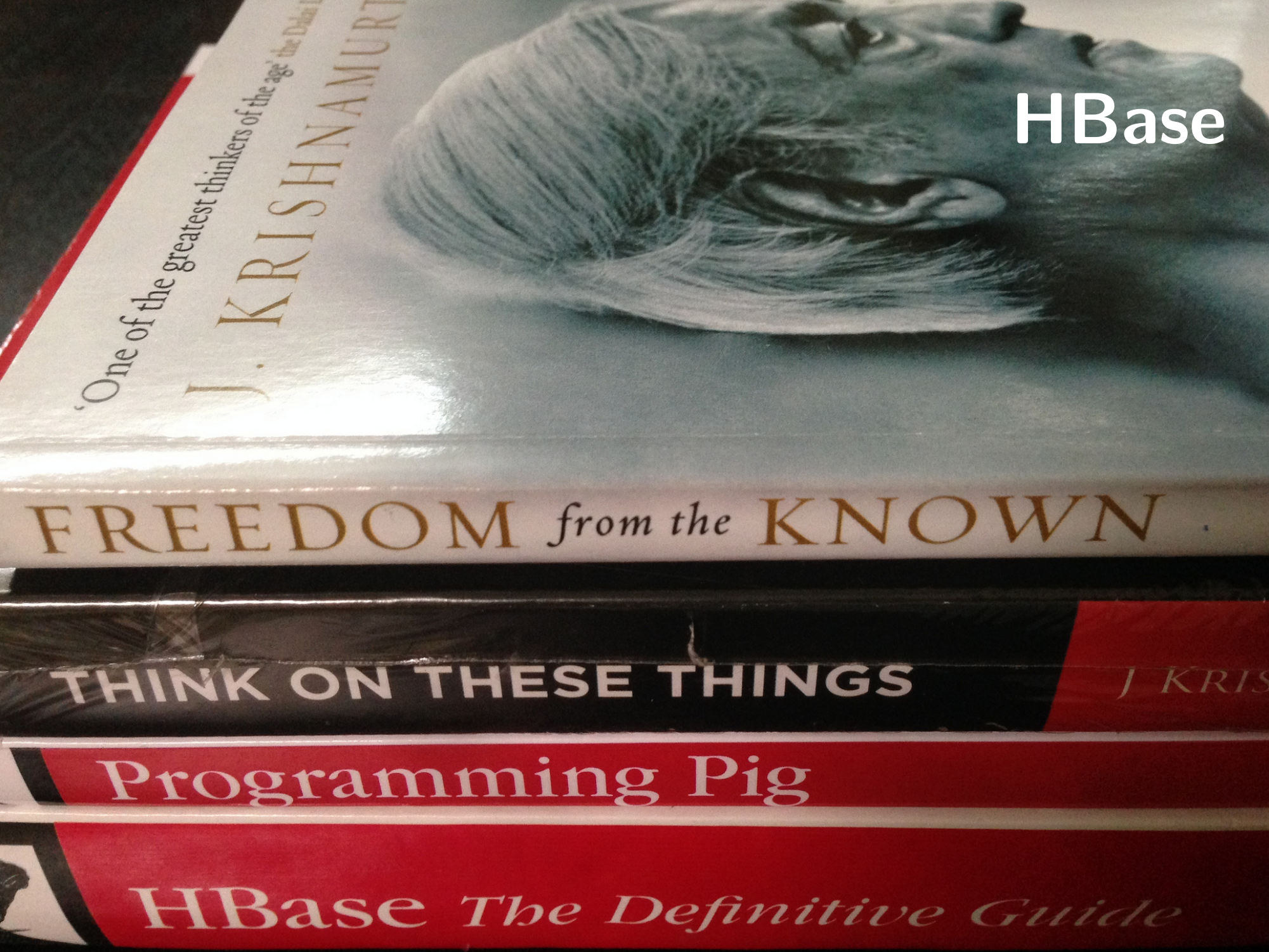
  No ACID transactions with column-oriented databases

- Data aggregation requests (SUM, AVG, etc.)

  First requires to get all the rows on the client side

- Do not use when in a prototyping phase

  The design of column families change with requests to perform


HBase
'One of the greatest thinkers of the age'
J. KRISHNAMURTI
FREEDOM from the KNOWN
THINK ON THESE THINGS
J KRIS
Programming Pig
HBase The Definitive Guide

HBase

- Open source implementation of the BigTable engine by Google

  Is part of the Hadoop project by Apache

- Executed on top of the HDFS file system

  Storage of sparse data while being fault-tolerant

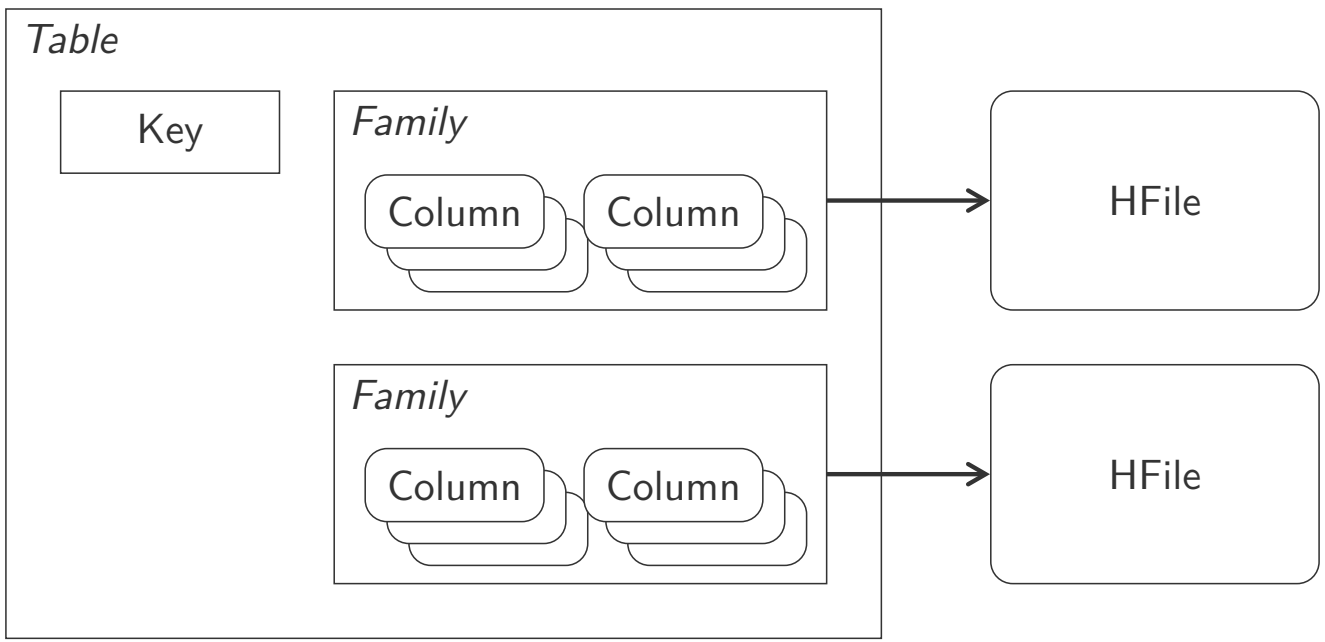
- A DB can serve as input/output of MapReduce (Hadoop)

  Possible to have a SQL layer thanks to Apache Phoenix

Data Model

- Set of versioned column families

  Columns of a given family stored together in a HFile



Path to find a value: Table → Key → Family → Column → Timestamp

Architecture (1)

- Based on Hadoop and HDFS to distribute the storage

  Combination of sharding and replication

- Sharding realised by region servers

  Split in several regions when a table becomes too big

- Replication ensured automatically by HDFS

  File split in blocks replicated with a given factor

Architecture (2)

- Written data are going through several steps
  - First handled in a WAL (*Write-Ahead Log*)
  - Data places in a buffer named *memstore*
- Memstore writes in a HFile on the HDFS when too big

  Sorted set of key-values serialised on disk and immutable
- Deletion managed thank to a *tombstone* marker

  Effective deletion at the same time than compaction

Installing HBase

- HBase is a program written in Java
- Several programs proposed after installation
  - `start-hbase` is a script that starts an HBase server
  - `stop-hbase` is a script that stops an HBase server
  - `hbase` is used to launch several management commands
    - `hbase shell` proposes a command line interface client
    - `hbase thrift` starts the Thrift gateway

Starting the Server

- Starting the server and verifying the connection

  Using `status` to check that everything is good

```
& start-hbase.sh
```

```
& hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.2, r3f671c1ead70d249ea4598f1bbcc5151322b3a13, Fri Jul
  1 08:28:55 CDT 2016

hbase(main):001:0> status
1 active master, 0 backup masters, 1 servers, 0 dead, 2.0000
average load
```

Creating a Table

- Creating a new table with the `create` command

  Specifying column families with the number of versions

```
hbase(main):002:0> create 'students', {NAME => 'infos', VERSIONS
=> 1}, {NAME => 'registrations', VERSIONS => 2}
0 row(s) in 1.2230 seconds

=> Hbase::Table - students

hbase(main):003:0> list
TABLE
students
1 row(s) in 0.0630 seconds

=> ["students"]
```

Adding a Row

- Adding values to different columns with `put`

  Specifying each time the column family

```
hbase(main):004:0> put 'students', '16107', 'infos:firstname', '
Smits'
0 row(s) in 0.1350 seconds

hbase(main):005:0> put 'students', '16107', 'infos:age', '22'
0 row(s) in 0.0120 seconds

hbase(main):006:0> put 'students', '16107', 'registrations:class
', '4MIN'
0 row(s) in 0.0110 seconds

hbase(main):007:0> get 'students', '16107'
COLUMN                          CELL
 infos:age                      timestamp=1477172359150, value=22
 infos:firstname                timestamp=1477172339414, value=Smits
 registrations:class            timestamp=1477172463762, value=4MIN
3 row(s) in 0.0750 seconds
```

New Version of a Column

- Possible to retrieve the different versions of a column

  Using parameters of the `get` command

```
hbase(main):008:0> put 'students', '16107', 'registrations:note',
 'Loves electronics'
0 row(s) in 0.0030 seconds

hbase(main):009:0> put 'students', '16107', 'registrations:note',
 'Loves informatics'
0 row(s) in 0.0030 seconds

hbase(main):010:0> get 'students', '16107', {COLUMN => '
registrations:note', VERSIONS => 2}
COLUMN                          CELL
 registrations:note             timestamp=1477173105470, value=Loves
 informatics
 registrations:note             timestamp=1477173102196, value=Loves
 electronics
2 row(s) in 0.0110 seconds
```

happybase Python module

- happybase Python module to query the database

  Thrift gateway to start with `hbase thrift start`

```
1  import happybase
2
3  connection = happybase.Connection('localhost')
4  print(connection.tables())
5
6  table = connection.table('students')
7  print(table)
```

```
[b'students']
<happybase.table.Table name=b'students'>
```

Inserting a Column

- Columns insertion with the `put` method of the table

  The different columns are provided by a dictionary

- Row columns retrieval with the `row` method

```
1  table.put('15154', {
2      'infos:firstname': 'Mathias',
3      'infos:sex': 'M',
4      'registrations:class': '4MIN'
5  })
6  print(table.row('15154'))
```

```
{b'infos:sex': b'M', b'infos:firstname': b'Mathias', b'
registrations:class': b'4MIN'}
```

Retrieving Columns

- Retrieving a row with `row` and several with `rows`

  Possible to filter the columns to only keep the desired ones

```
1  users = [b'16107', b'15154']
2  classes = {}
3  rows = table.rows(users, columns=[b'infos:firstname', b'
   registrations:class'])
4  for key, value in rows:
5      students = classes.setdefault(value[b'registrations:class'],
       set())
6      students.add(value[b'infos:firstname'])
7  print(classes)
```

```
{b'4MIN': {b'Mathias'}, b'4MIN': {b'Smits'}}
```


Cassandra

Cassandra

- Originally developed by Facebook and open sourced in 2008

  Is not part of Apache's lap

- Fast and scalable database, peer-to-peer replication on cluster

  Commodity servers, no single point of failure

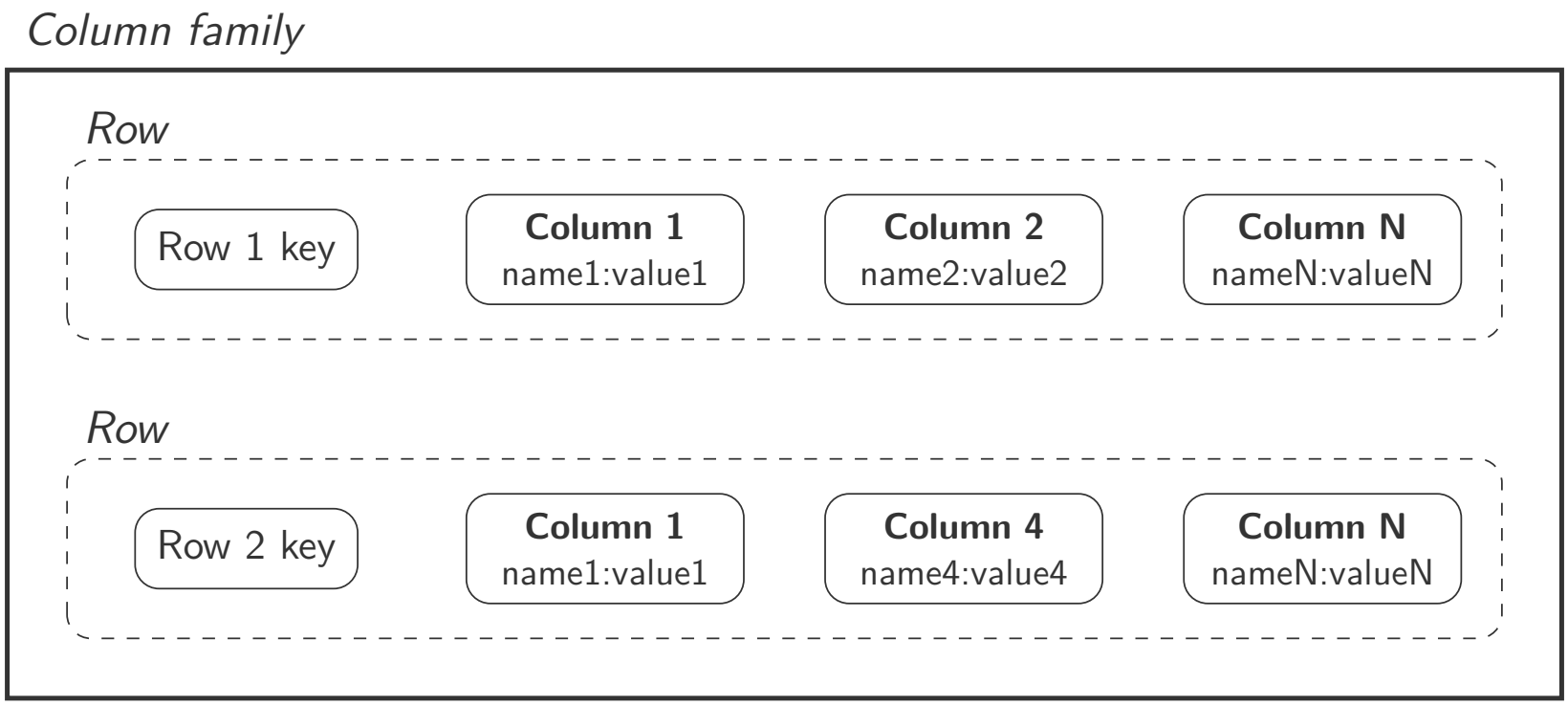
- Query language Cassandra Query Language (CQL)

  Variant of SQL to query Cassandra keyspaces

Data Model

- Column families set with rows

  Rows can contain different columns of the family



Column

- A column is a key-value pair with a timestamp

  The name of the column also plays the role of a key

- The timestamp defines the lifetime of the column

  And write conflict resolution, stale data, etc.

```
{
  name: "FirstName",
  value: "Smits",
  timestamp: 1234567890
}
```

```
{
  name: "Class",
  value: "4MIN",
  timestamp: 1234567890
}
```

Standard Column Family

- A row is a collection of columns

  A key is attached to this collection of columns

- A column family is a collection of similar rows

  Columns are simple, just a name and a value

```
{
  smits: {                          # row with 3 columns, key "smits"
    FirstName: "Smits",
    Class: "4MIN",
    Age: 22
  },
  mathias: {                        # row with 3 columns, key "mathias"
    FirstName: "Mathias",
    Class: "4MIN",
    Sex: "M"
  }
}
```

Supercolumn

- The value of a supercolumn is a map

  "Several columns" as the value of a column

- A supercolumn is a container of columns

  Each contained column has a timestamp

```
{
  name: "04020",
  value: {
    name: "Data acquisition and treatment",
    coordinator: "MCH",
    credits: 4
  },
  timestamp: 1234567890
}
```

Supercolumn Family

- A supercolumn family gathers supercolumns

  Watch out that Cassandra retrieves all, not always optimal

```
{
  3BE: {
    E3050: {
      name: "Signals, systems and telecommunications",
      coordinator: "DBR",
      credits: 6
    },
    E3010: {
      name: "Microcontroller and Logic Design",
      coordinator: "FLE",
      credits: 6
    }
  },
  4MIN: {
    04020: {
      name: "Data acquisition and treatment",
      credits: 4
    }
  }
}
```

Keyspace

- Cassandra organises the column families into keyspaces

  Acts like a namespace for column families

- Similar to the notion of base of relational engines

  Gathering families linked to a same application

Installing Cassandre

- Cassandra is a program written in Java
- Several programs proposed after installation
  - `cassandra` starts a Cassandra server
  - `cqlsh` is a client command line interface
  - `nodetool` gives information about Cassandra server

Starting the Server

- Starting the server and checking the connection

  Immediate indication of whether a server has been found

```
& cassandra
```

```
& cqlsh
Connected to Test Cluster at localhost:9042.
[cqlsh 5.0.1 | Cassandra 3.7 | CQL spec 3.4.2 | Native protocol
v4]
Use HELP for help.
cqlsh>
```

Executing a Query

- Obtaining information on the cluster with a CQL query

  Information retrieved from the `system.local` table

- Great similarity with SQL queries

```
cqlsh> SELECT cluster_name, listen_address FROM system.local;

 cluster_name | listen_address
--------------+----------------
 Test Cluster |      127.0.0.1

(1 rows)
```

Information on the Base

- Obtaining information with the `DESCRIBE` command

  Description of `cluster`, `keyspaces`, `tables`, etc.

```
cqlsh> DESCRIBE CLUSTER;

Cluster: Test Cluster
Partitioner: Murmur3Partitioner

cqlsh> DESCRIBE KEYSPACES;

system_traces  system_schema  system_auth  system
system_distributed

cqlsh> DESCRIBE TABLES;

Keyspace system_traces
----------------------
events  sessions
[...]
```

Creating a Keyspace

- Creating a new keyspace with `CREATE KEYSPACE`

  Configuring the keyspace properties, for example replication

- Example with simple replication with a given factor

```
cqlsh> CREATE KEYSPACE myschool
   ... WITH replication={'class': 'SimpleStrategy', '
   replication_factor': 3};

cqlsh> DESCRIBE keyspaces;

myschool  system_schema  system_auth  system  system_distributed
 system_traces

cqlsh> USE myschool;
cqlsh:myschool>
```

Creating a Table

- Creating a new table with `CREATE TABLE`

  Definition of the different columns of the table

- Primary key to uniquely identify rows

```
cqlsh:myschool> CREATE TABLE students (
           ... serial int PRIMARY KEY,
           ... firstName text,
           ... class text,
           ... age int,
           ... sesque text
           ... );

cqlsh:myschool> SELECT * FROM students;

 serial | age | class | firstName | sesque
--------+-----+-------+-----------+--------

(0 rows)
```

Adding and Removing Column

- The table structure can be changed with `ALTER TABLE`

  Possibility to add and remove columns

- Example of a correction of the column sesque in sex

```
cqlsh:myschool> ALTER TABLE students DROP sesque;

cqlsh:myschool> ALTER TABLE students ADD sex text;

cqlsh:myschool> SELECT * FROM students;

 serial | age | class | firstName | sex
--------+-----+-------+-----------+-----

(0 rows)
```

Adding Row

- Adding a row in the table with `INSERT INTO`

  Specifying the columns for which there is a value to set

- Example of adding Smits in the `students` table

```
cqlsh:myschool> INSERT INTO students (serial, firstName, class,
age)
            ... VALUES (16107, 'Smits', '4MIN', 22);
cqlsh:myschool> SELECT * FROM students;

 serial | age | class | firstName | sex
--------+-----+-------+-----------+------
  16107 |  22 |  4MIN |     Smits | null

(1 rows)
```

Other CRUD Operations

- Three other CRUD operations as with SQL
  - **Update** rows

    ```
    UPDATE table SET n1=v1, n2=v2...  WHERE cond
    ```

  - **Read** rows

    ```
    SELECT c1, c2...  FROM table WHERE cond
    ```

  - **Delete** rows

    ```
    DELETE c1, c2...  FROM table WHERE cond
    ```

- Operation on a single row with a condition on its key

  Not specifying `c1, c2...` acts on a whole column

cassandra Python Module

- cassandra Python Module to query the database

  Creation of a cluster and connection on a keyspace

```
1  from cassandra.cluster import Cluster
2
3  cluster = Cluster(['127.0.0.1'])
4  session = cluster.connect('myschool')
5
6  print(cluster)
7  print(session)
```

```
<cassandra.cluster.Cluster object at 0x1096af240>
<cassandra.cluster.Session object at 0x10a6bed30>
```

Executing a Query

- Using the `execute` method on the session

  Executing a CQL query, retrieving a named tupled

- The `class` column will not be accessible as a field

  Because of a conflict with the `class` property of Python

```
1 rows = session.execute('SELECT * FROM students')
2 for row in rows:
3     print(row)
4     print('=> {} ({} y.o.)'.format(row.firstName, row.age))
```

```
Row(serial=16107, age=22, field_2_='4MIN', firstName='Smits', sex
=None)
=> Smits (22 y.o.)
```

Building a Query

- Query by inserting values in a string

  Similar to formatted outputs

```
1  session.execute(
2      '''
3      INSERT INTO students (serial, firstName, class, sex)
4      VALUES (%s, %s, %s, %s)
5      ''',
6      (15154, 'Mathias', '4MIN', 'M')
7  )
8
9  rows = session.execute('SELECT * FROM students')
10 for row in rows:
11     print(row)
```

```
Row(serial=15154, age=None, field_2_='4MIN', firstName='Mathias',
 sex='M')
Row(serial=16107, age=22, field_2_='4MIN', firstName='Smits', sex
=None)
```

Prepared Query

- Building a prepared query with the `prepare` method

  Then execution with the `execute` method

- Authorise search on a column with `ALLOW FILTERING`

```
1  search_class = session.prepare('SELECT class FROM students WHERE
   firstName=? ALLOW FILTERING')
2
3  users = ['Harold', 'Smits', 'Théo', 'Mathias']
4  classes = {}
5  for user in users:
6      rows = session.execute(search_class, [user])
7      for row in rows:
8          students = classes.setdefault(row[0], set())
9          students.add(user)
10 print(classes)
```

```
{'4MIN': {'Smits', 'Mathias'}}
```

References

- Mangat Rai Modi, *Rowise vs Columnar Database? Theory and in Practice*, January 26, 2018.
  https://medium.com/@mangatmodi/rowise-vs-columnar-database-theory-and-in-practice-53f54c8f6505
- Ameya, *C-Store: A Columnar Database: Introduction*, April 5, 2019.
  https://medium.com/@ameya_s/c-store-a-columnar-database-1fe7e84d7247

Credits

- Logo pictures from Wikipedia.
- zolakoma, August 8, 2008, https://www.flickr.com/photos/zolakoma/2847597889.
- balu, May 15, 2014, https://www.flickr.com/photos/balusss/14004726607.
- Simon Winch, August 23, 2010, https://www.flickr.com/photos/110777427@N06/14184365994.