Hadoop Advanced

SMU, 2019

Nikita Neveditsin
nikita.neveditsin@smu.ca

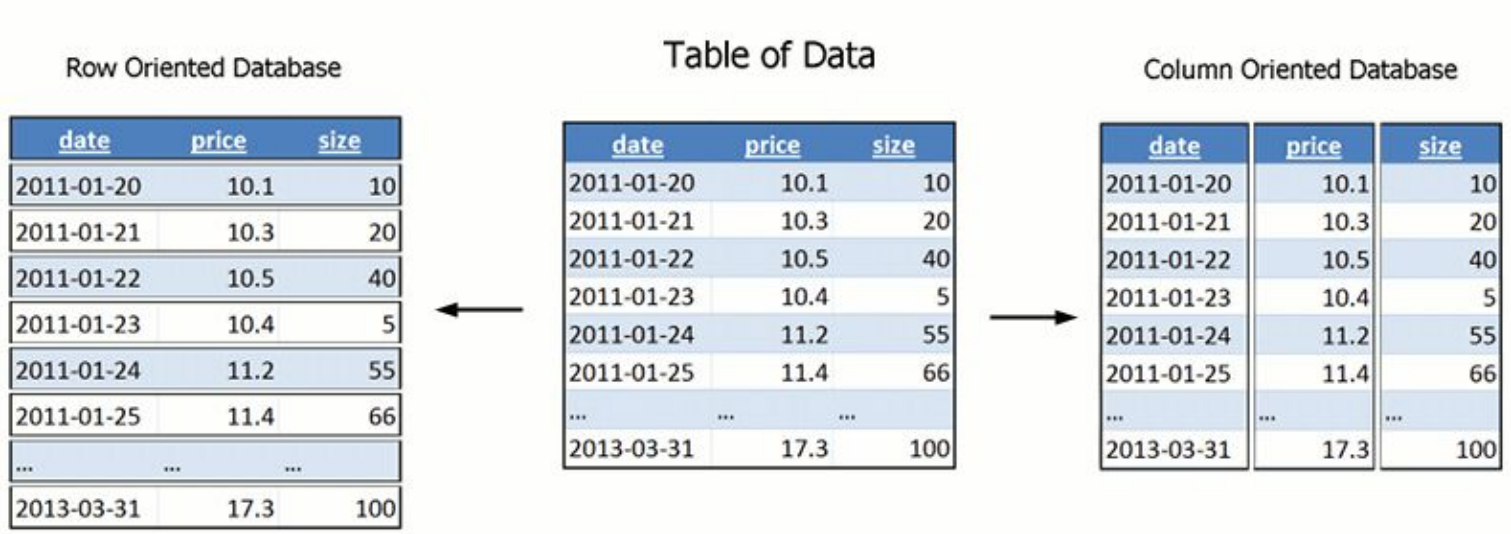
Workshop outline

1. HBase
    - Exercise
2. Pig
    - Exercises
3. Flume
    - Exercise

HBase

HBase is an open-source, **non-relational**, distributed, column-oriented (**column family oriented** if more precisely) database modeled after Google's Bigtable (2006) and written in Java

What does “column-oriented” mean?



date	price	size
2011-01-20	10.1	10
2011-01-21	10.3	20
2011-01-22	10.5	40
2011-01-23	10.4	5
2011-01-24	11.2	55
2011-01-25	11.4	66
...	...	...
2013-03-31	17.3	100

[link]

Relational databases are usually **row-oriented**: they physically store rows (e.g., **all columns together**)

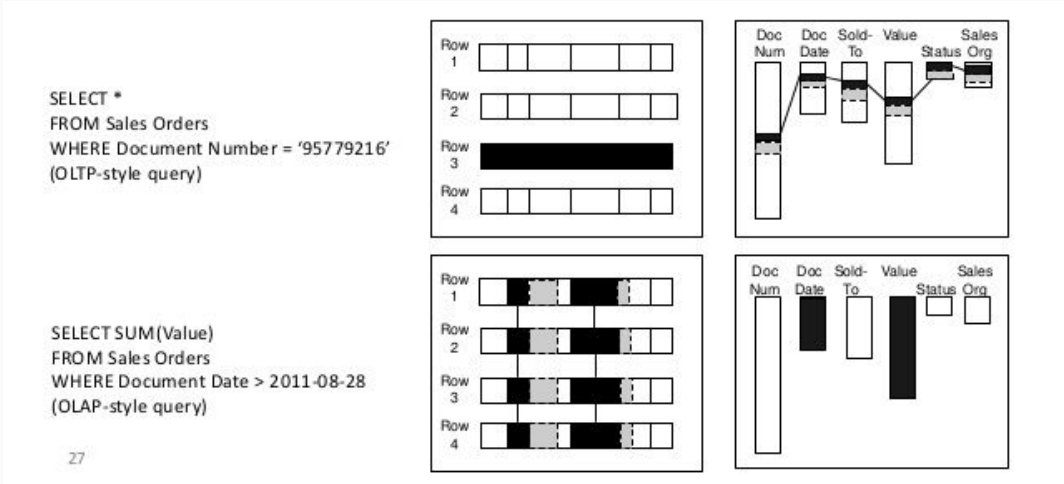
Column-oriented databases physically store each column data **continuously**

What does it give us?

- **Fast aggregation queries** (AVG, MAX, MIN, SUM, etc.) and OLAP queries in general:
  - we don’t need to read all unnecessary data (e.g. columns that are not needed)
  - values of columns are stored continuously
- **Horizontal scalability** (columns can be stored on different physical machines) => implies distributed nature of the database
- **Better compression potential** than in row-oriented database => more space-efficient
- etc.

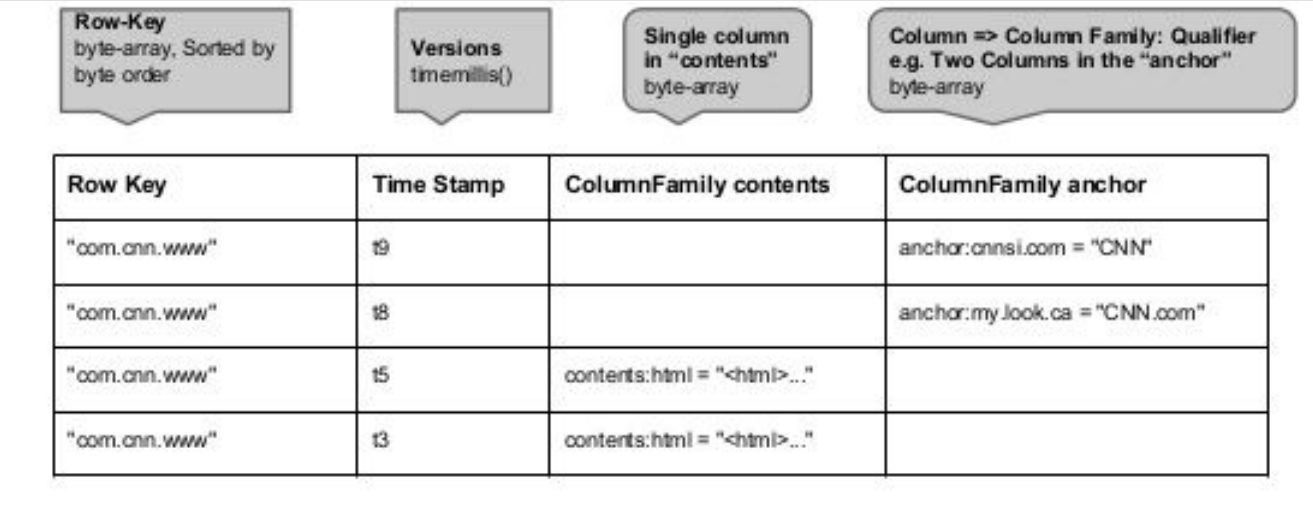
Can they replace row-oriented DB?

- Random inserts in the table are generally heavy (**but not in HBase which is optimized for random r/w**)
- Lack of support of ACID transactions
- In general, **they are not suitable for OLTP and were not designed for it**



[link]

HBase: columns vs column families



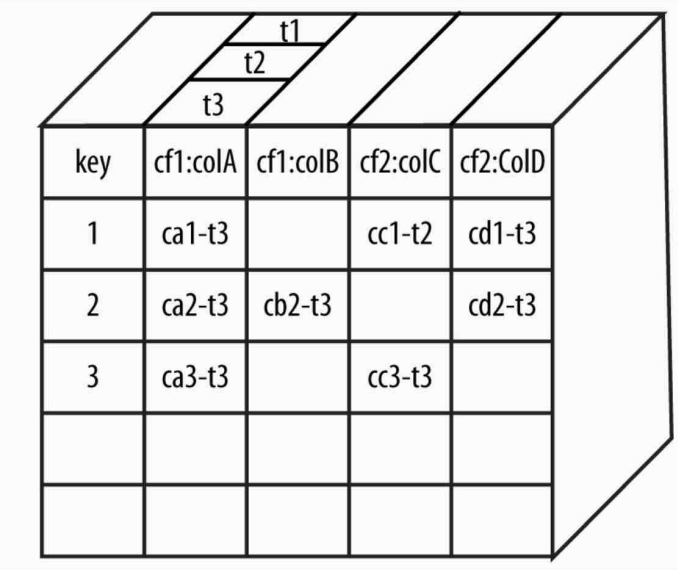
Row Key	Time Stamp	ColumnFamily contents	ColumnFamily anchor
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"
"com.cnn.www"	t5	contents:html = "<html>..."	
"com.cnn.www"	t3	contents:html = "<html>..."	

[link]

- **HBase** is different from plain column-oriented databases
- In **HBase** **row** is a collection of column families
- **Column family** is a collection of columns (usually **related** to each other as they tend to be located on one physical device: e.g., *NAME {fname, lname, midname}*)
- **Column** is a collection of **key value pairs**

HBase: timestamps

- HBase stores **multiple versions of cell** (each time data is inserted/updated - the new version of cell is created with the current timestamp)
- By default, when you read data the **latest** version is returned. However, you may return one of the previous versions of the cell
- How many versions are stored? You can define it for **column family** (default is 3)



[link]

HBase: user’s perspective

- You may compare HBase with a huge multidimensional **map (or dictionary)** when you work with cells:
  - To access a specific **cell** you should specify multiple keys:
    - Row key
    - Column family name (column family name is a **key** here, should be unique in a table)
    - Column name (column name is also a key, should be unique in a column family)
    - Timestamp (optional)
- HBase can also be treated as a **key-value** store where key is a rowId
- You may access rows and columns as you do with regular tables

HBase vs Relational Database

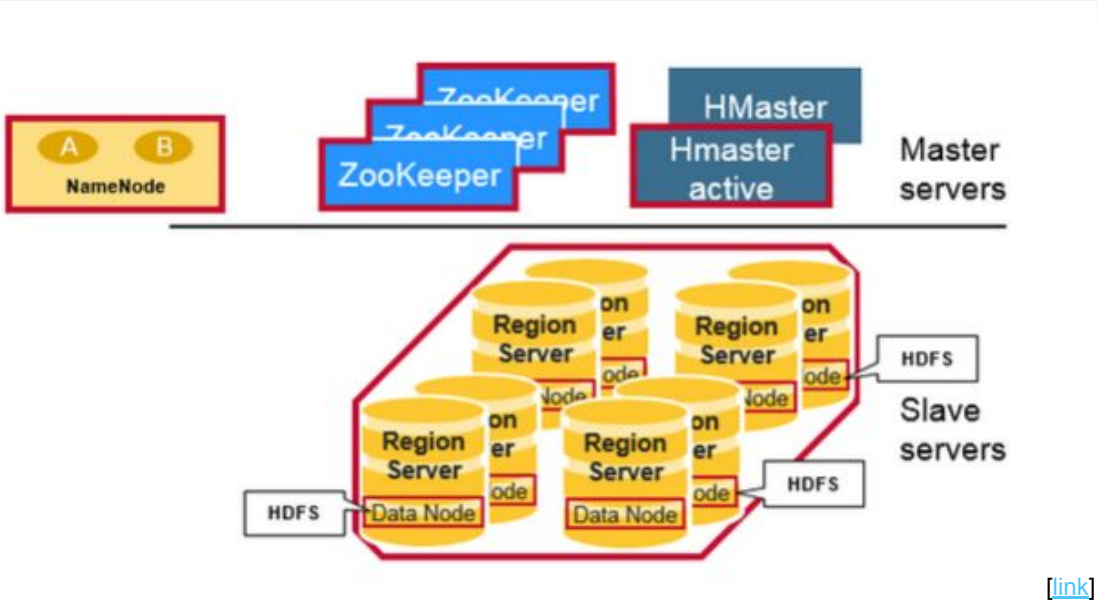
	HBase	Relational DB
Use case	To **store** data	OLTP
SQL	No	Yes
Schema	No schema, only have to define column families (usually not more than 3)	Have to define schema before use
ACID support	No	Yes
Data	Semi-structured or structured	Structured only
Data types	Just array of bytes - client should care about casting. **No NULL values: no value == no column for the row**	CHAR, VARCHAR, INT NULL, etc..

HBase: use cases

- Usually used from applications (you have to write code in most cases)
- When you expect hundreds, thousands (or even millions) of columns and millions to billions of rows
- When you have a *variable schema*: e.g., some rows have one set of columns, another have different set of columns
- Features of relational databases are not required (no joins, transactions, triggers, etc.)
- When you may need to get “flashbacks” of data
- **When you need to access data quickly (e.g., random reads and writes work fast with HBase)**
- When distributed computing is used. There is no point to use HBase on single (even a really powerful) machine
- When you need horizontal scalability and **fault-tolerance** (thank you HDFS)
- When you can integrate with other Hadoop services

HBase: Architecture

- HBase works on top of **HDFS**
- HBase uses **ZooKeeper** as a distributed *coordination* service to maintain server state in the cluster
- **Hmaster**: master node - responsible for region servers assignment, administrative functions (create/update/delete)
- **Region Servers:** slave nodes. Store actual data and metadata, perform reads and writes a (up to ~1000 regions)



[link]

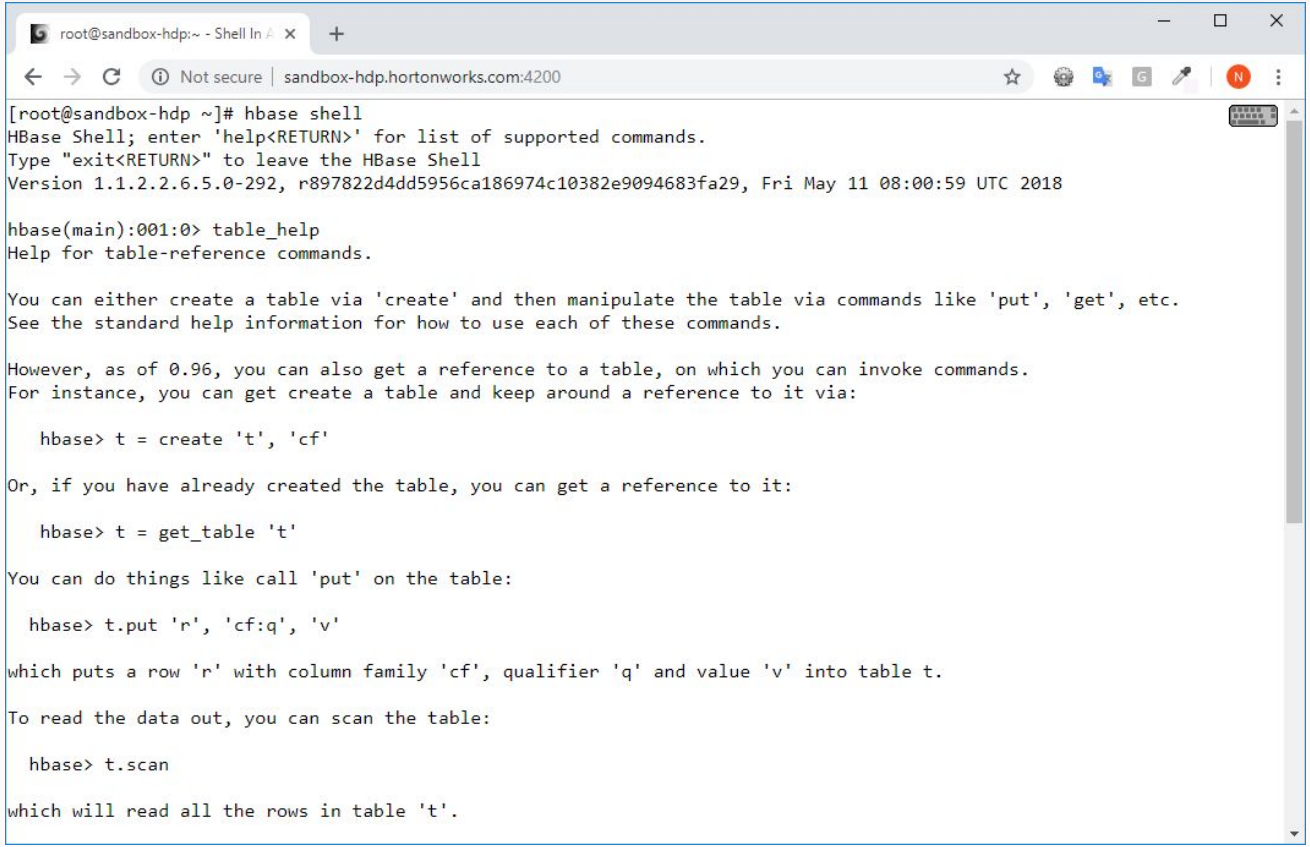
HBase: Important points

- Values are stored in key order (lexicographical order). All data is **sorted**
- No data types in HBase - everything is an array of bytes
- You have to define only **column families** for a table
- Rows may or may not have common **columns**
- Data is stored in HFiles that are stored on region servers. HFiles are immutable. However, they are merged when “**compaction**” is being done. Data is usually written first in memory (cache) then, when cache is full - stored on a disk
- Bloom filters are widely used - they are loaded into memory and speed up access to data

HBase: shell

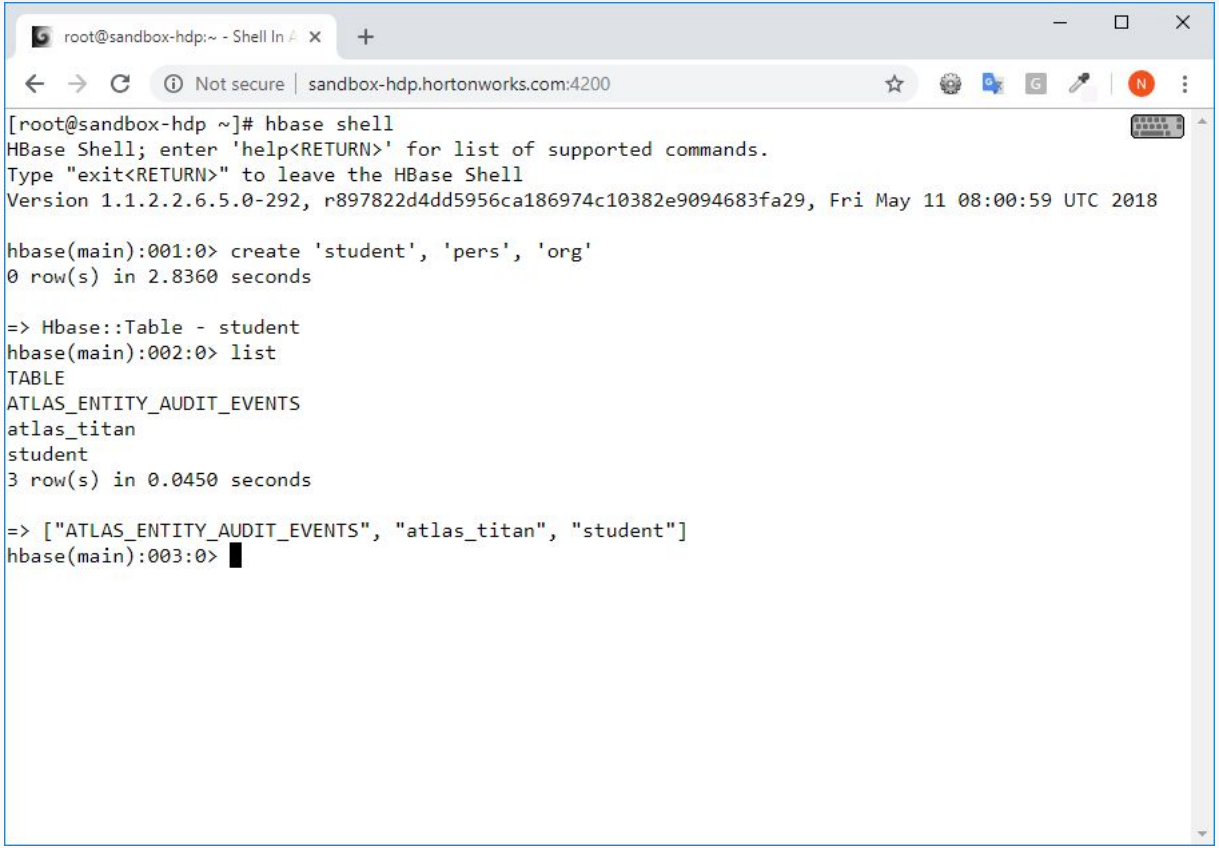
- **hbase shell** command starts shell
- **list** command lists all tables is the database (don't touch the two)

HBase: shell (cont-d)



table_help command is used to display commands available for working with a table

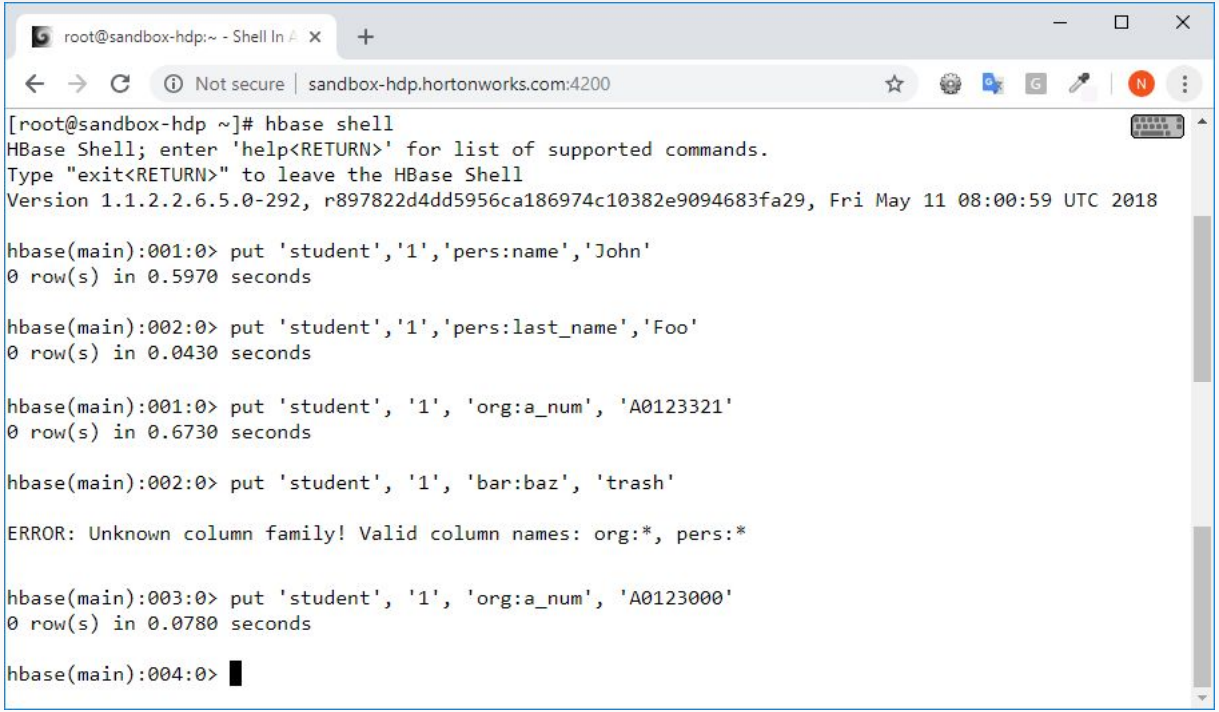
HBase: shell (cont-d)



create command is used to create a new table:

create ‘student’, ‘pers’, ‘org’ creates a table **student** with two ***column families***: ‘pers’ and ‘org’

HBase: shell (cont-d)



put command is used to create or update data in the tabe:

put ‘student’, ‘1’, ‘pers:name’, ‘John’ creates a new row in the table student:

- row_id = ‘1’
- column ‘name’ if column family ‘pers’ = John

HBase: shell (cont-d)

```
hbase(main):006:0> get 'student', 1
COLUMN                    CELL
 org:a_num                timestamp=1549464484010, value=A0123000
 pers:last_name           timestamp=1549464119341, value=Foo
 pers:name                timestamp=1549464099022, value=John
3 row(s) in 0.1130 seconds

hbase(main):007:0> get student, 1
NameError: undefined local variable or method `student' for #<Object:0x65be88ae>

hbase(main):008:0> get 'student', 1
COLUMN                    CELL
 org:a_num                timestamp=1549464484010, value=A0123000
 pers:last_name           timestamp=1549464119341, value=Foo
 pers:name                timestamp=1549464099022, value=John
3 row(s) in 0.0260 seconds

hbase(main):009:0> get 'student', 2
COLUMN                    CELL
 pers:name                timestamp=1549464778458, value=123
1 row(s) in 0.0140 seconds

hbase(main):010:0>
```

get command is used to get data from table by **row_id**:

get ‘student’, 1

gets data from student table for row_id = 1

HBase: shell (cont-d)

```
hbase(main):003:0> scan 'student'
ROW                     COLUMN+CELL
 1                      column=org:a_num, timestamp=1549464484010, value=A0123000
 1                      column=pers:last_name, timestamp=1549464119341, value=Foo
 1                      column=pers:name, timestamp=1549464099022, value=John
1 row(s) in 0.5850 seconds

hbase(main):004:0> put 'student', 2, 'pers:name', 123
0 row(s) in 0.2090 seconds

hbase(main):005:0> scan 'student'
ROW                     COLUMN+CELL
 1                      column=org:a_num, timestamp=1549464484010, value=A0123000
 1                      column=pers:last_name, timestamp=1549464119341, value=Foo
 1                      column=pers:name, timestamp=1549464099022, value=John
 2                      column=pers:name, timestamp=1549464778458, value=123
2 row(s) in 0.0670 seconds

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

scan command is used to view data in table:

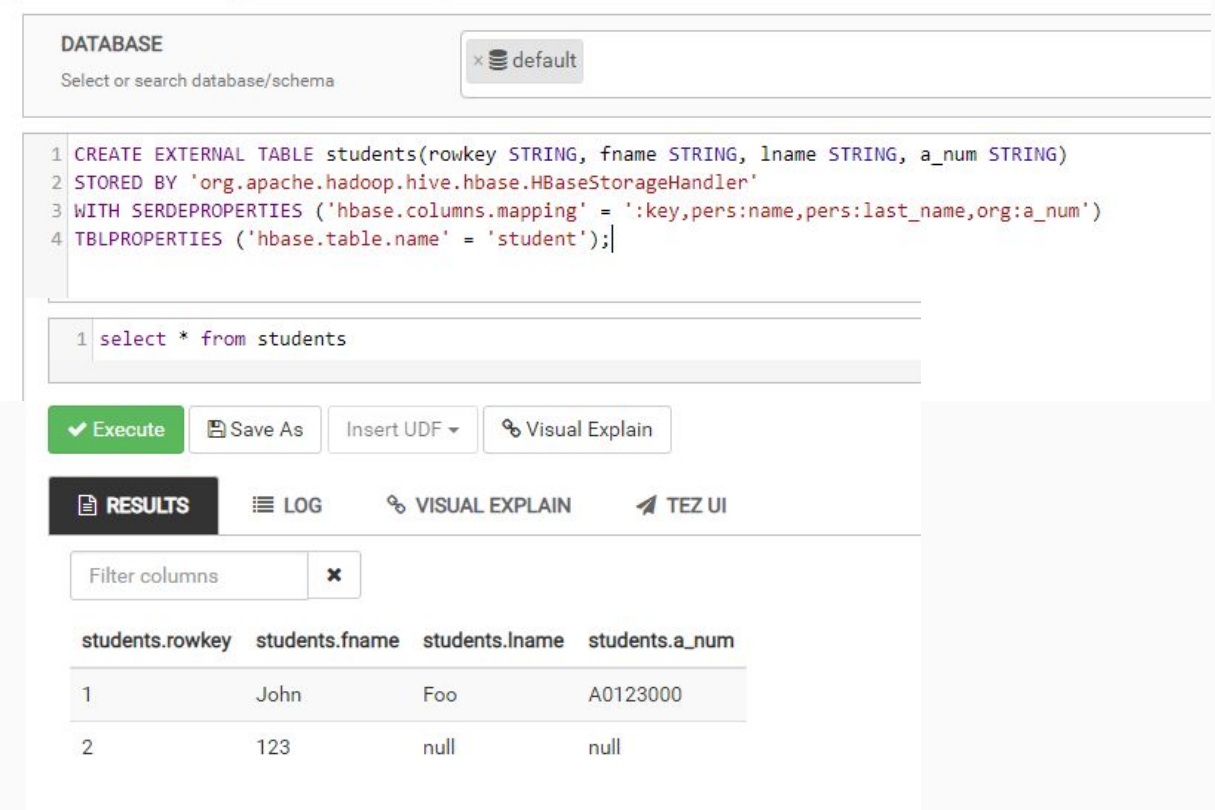
scan 'student'

command prints data from the student table

You can define a filter as in the example below:

```
hbase(main):032:0> scan 'student', { COLUMNS => 'org:a_num', FILTER => "ValueFilter( =, 'binaryprefix:A0123000')"}
ROW                          COLUMN+CELL
 1                           column=org:a_num, timestamp=1549464484010, value=A0123000
1 row(s) in 0.0710 seconds

hbase(main):033:0>
```

HBase to Hive



You may create a Hive table based on HBase table

HBase to Hive: explained

```
CREATE TABLE students(rowkey STRING, fname STRING, lname STRING, a_num STRING)
//we define “schema” for the new table here
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
//just defines HBaseStorageHandler as storage
WITH SERDEPROPERTIES ('hbase.columns.mapping' = ':key,pers:name,pers:last_name,org:a_num')
//Important thing: Mapping:
```

- :key -> rowkey
- pers:name -> fname
- pers:last_name -> lname
- org:a_num -> a_num

```
TBLPROPERTIES ('hbase.table.name' = 'student');
//table name ‘students’ in HBase
```

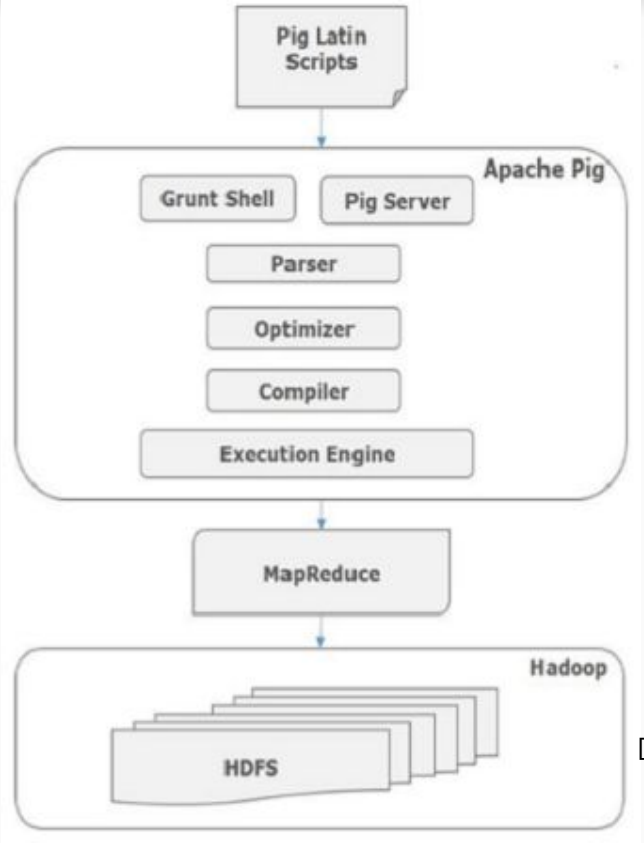
Exercise (may work in groups)

- Using HBase shell define a table (with at least 3 **columns** and 1-3 column families)
- Create at least 3 rows in that table
- Export the table into Hive

Apache Pig: Application architecture

Apache Pig is a high-level platform for creating programs that run on Apache Hadoop. The language for this platform is called Pig Latin. Pig can execute its Hadoop jobs in MapReduce, Apache Tez, or Apache Spark. Developed at Yahoo around 2006

Pig Latin abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming **high level**.



[link]

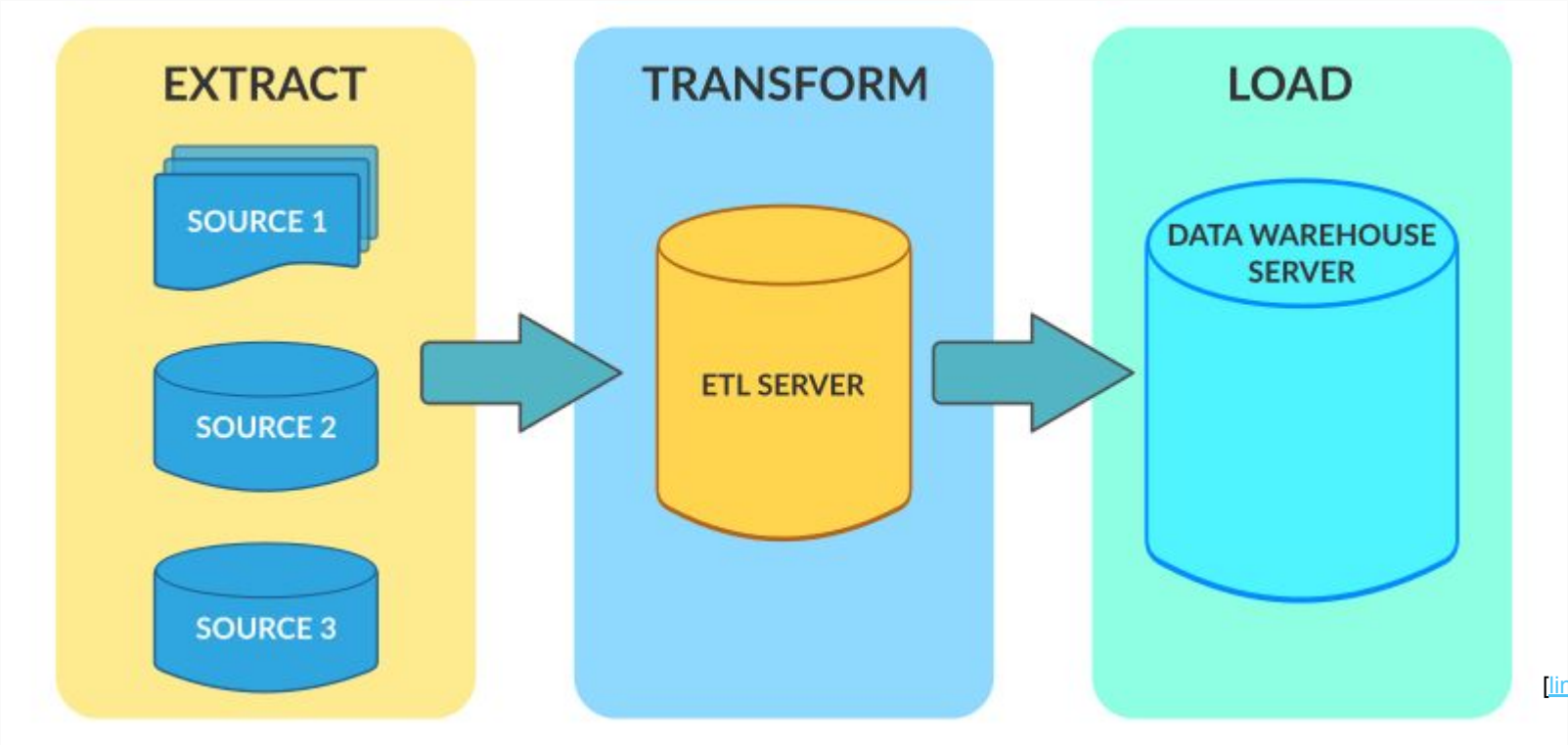
Apache Pig

In simple words: instead of writing complex Java MapReduce applications, you may use Pig Latin language to work with data

Apache Pig vs Apache Hive

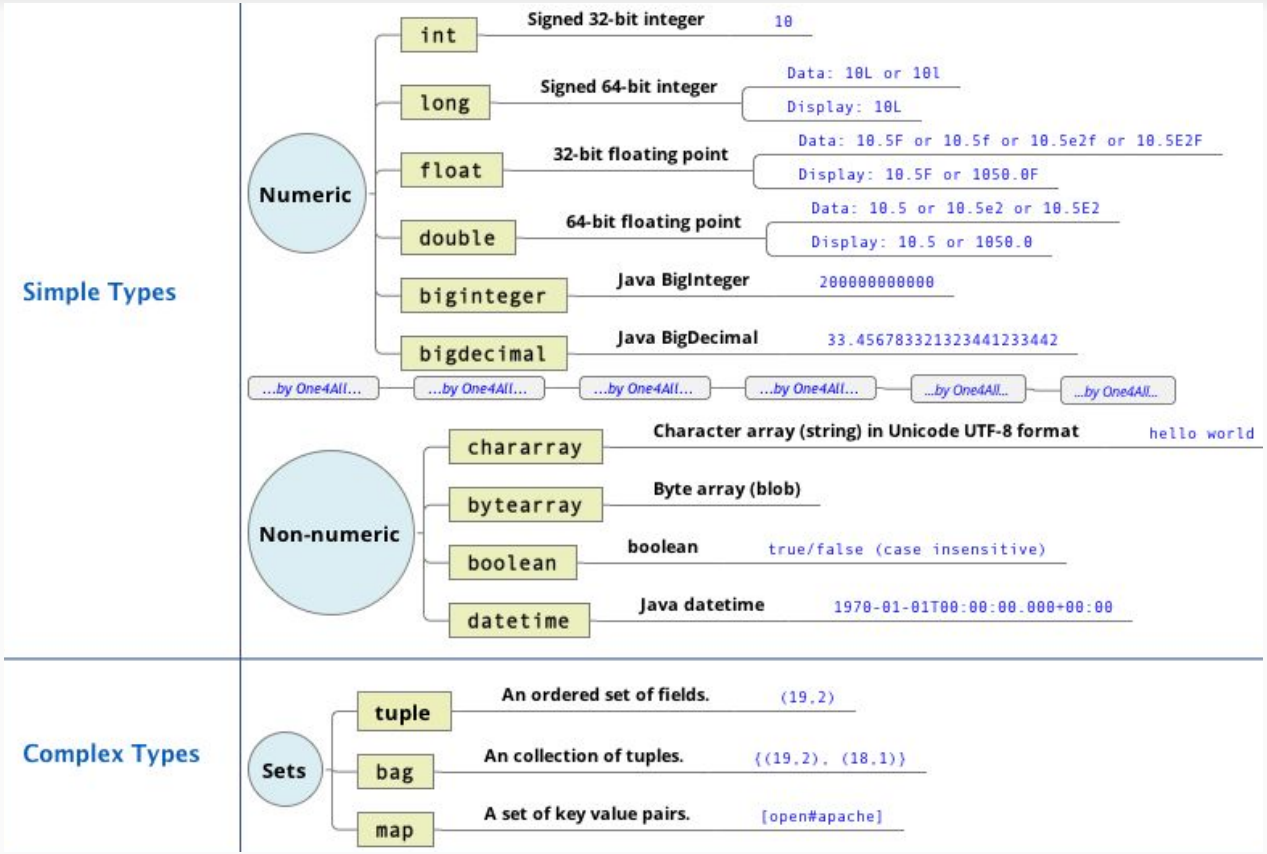
	Pig	Hive
Use case	ETL, data flow	Query processing
Language	Scripting language	SQL-like HiveQL
Best for	Unstructured and semi-structured data	Structured and semi-structured data
Programming efforts	May be difficult on the first steps	Relatively easy

Apache Pig: Perfect for ETL



[link]

Pig Data Types



[link]

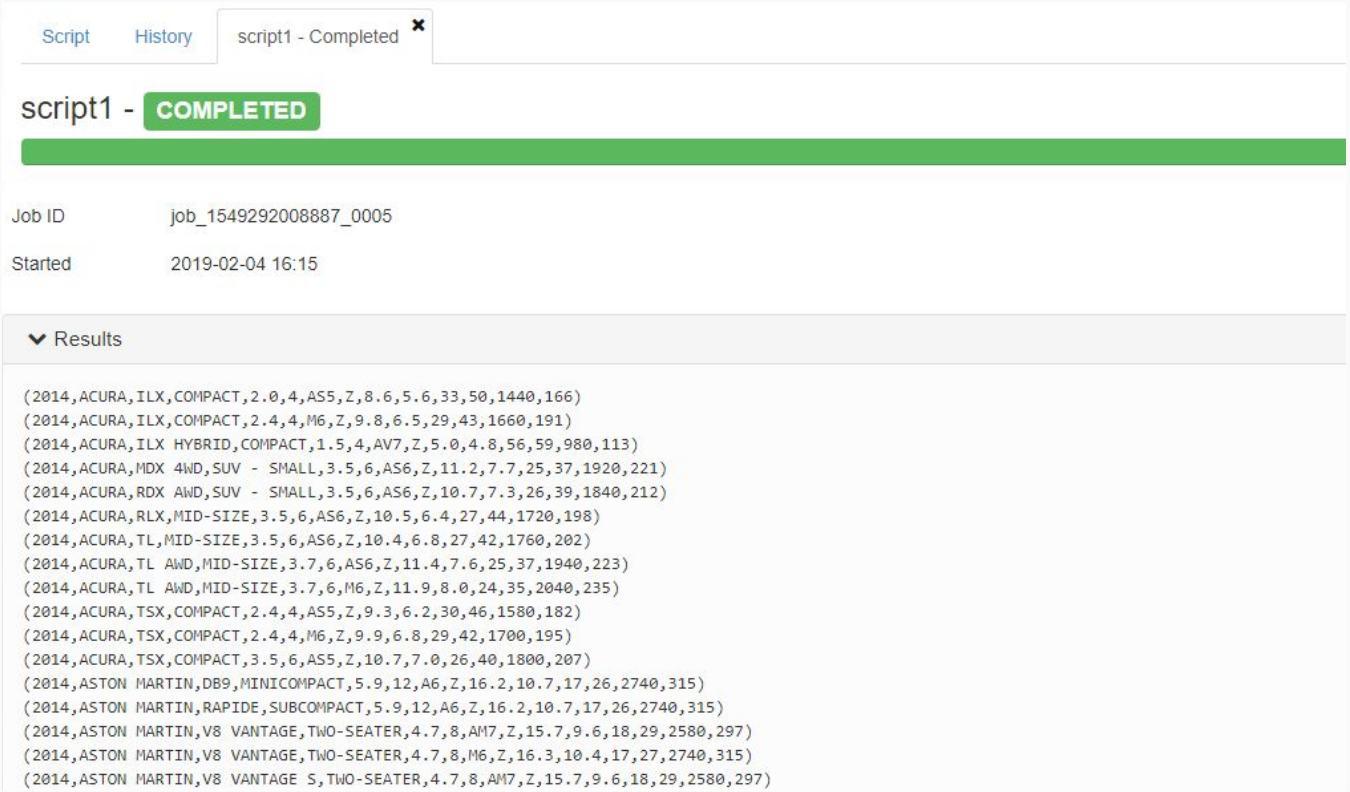
Pig Operations: load data

LOAD command is used to load data into **fuel** bag. Here we load data from file on HDFS

USING PigStorage(‘,’) is used to set delimiter

DUMP command just outputs contents of variable

Pig Operations: load data: output

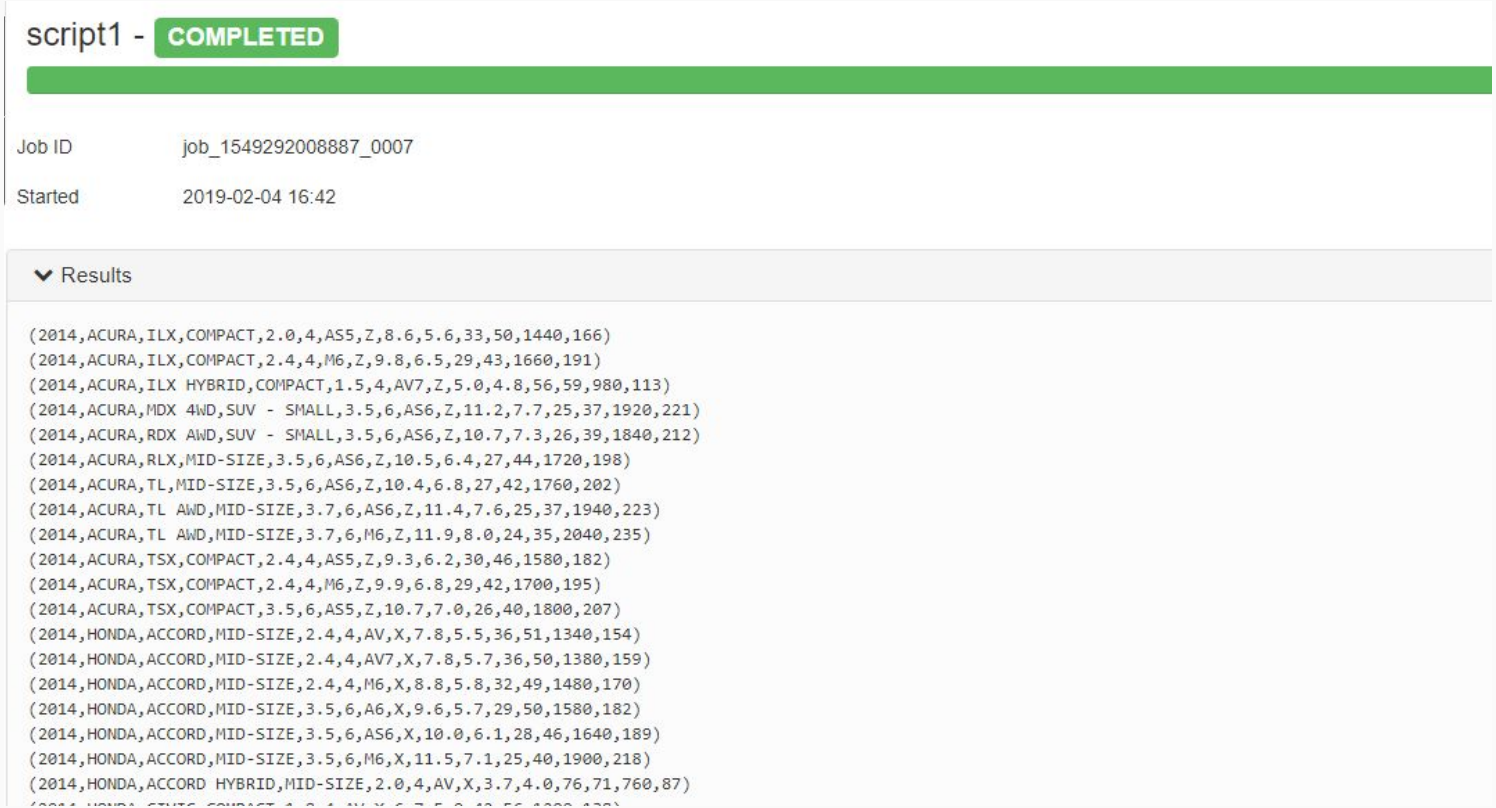


Pig Operations: filter data

FILTER command is used to **filter data**. Here we filter the second field ($1 -- fields are numbered from 0) with RegEx '(HONDA|ACURA)' and save result as fuel_ha.

Then, we output our result to console with DUMP command

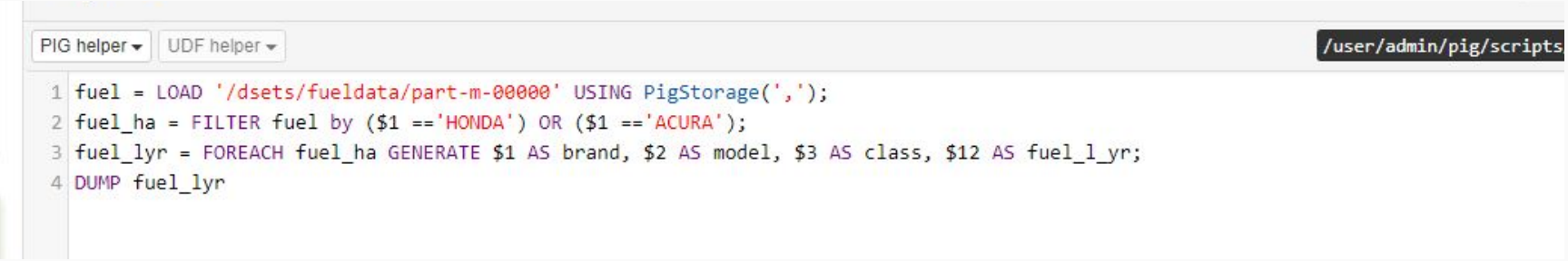
Pig Operations: filter data: output



Pig Operations: filter data: logical operators

You can use AND, OR, NOT logical operators to do some complex filtering

Pig Operations: selecting columns



Use FOREACH .. GENERATE to select the needed columns

Pig Operations: selecting columns: results

script1 - COMPLETED

Job ID	job_1549292008887_0011
Started	2019-02-04 16:59

Results

```
(ACURA,ILX,COMPACT,1440)
(ACURA,ILX,COMPACT,1660)
(ACURA,ILX HYBRID,COMPACT,980)
(ACURA,MDX 4WD,SUV - SMALL,1920)
(ACURA,RDX AWD,SUV - SMALL,1840)
(ACURA,RLX,MID-SIZE,1720)
(ACURA,TL,MID-SIZE,1760)
(ACURA,TL AWD,MID-SIZE,1940)
(ACURA,TL AWD,MID-SIZE,2040)
(ACURA,TSX,COMPACT,1580)
(ACURA,TSX,COMPACT,1700)
(ACURA,TSX,COMPACT,1800)
(HONDA,ACCORD,MID-SIZE,1340)
(HONDA,ACCORD,MID-SIZE,1380)
(HONDA,ACCORD,MID-SIZE,1480)
(HONDA,ACCORD,MID-SIZE,1580)
(HONDA,ACCORD,MID-SIZE,1640)
(HONDA,ACCORD,MID-SIZE,1900)
(HONDA,ACCORD HYBRID,MID-SIZE,760)
(HONDA,CIVIC,COMPACT,1200)
```

Pig Operations: grouping

PIG helper ▾ UDF helper ▾ /user/admin/pig/scrip

```
1 fuel = LOAD '/dsets/fueldata/part-m-00000' USING PigStorage(',');
2 fuel_ha = FILTER fuel by ($1 =='HONDA') OR ($1 =='ACURA');
3 fuel_lyr = FOREACH fuel_ha GENERATE $1 AS brand, $2 AS model, $3 AS class, $12 AS fuel_l_yr;
4 fuel_grp = GROUP fuel_lyr BY brand;
5 DUMP fuel_grp
```

You can use AND, OR, NOT logical operators to do some complex filtering

Pig Operations: grouping

Results

```
(ACURA,{(ACURA,ILX,COMPACT,1660),(ACURA,ILX HYBRID,COMPACT,980),(ACURA,MDX 4WD,SUV - SMALL,1920),(ACURA,RDX AWD,SUV - SMALL,1840),(AC
(HONDA,{(HONDA,ACCORD,MID-SIZE,1640),(HONDA,ACCORD,MID-SIZE,1340),(HONDA,ACCORD,MID-SIZE,1380),(HONDA,ACCORD,MID-SIZE,1480),(HONDA,AC
```

Pig Operations: grouping: Describe

PIG helper ▾ UDF helper ▾ /user/admin/pig/scripts/script

```
1 fuel = LOAD '/dsets/fueldata/part-m-00000' USING PigStorage(',');
2 fuel_ha = FILTER fuel by ($1 =='HONDA') OR ($1 =='ACURA');
3 fuel_lyr = FOREACH fuel_ha GENERATE $1 AS brand, $2 AS model, $3 AS class, $12 AS fuel_l_yr;
4 fuel_grp = GROUP fuel_lyr BY brand;
5
6 Describe fuel_grp;
```

▾ Results

```
fuel_grp: {group: bytearray,fuel_lyr: {(brand: bytearray,model: bytearray,class: bytearray,fuel_l_yr: bytearray)}}
```

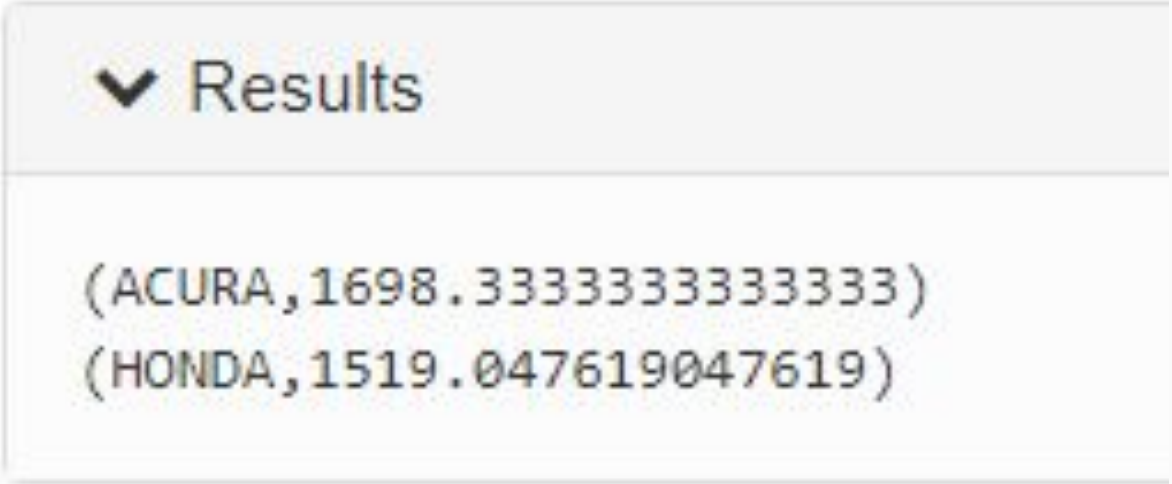
Pig Operations: aggregating

PIG helper ▾ UDF helper ▾ /user/admin/pig/scripts

```
1 fuel = LOAD '/dsets/fueldata/part-m-00000' USING PigStorage(',');
2 fuel_ha = FILTER fuel by ($1 =='HONDA') OR ($1 =='ACURA');
3 fuel_lyr = FOREACH fuel_ha GENERATE $1 AS brand, $2 AS model, $3 AS class, $12 AS fuel_l_yr;
4 fuel_grp = GROUP fuel_lyr BY brand;
5
6 fuel_avg = FOREACH fuel_grp GENERATE group as brand, AVG(fuel_lyr.fuel_l_yr);
7
8 DUMP fuel_avg
```

You can use AND, OR, NOT logical operators to do some complex filtering

Pig Operations: aggregating: results



Pig Operations: storing results

script1

PIG helper ▾ UDF helper ▾

```
1 fuel = LOAD '/dsets/fueldata/part-m-00000' USING PigStorage(',');
2 fuel_ha = FILTER fuel by ($1 =='HONDA') OR ($1 =='ACURA');
3 fuel_lyr = FOREACH fuel_ha GENERATE $1 AS brand, $2 AS model, $3 AS class, $12 AS fuel_l_yr;
4 fuel_grp = GROUP fuel_lyr BY brand;
5
6 fuel_avg = FOREACH fuel_grp GENERATE group as brand, AVG(fuel_lyr.fuel_l_yr);
7
8 --DUMP fuel_avg
9 STORE fuel_avg INTO '/dsets/fueldata/avg' USING PigStorage (',');
```

To save your results in HDFS use STORE .. INTO command

Pig Operations: storing results (cont-d)

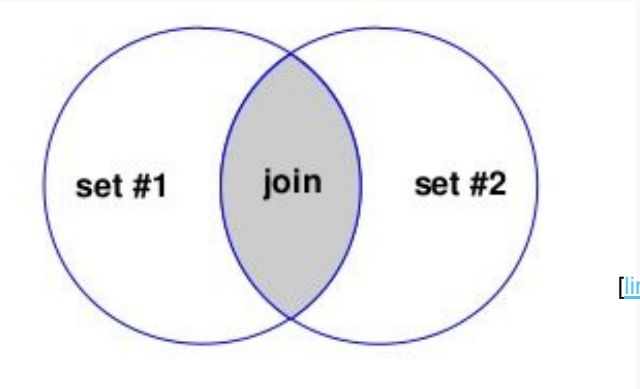
```
sandbox-hdp login: root
root@sandbox-hdp.hortonworks.com's password:
Last login: Thu Feb  7 01:50:52 2019 from 172.20.0.3
[root@sandbox-hdp ~]# hadoop fs -ls /dsets/fuel
ls: `/dsets/fuel': No such file or directory
[root@sandbox-hdp ~]# hadoop fs -ls /dsets/fueldata
Found 3 items
-rw-r--r--   1 root  hdfs          0 2019-02-01 14:58 /dsets/fueldata/_SUCCESS
drwxr-xr-x   - admin hdfs          0 2019-02-07 17:41 /dsets/fueldata/avg
-rw-r--r--   1 root  hdfs      76462 2019-02-01 14:58 /dsets/fueldata/part-m-00000
[root@sandbox-hdp ~]# hadoop fs -cat /dsets/fueldata/avg/*
ACURA,1698.3333333333333
HONDA,1519.047619047619
[root@sandbox-hdp ~]#
```

Pig Operations: JOIN

Default **JOIN** in Pig is **INNER JOIN (**rows are joined where keys match**).**

Syntax: **JOIN** set1 by *key1*, set2 by *key2*;

When **key1==key2** rows are joined into a new row



[link]

Pig supports also:

- Joining by multiple keys
- Outer joins (left/right/full)

Exercise (based on data from Hortonworks)

- Download a dataset http://wolly.cs.smu.ca/tmp/drivers.csv
- Download a dataset http://wolly.cs.smu.ca/tmp/timesheet.csv
- Put both files in "**/dsets/trucks**" directory in HDFS
- Download a script http://wolly.cs.smu.ca/tmp/ex1.txt
- Run it with Pig, try to understand each step
- Modify it:
  - The resulting **join_data** bag should also contain the "**certified**" column from the drivers.csv dataset
  - The result should be stored in **/dsets/trucks/result**

Pig with unstructured data: letters count

unstructured_1

PIG helper ▾ UDF helper ▾ /user/admin

```
1 text = LOAD '/dsets/textdata' USING TextLoader();
2 chars = FOREACH text GENERATE FLATTEN(TOKENIZE(REPLACE($0,'',','), ',')) as char;
3 letters = FILTER chars BY char matches '[A-Za-z]';
4 letters_grouped = GROUP letters BY char;
5 letters_counted = FOREACH letters_grouped GENERATE group as letter, COUNT(letters.char) as cnt;
6 letters_counted_ordered = ORDER letters_counted BY cnt DESC;
7
8 DUMP letters_counted_ordered
```

Pig with unstructured data

```
(e,628234)
(t,444459)
(a,395872)
(o,382683)
(n,362397)
(i,348464)
(s,326238)
(r,303977)
(h,287323)
(d,211677)
(l,195904)
(c,138853)
(u,137114)
(m,120829)
(f,116374)
(w,94576)
(g,93732)
(p,89967)
(y,88196)
(b,67206)
(v,50525)
(k,31160)
(I,17174)
(T,16282)
(A,12217)
(x,9196)
(P,8946)
(S,8659)
(H,7358)
```

Exercise

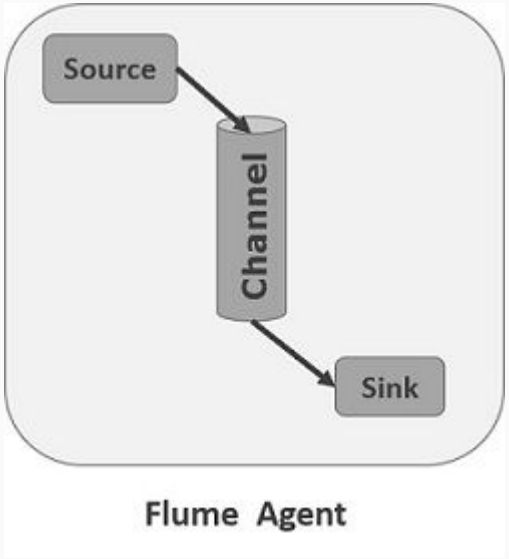
- Download a text data from http://wolly.cs.smu.ca/tmp/big.txt and put to '**/dsets/textdata**' folder in HDFS
- Download a script http://wolly.cs.smu.ca/tmp/ex2.txt
- Run it with Pig
- *You may notice that it shows letter count for both lowercase and uppercase letters (e.g., (I,17174) and (i,348464))*
- **You should modify the script so that it will show letter count regardless of case (e.g., (I, 365638)). What are the most used 5 letters? 5 least used?**

Flume

Flume is a software for efficiently collecting, aggregating, and moving large amounts of **log data (and other data)**

Primary use of flume: **get** data from various sources (logs, Tweets, web applications, etc.) and **store** in Hadoop ecosystem

Flume: Architecture



[link]

Flume instances are called **Agents**

You can have many of them.

Each agent should have:

- **Source**: where to get data from? (logs, tweets, web, etc.). Units of data transported into Flume are called **Events**
- **Channel**: how to *efficiently and correctly* transfer data from source to sink?
- **Sink**: where to put data? (HDFS, local file system, or maybe HBase?)

Additional components: interceptors, selectors, serializers, sink processors (out of scope of the workshop)

Flume: Sources

- **avro**: Avro Netty RPC event source. Listens on Avro port and receives events from external Avro streams (usually used for **communicating between flume agents**)
- **netcat**: Remember NetCat from tutorials? Listens on a given port and turns each line of text into an event
- **exec**: Execute a Unix process and read from stdout
- **syslogtcp/syslogudp**: reads syslog data and generates flume events
- **http**: accepts Flume events by HTTP POST and GET
- **spooldir**: ingesting data by placing files to be ingested into a directory on disk
- Other: Kafka, Twitter, etc.
- You may write your own as well

Flume: Channels

- **memory**: in-memory queue (high speed)
- **file:** events are stored in file
- **JDBC**: events are stored in the database (high durability)
- **kafka**: (Kafka must be installed)
- **etc**.
- You may write your own as well

Flume: Sinks

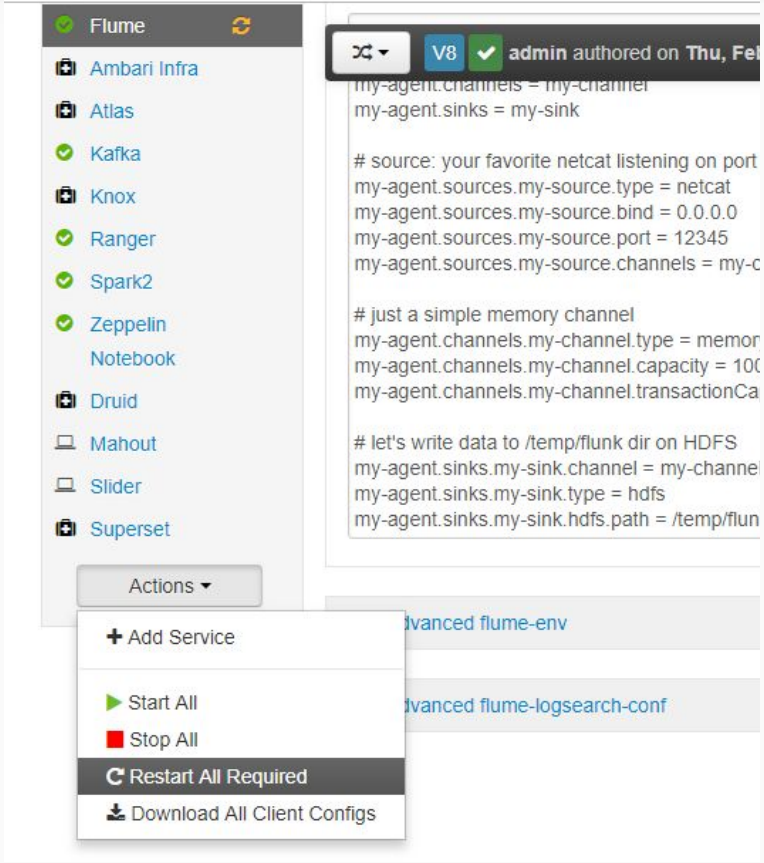
- **avro**: invokes a pre-defined Avro (RPC) protocol method for all events
- **hdfs**: stores data in HDFS
- **logger**: just logs received data in log
- **hbase**: stores data in HBase
- **kafka**: send data to Kafka
- You may write your own as well

Flume: A Simple Agent

```
# let's define an agent with name 'my-agent' '
my-agent.sources = my-source
my-agent.channels = my-channel
my-agent.sinks = my-sink

# source: your favorite netcat listening on port 12345
my-agent.sources.my-source.type = netcat
my-agent.sources.my-source.bind = 0.0.0.0
my-agent.sources.my-source.port = 12345
my-agent.sources.my-source.channels = my-channel

# just a simple memory channel
my-agent.channels.my-channel.type = memory
my-agent.channels.my-channel.capacity = 10000

|
# let's write data to /temp/flunk dir on HDFS
my-agent.sinks.my-sink.channel = my-channel
my-agent.sinks.my-sink.type = hdfs
my-agent.sinks.my-sink.hdfs.path = /temp/flunk
my-agent.sinks.my-sink.hdfs.fileType = DataStream
my-agent.sinks.my-sink.hdfs.writeFormat = Text
```

Sorry, no GUI: configuration of agents is text-based.

Flume: A Simple Agent (cont-d)



Restart Flume after configuration is changed

Flume: A Simple Agent (cont-d)

Let’s see how it works!

```
[root@sandbox-hdp ~]#
[root@sandbox-hdp ~]# echo "hello" | nc 127.0.0.1 12345
OK
[root@sandbox-hdp ~]# echo "hello there" | nc 127.0.0.1 12345
OK
[root@sandbox-hdp ~]# echo "hello there\n" | nc 127.0.0.1 12345
OK
[root@sandbox-hdp ~]# hadoop fs -ls /temp/flunk
Found 1 items
-rw-r--r--   1 flume hdfs         32 2019-02-07 02:01 /temp/flunk/FlumeData.1549504851743
[root@sandbox-hdp ~]# hadoop fs -cat /temp/flunk/*
hello
hello there
hello there\n
[root@sandbox-hdp ~]#
```

Flume: A Simple Agent: Explained

```
# let's define an agent with name 'my-agent' ' #just a comment
#'my-source' is just the name of Source. You may choose your own
my-agent.sources = my-source
 #'my-channel' is just the name of Channel. You may choose your own
my-agent.channels = my-channel
#'my-sink' is just the name of Sink. You may choose your own
my-agent.sinks = my-sink
# source: your favorite netcat listening on port 12345 #just a comment: now we define a Source
 #type of source is netcat
my-agent.sources.my-source.type = netcat
#netcat-specific: bind address
my-agent.sources.my-source.bind = 0.0.0.0
#netcat-specific: port (which port to listen on?)
my-agent.sources.my-source.port = 12345
#what channel to use to put "Events"?
my-agent.sources.my-source.channels = my-channel

# just a simple memory channel #just a comment: now we define a Channel
#use the simplest channel: memory
my-agent.channels.my-channel.type = memory
#100 is max number of events in the queue
my-agent.channels.my-channel.capacity = 100
```

Flume: A Simple Agent: Explained (cont-d)

```
# let's write data to /temp/flunk dir on HDFS   #just a comment: now we define a Sink
#from which channel to get data?
my-agent.sinks.my-sink.channel = my-channel
#type of sink is HDFS
my-agent.sinks.my-sink.type = hdfs
#HDFS-specific: path on HDFS where to write data
my-agent.sinks.my-sink.hdfs.path = /temp/flunk
#HDFS-specific: default is SequenceFile
my-agent.sinks.my-sink.hdfs.fileType = DataStream
#HDFS-specific: write as a plain text
my-agent.sinks.my-sink.hdfs.writeFormat = Text
```

Q: Where to get all these parameters?

A: Here: https://flume.apache.org/FlumeUserGuide.html

Exercise (may work in groups)

- Set up a flume agent as in the example. Test it

(http://wolly.cs.smu.ca/tmp/flume.txt)

- Change either source or sink to some other type and see if it works

EOW

EOW

Thanks for your attention!