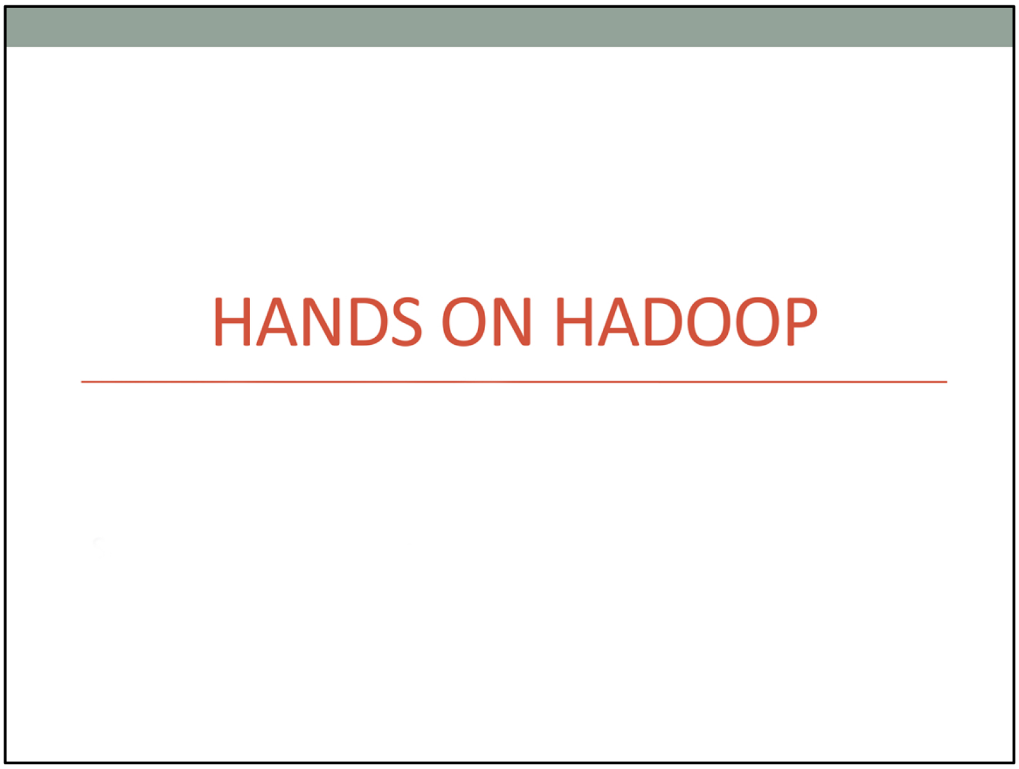
Transforming Data
With Intelligence™

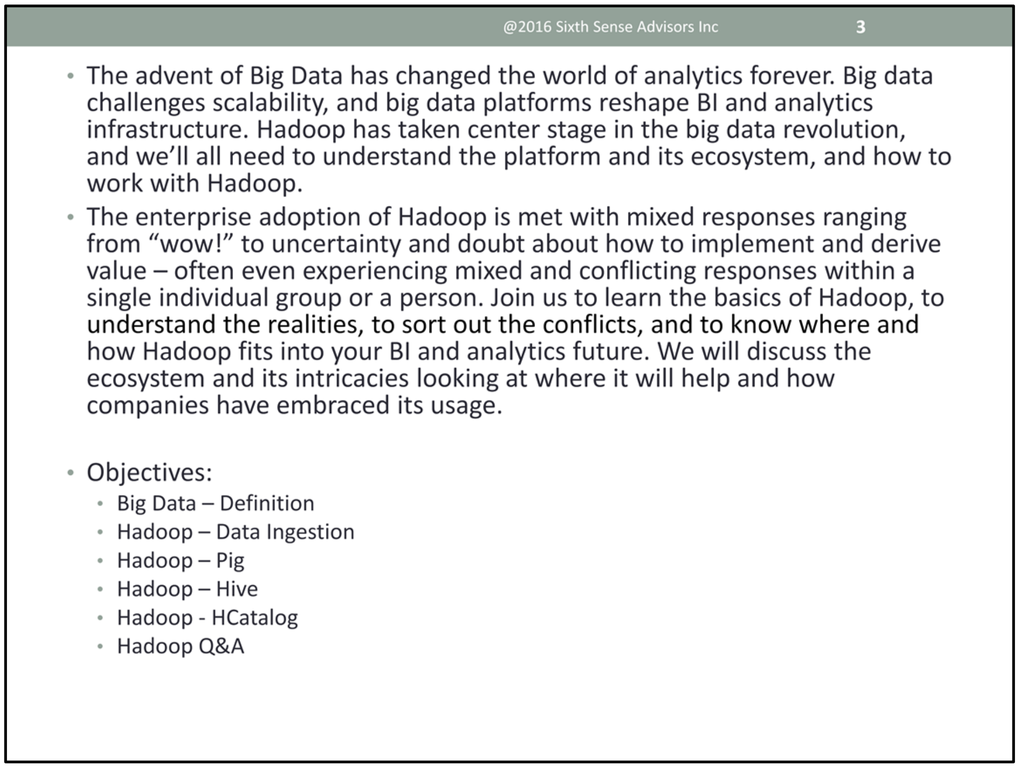
Previews of TDWI course books offer an opportunity to see the quality of our material and help you to select the courses that best fit your needs. The previews cannot be printed.

TDWI strives to provide course books that are content-rich and that serve as useful reference documents after a class has ended.

This preview shows selected pages that are representative of the entire course book; pages are not consecutive. The page numbers shown at the bottom of each page indicate their actual position in the course book. All table-of-contents pages are included to illustrate all of the topics covered by the course.



HANDS ON HADOOP

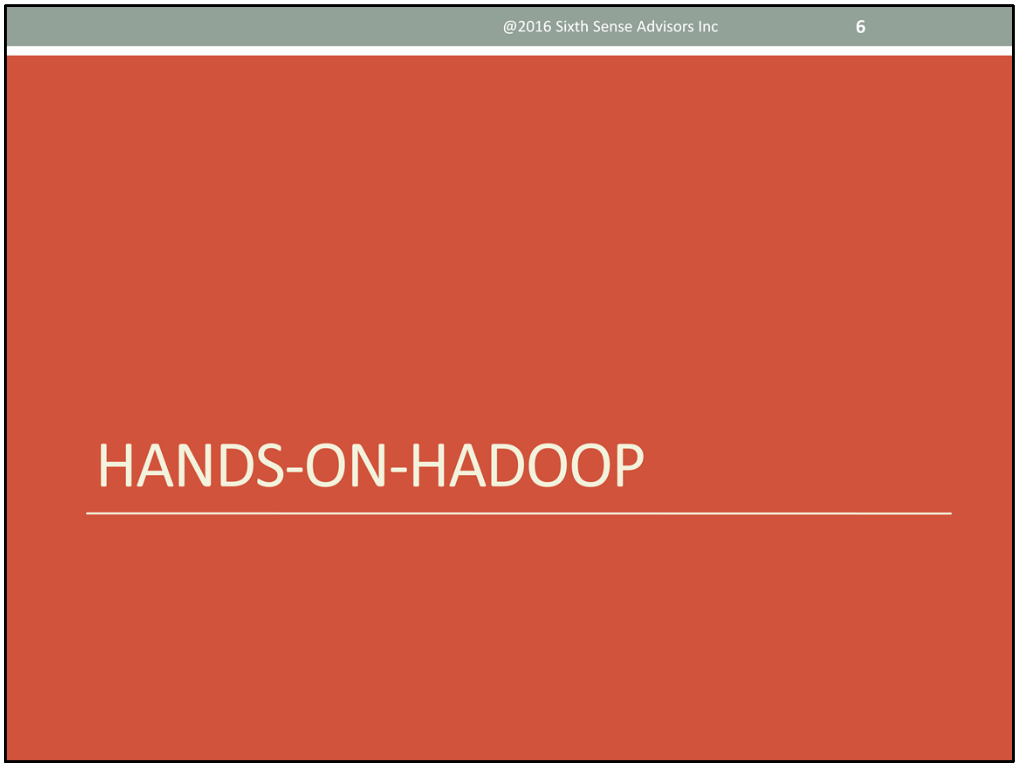


- The advent of Big Data has changed the world of analytics forever. Big data challenges scalability, and big data platforms reshape BI and analytics infrastructure. Hadoop has taken center stage in the big data revolution, and we'll all need to understand the platform and its ecosystem, and how to work with Hadoop.
- The enterprise adoption of Hadoop is met with mixed responses ranging from "wow!" to uncertainty and doubt about how to implement and derive value – often even experiencing mixed and conflicting responses within a single individual group or a person. Join us to learn the basics of Hadoop, to understand the realities, to sort out the conflicts, and to know where and how Hadoop fits into your BI and analytics future. We will discuss the ecosystem and its intricacies looking at where it will help and how companies have embraced its usage.

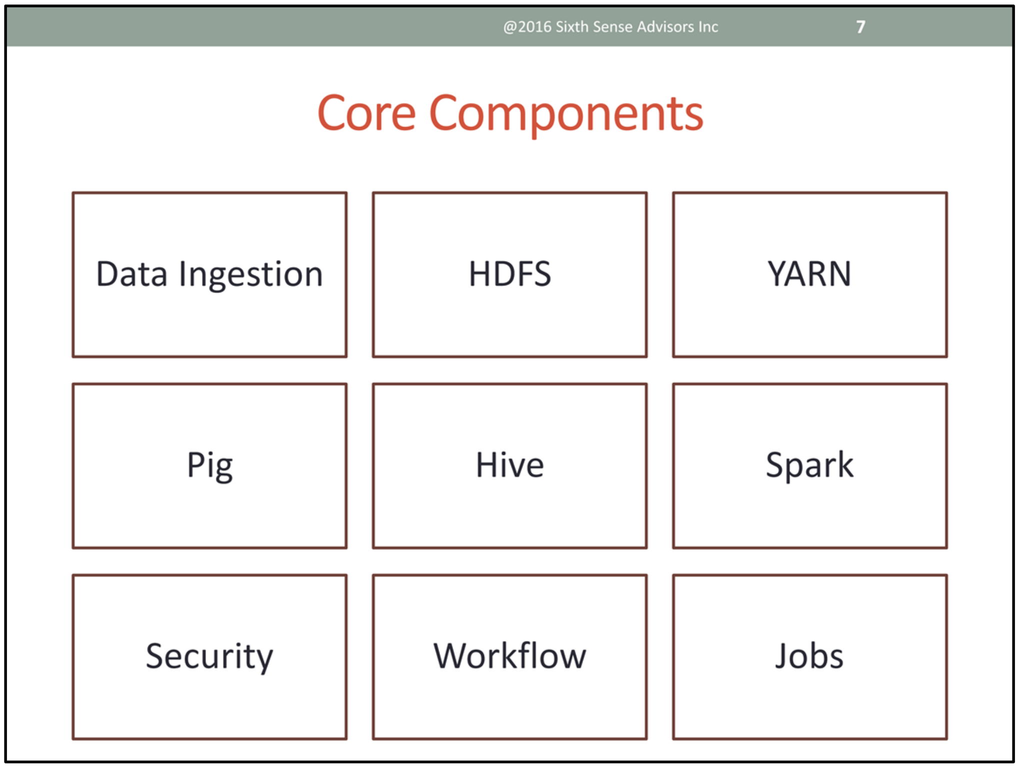
- Objectives:
  - Big Data – Definition
  - Hadoop – Data Ingestion
  - Hadoop – Pig
  - Hadoop – Hive
  - Hadoop - HCatalog
  - Hadoop Q&A

What disrupted the data center?

Data



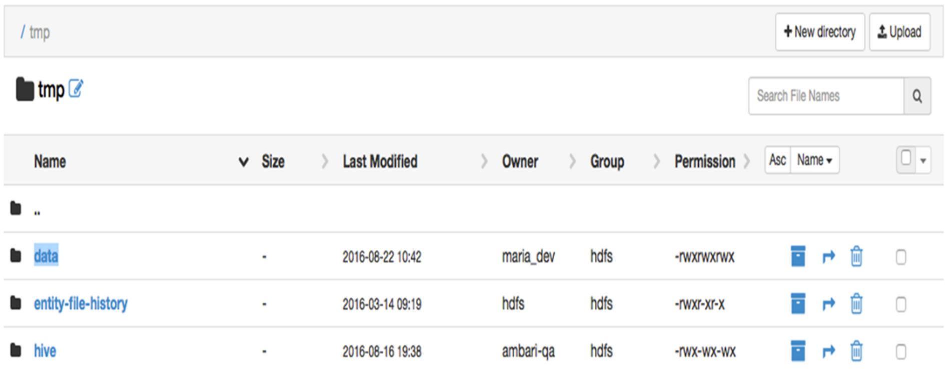
HANDS-ON-HADOOP



Core Components

Data Ingestion	HDFS	YARN
Pig	Hive	Spark
Security	Workflow	Jobs

HDFS Browser


/ tmp

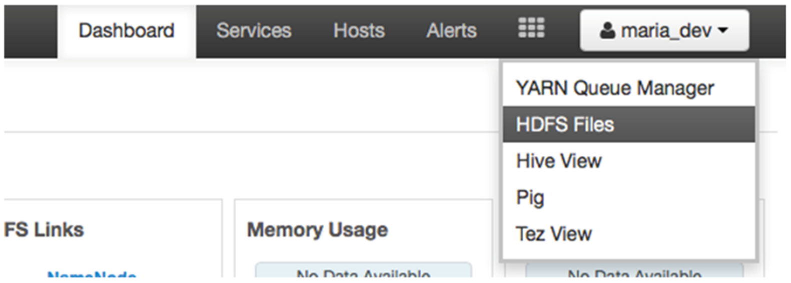
New directory | Upload

tmp

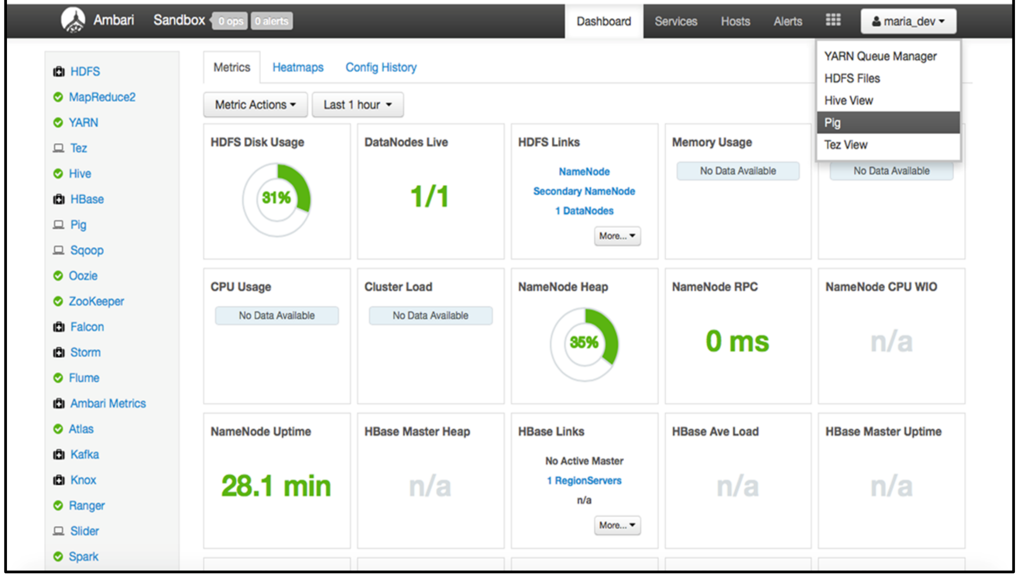
Search File Names

Name	Size	Last Modified	Owner	Group	Permission
..					
data	-	2016-08-22 10:42	maria_dev	hdfs	-rwxrwxrwx
entity-file-history	-	2016-03-14 09:19	hdfs	hdfs	-rwxr-xr-x
hive	-	2016-08-16 19:38	ambari-qa	hdfs	-rwx-wx-wx

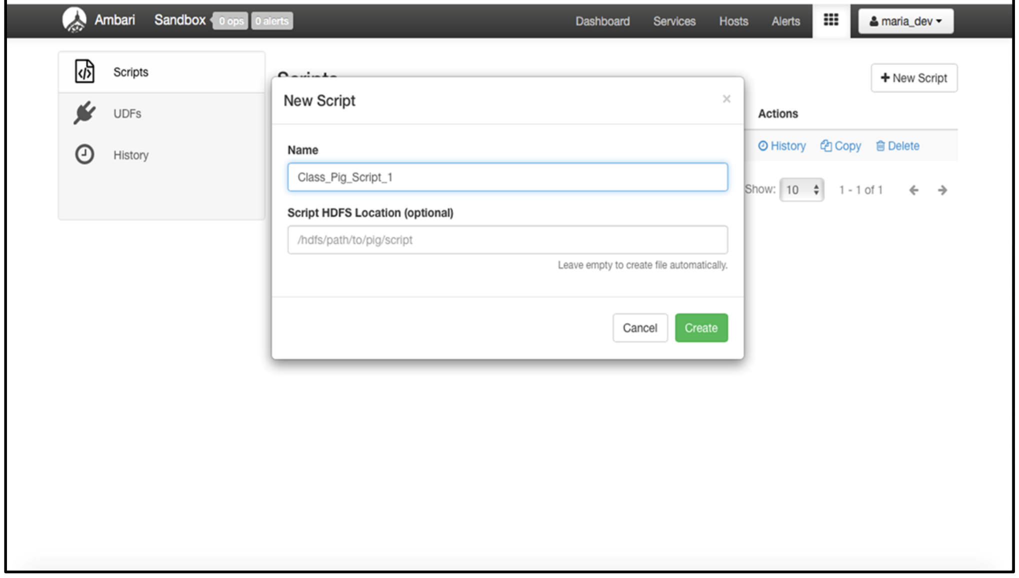
Launch HDFS File Browser



Pig UI Launch



Create New Script



New Script

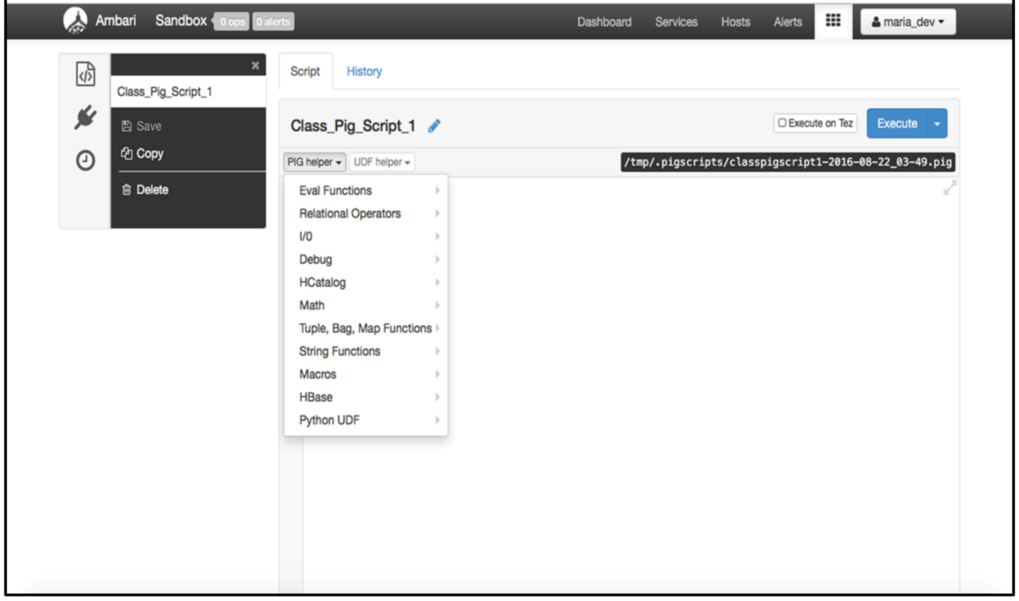
Name: Class_Pig_Script_1

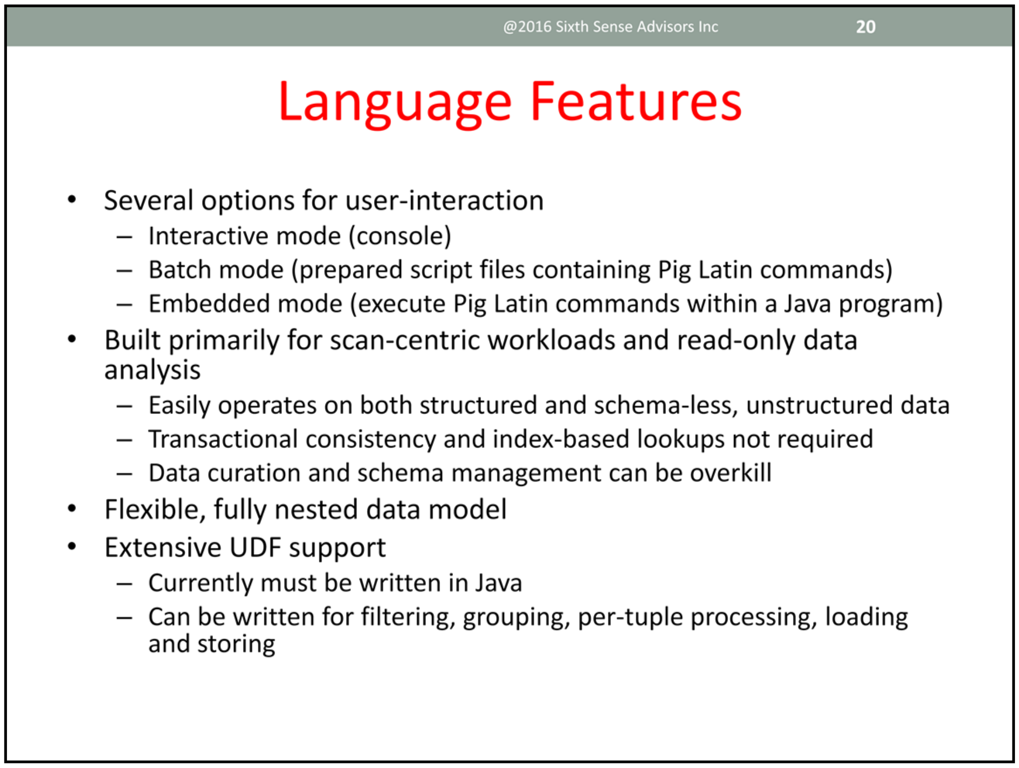
Script HDFS Location (optional): /hdfs/path/to/pig/script

Leave empty to create file automatically.

Cancel Create

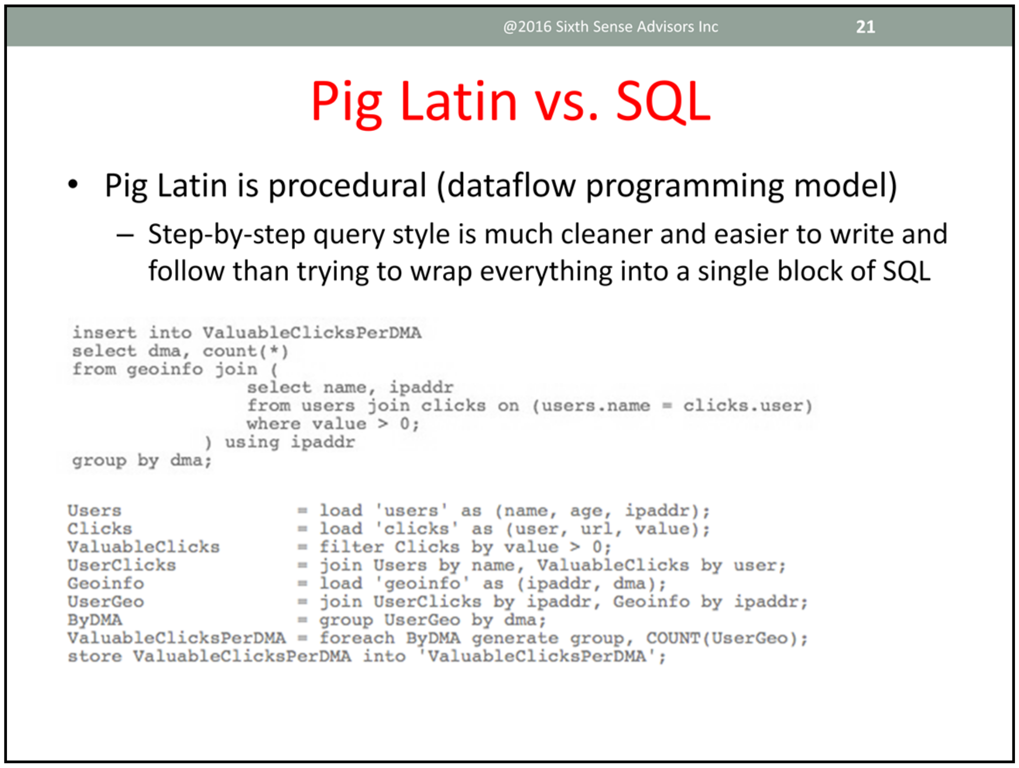
Pig Functions Help





Language Features

- Several options for user-interaction
  - Interactive mode (console)
  - Batch mode (prepared script files containing Pig Latin commands)
  - Embedded mode (execute Pig Latin commands within a Java program)
- Built primarily for scan-centric workloads and read-only data analysis
  - Easily operates on both structured and schema-less, unstructured data
  - Transactional consistency and index-based lookups not required
  - Data curation and schema management can be overkill
- Flexible, fully nested data model
- Extensive UDF support
  - Currently must be written in Java
  - Can be written for filtering, grouping, per-tuple processing, loading and storing



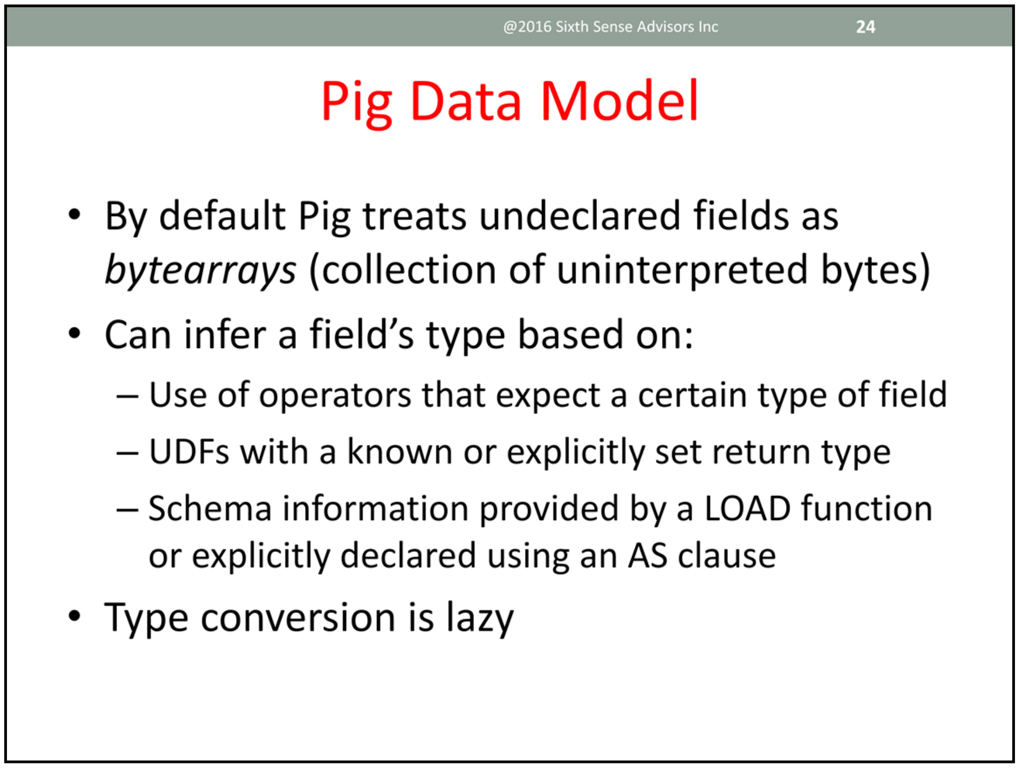
Pig Latin vs. SQL

- Pig Latin is procedural (dataflow programming model)
  - Step-by-step query style is much cleaner and easier to write and follow than trying to wrap everything into a single block of SQL

```
insert into ValuableClicksPerDMA
select dma, count(*)
from geoinfo join (
                select name, ipaddr
                from users join clicks on (users.name = clicks.user)
                where value > 0;
            ) using ipaddr
group by dma;
```

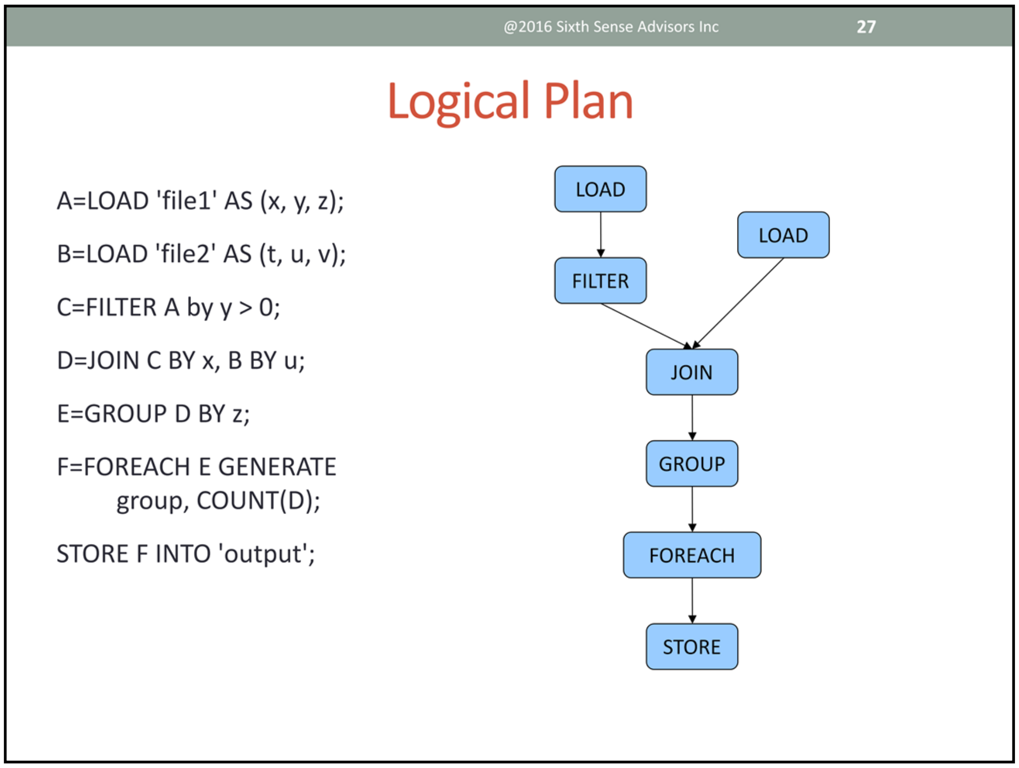
```
Users                = load 'users' as (name, age, ipaddr);
Clicks               = load 'clicks' as (user, url, value);
ValuableClicks       = filter Clicks by value > 0;
UserClicks           = join Users by name, ValuableClicks by user;
Geoinfo              = load 'geoinfo' as (ipaddr, dma);
UserGeo              = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA                = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Source:
http://developer.yahoo.net/blogs/hadoop/2010/01/comparing_pig_latin_and_sql_fo.html



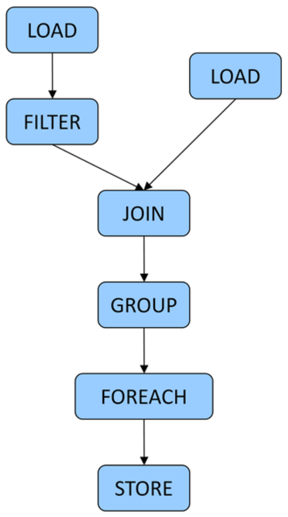
Pig Data Model

- By default Pig treats undeclared fields as *bytearrays* (collection of uninterpreted bytes)
- Can infer a field's type based on:
  - Use of operators that expect a certain type of field
  - UDFs with a known or explicitly set return type
  - Schema information provided by a LOAD function or explicitly declared using an AS clause
- Type conversion is lazy



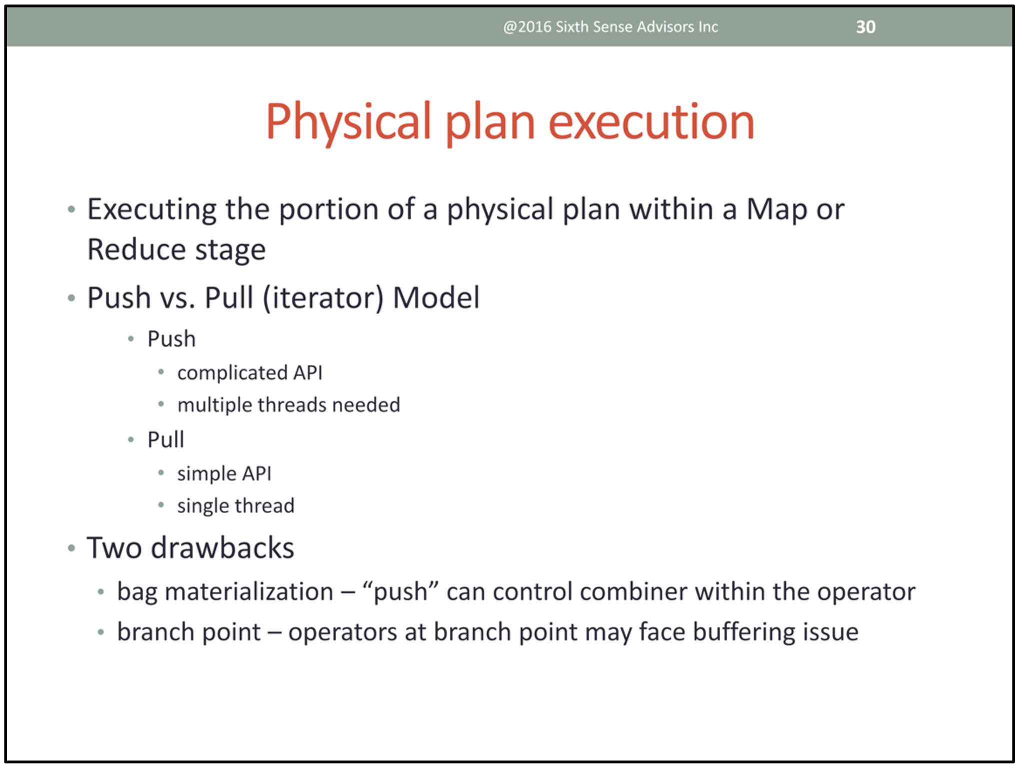
Logical Plan

```
A=LOAD 'file1' AS (x, y, z);

B=LOAD 'file2' AS (t, u, v);

C=FILTER A by y > 0;

D=JOIN C BY x, B BY u;

E=GROUP D BY z;

F=FOREACH E GENERATE
        group, COUNT(D);

STORE F INTO 'output';
```



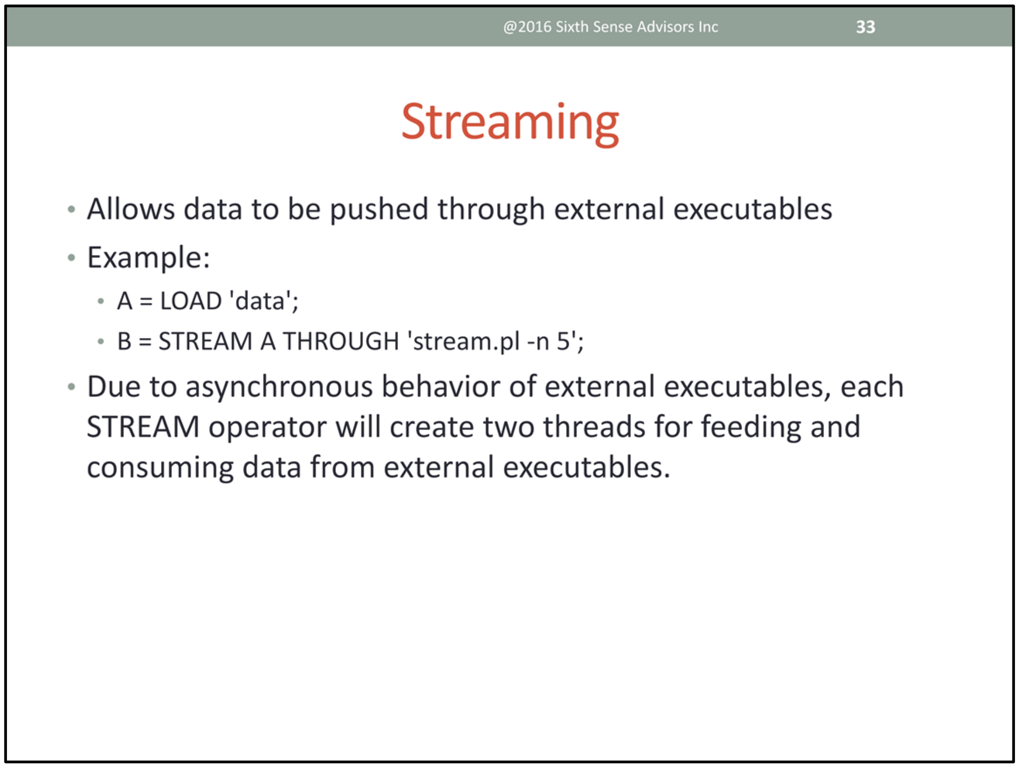
Physical Plan

- 1:1 correspondence with most logical operators
- Except for:
  - DISTINCT
  - (CO)GROUP
  - JOIN
  - ORDER



Physical plan execution

- Executing the portion of a physical plan within a Map or Reduce stage
- Push vs. Pull (iterator) Model
    - Push
        - complicated API
        - multiple threads needed
    - Pull
        - simple API
        - single thread
- Two drawbacks
    - bag materialization – "push" can control combiner within the operator
    - branch point – operators at branch point may face buffering issue



Streaming

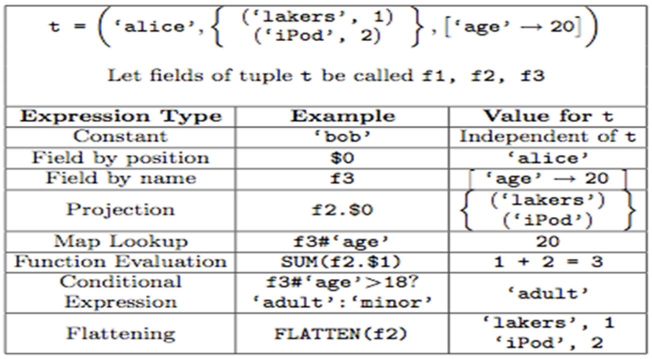
- Allows data to be pushed through external executables
- Example:
  - A = LOAD 'data';
  - B = STREAM A THROUGH 'stream.pl -n 5';
- Due to asynchronous behavior of external executables, each STREAM operator will create two threads for feeding and consuming data from external executables.

Pig Latin

- FOREACH-GENERATE (per-tuple processing)
  - Iterates over every input tuple in the bag, producing one output each, allowing efficient parallel implementation

```
expanded_queries = FOREACH queries GENERATE
                   userId, expandQuery(queryString);
```

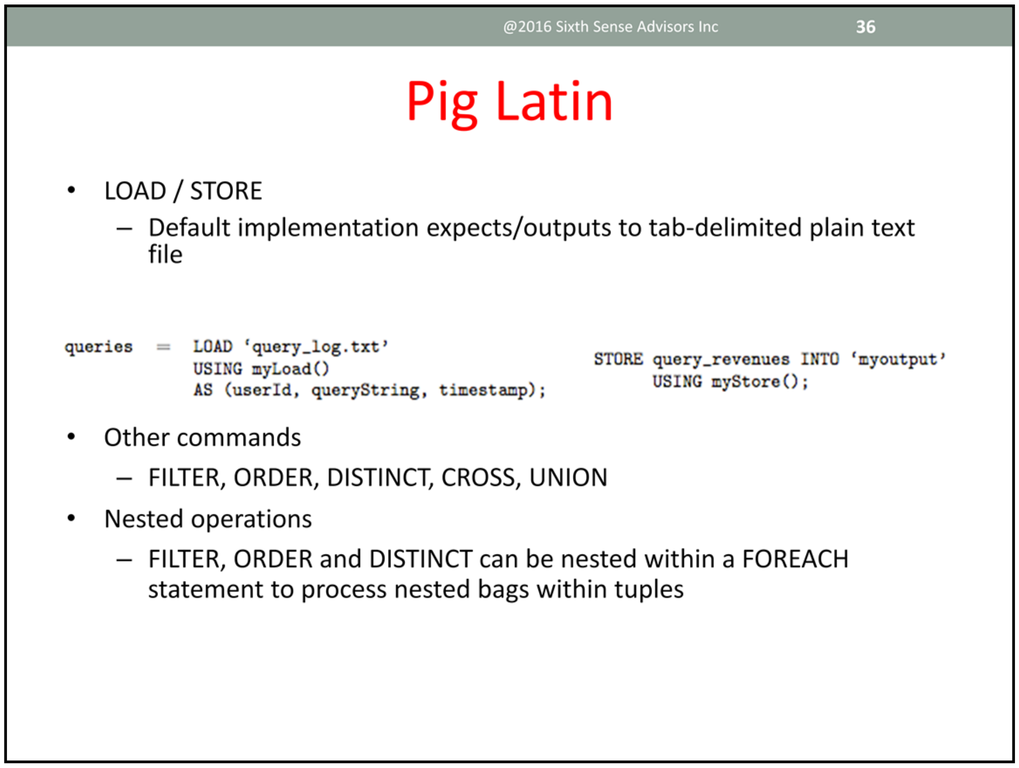
- Expressions within the GENERATE clause can take the form of the any of these expressions



$t = \left(\text{'alice'}, \left\{ \begin{matrix} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{matrix} \right\}, [\text{'age'} \rightarrow 20] \right)$

Let fields of tuple t be called f1, f2, f3

Expression Type	Example	Value for t
Constant	'bob'	Independent of t
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2



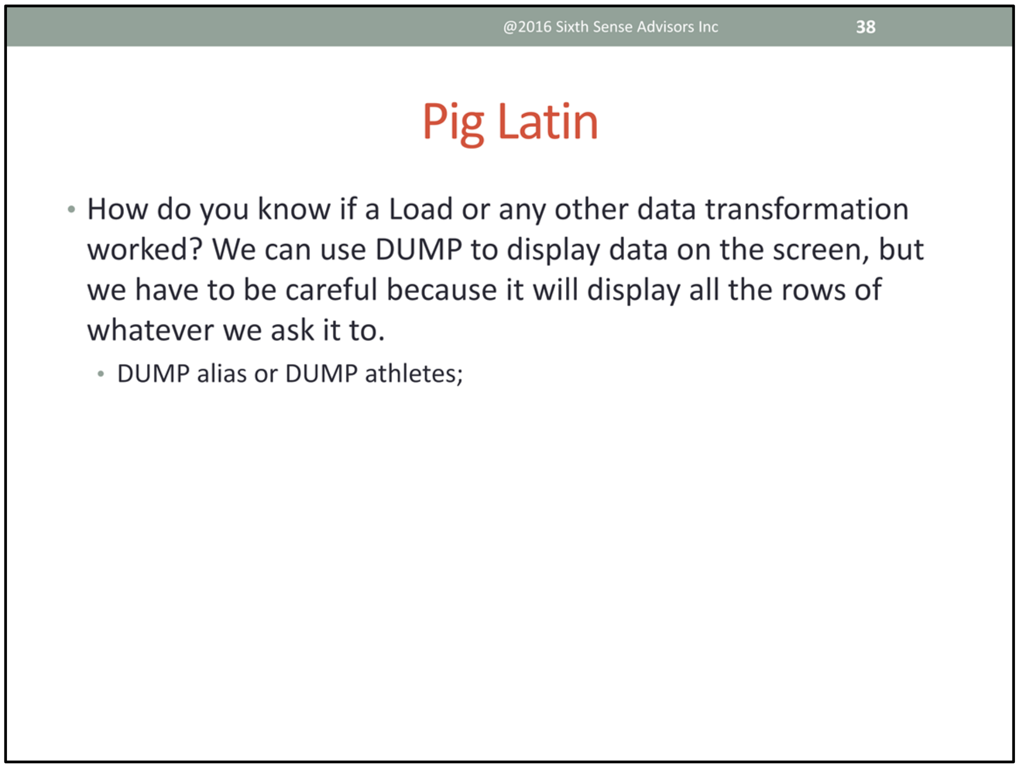
Pig Latin

- LOAD / STORE
  - Default implementation expects/outputs to tab-delimited plain text file

```
queries  =  LOAD 'query_log.txt'
            USING myLoad()
            AS (userId, queryString, timestamp);
```

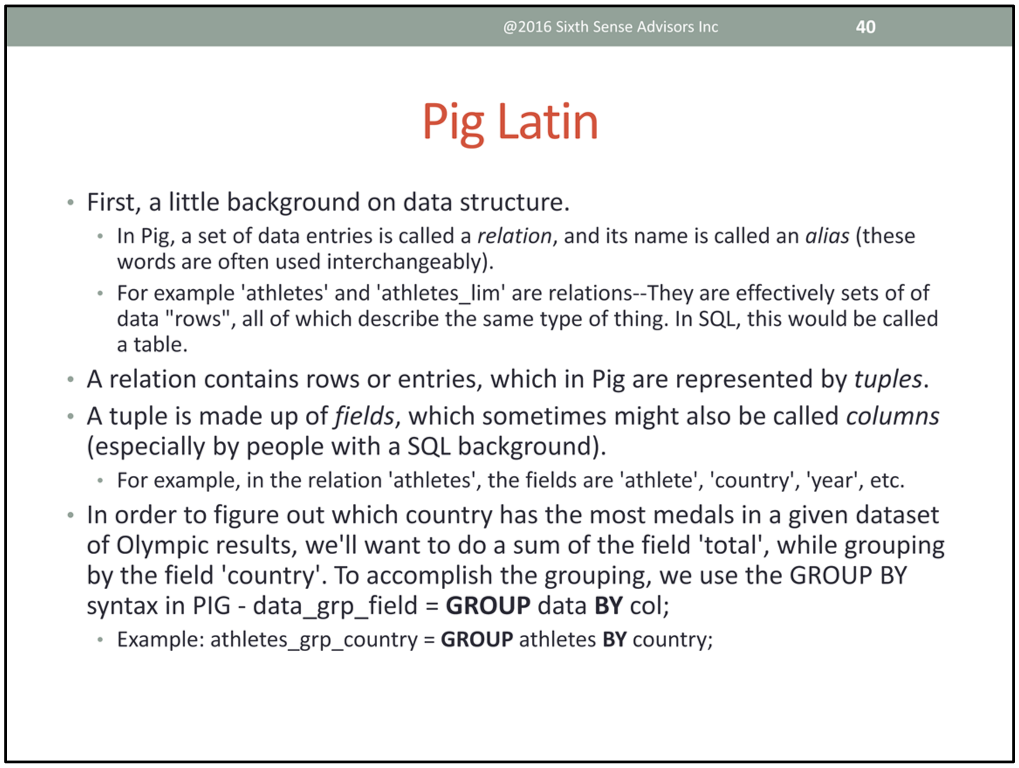
```
STORE query_revenues INTO 'myoutput'
      USING myStore();
```

- Other commands
  - FILTER, ORDER, DISTINCT, CROSS, UNION
- Nested operations
  - FILTER, ORDER and DISTINCT can be nested within a FOREACH statement to process nested bags within tuples



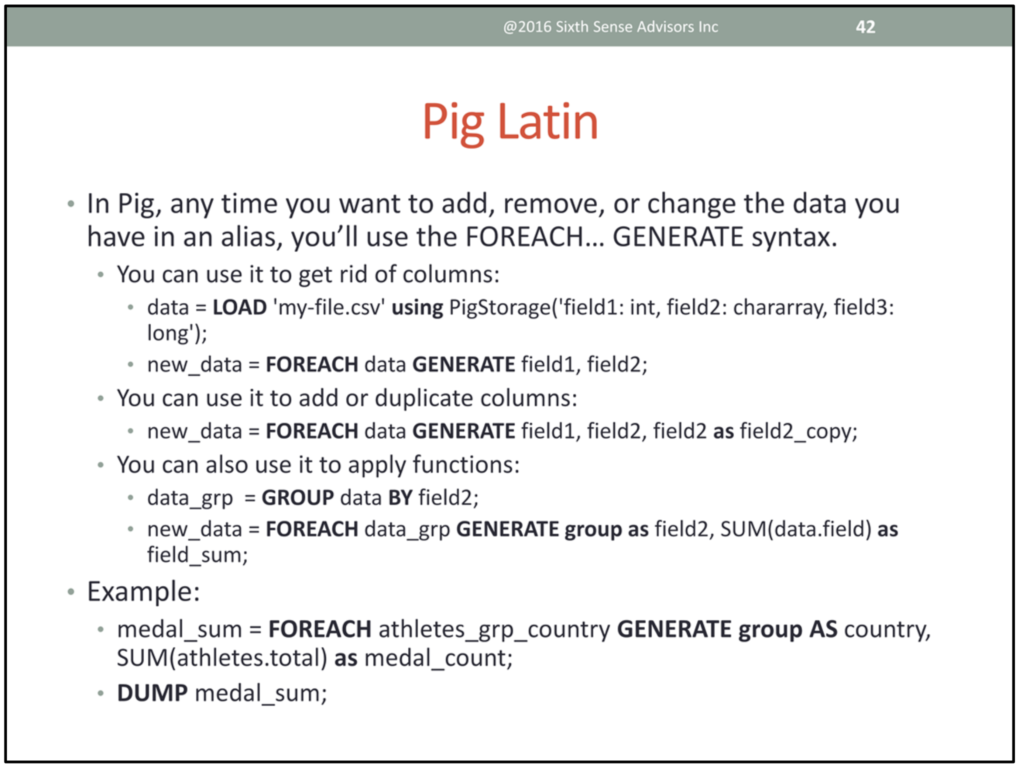
Pig Latin

- How do you know if a Load or any other data transformation worked? We can use DUMP to display data on the screen, but we have to be careful because it will display all the rows of whatever we ask it to.
  - DUMP alias or DUMP athletes;



Pig Latin

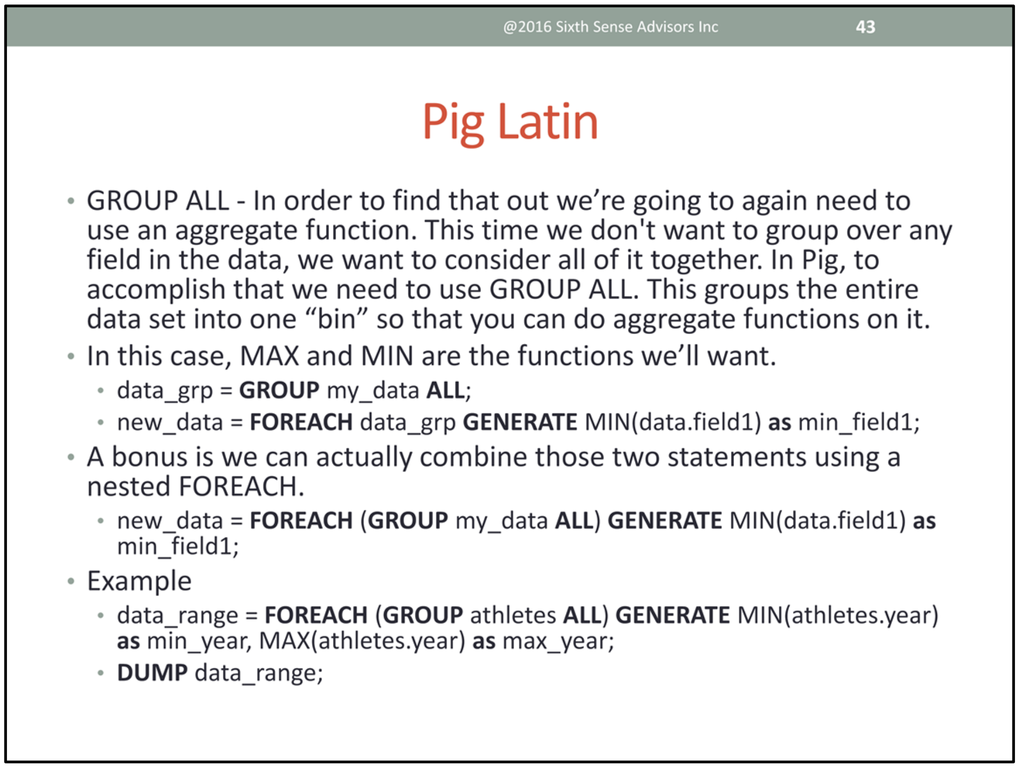
- First, a little background on data structure.
  - In Pig, a set of data entries is called a *relation*, and its name is called an *alias* (these words are often used interchangeably).
  - For example 'athletes' and 'athletes_lim' are relations--They are effectively sets of of data "rows", all of which describe the same type of thing. In SQL, this would be called a table.
- A relation contains rows or entries, which in Pig are represented by *tuples*.
- A tuple is made up of *fields*, which sometimes might also be called *columns* (especially by people with a SQL background).
  - For example, in the relation 'athletes', the fields are 'athlete', 'country', 'year', etc.
- In order to figure out which country has the most medals in a given dataset of Olympic results, we'll want to do a sum of the field 'total', while grouping by the field 'country'. To accomplish the grouping, we use the GROUP BY syntax in PIG - data_grp_field = **GROUP** data **BY** col;
  - Example: athletes_grp_country = **GROUP** athletes **BY** country;


@2016 Sixth Sense Advisors Inc 42

Pig Latin

- In Pig, any time you want to add, remove, or change the data you have in an alias, you'll use the FOREACH… GENERATE syntax.
  - You can use it to get rid of columns:
    - data = **LOAD** 'my-file.csv' **using** PigStorage('field1: int, field2: chararray, field3: long');
    - new_data = **FOREACH** data **GENERATE** field1, field2;
  - You can use it to add or duplicate columns:
    - new_data = **FOREACH** data **GENERATE** field1, field2, field2 **as** field2_copy;
  - You can also use it to apply functions:
    - data_grp = **GROUP** data **BY** field2;
    - new_data = **FOREACH** data_grp **GENERATE group as** field2, SUM(data.field) **as** field_sum;
- Example:
  - medal_sum = **FOREACH** athletes_grp_country **GENERATE group AS** country, SUM(athletes.total) **as** medal_count;
  - **DUMP** medal_sum;

We've grouped our data but we still don't have our answer. We need to use an aggregate function—that is to say, a function that will look at the data in a single field across all rows, and tell us something about it. In this case, the function called SUM will add up the number of medals.

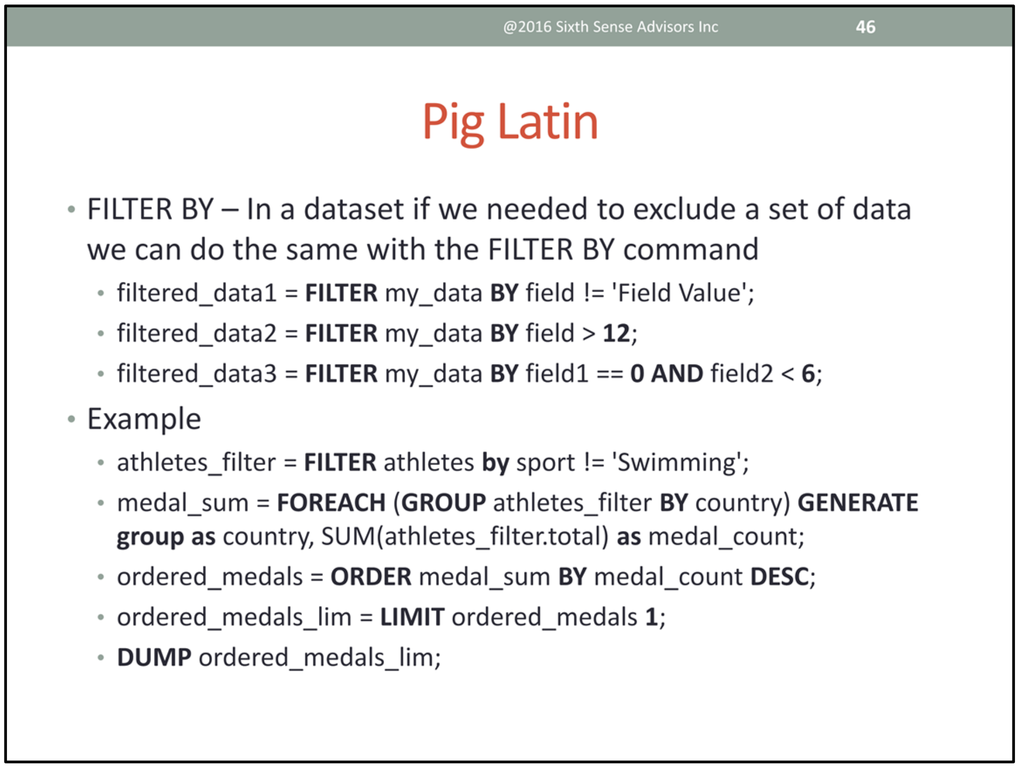


Pig Latin

- GROUP ALL - In order to find that out we're going to again need to use an aggregate function. This time we don't want to group over any field in the data, we want to consider all of it together. In Pig, to accomplish that we need to use GROUP ALL. This groups the entire data set into one "bin" so that you can do aggregate functions on it.
- In this case, MAX and MIN are the functions we'll want.
  - data_grp = **GROUP** my_data **ALL**;
  - new_data = **FOREACH** data_grp **GENERATE** MIN(data.field1) **as** min_field1;
- A bonus is we can actually combine those two statements using a nested FOREACH.
  - new_data = **FOREACH** (**GROUP** my_data **ALL**) **GENERATE** MIN(data.field1) **as** min_field1;
- Example
  - data_range = **FOREACH** (**GROUP** athletes **ALL**) **GENERATE** MIN(athletes.year) **as** min_year, MAX(athletes.year) **as** max_year;
  - **DUMP** data_range;

Pig Latin

- ORDER BY - You probably just got a lot of data streaming across the screen in a way that wasn't abundantly useful. We already know we can get a smaller data set using LIMIT, but we don't really want a random sampling of data. To get the result set in an ascending or descending rank, we can do the result set order by first, and then using LIMIT.
  - ordered_data = **ORDER** summed_data **BY** field_sum **DESC (or ASC)**;
- Example
  - ordered_medals = **ORDER** medal_sum **BY** medal_count **DESC**;
  - ordered_medals_lim = **LIMIT** ordered_medals **1**;
  - **DUMP** ordered_medals_lim;



Pig Latin

- FILTER BY – In a dataset if we needed to exclude a set of data we can do the same with the FILTER BY command
  - filtered_data1 = **FILTER** my_data **BY** field != 'Field Value';
  - filtered_data2 = **FILTER** my_data **BY** field > **12**;
  - filtered_data3 = **FILTER** my_data **BY** field1 == **0** **AND** field2 < **6**;
- Example
  - athletes_filter = **FILTER** athletes **by** sport != 'Swimming';
  - medal_sum = **FOREACH** (**GROUP** athletes_filter **BY** country) **GENERATE** **group** **as** country, SUM(athletes_filter.total) **as** medal_count;
  - ordered_medals = **ORDER** medal_sum **BY** medal_count **DESC**;
  - ordered_medals_lim = **LIMIT** ordered_medals **1**;
  - **DUMP** ordered_medals_lim;

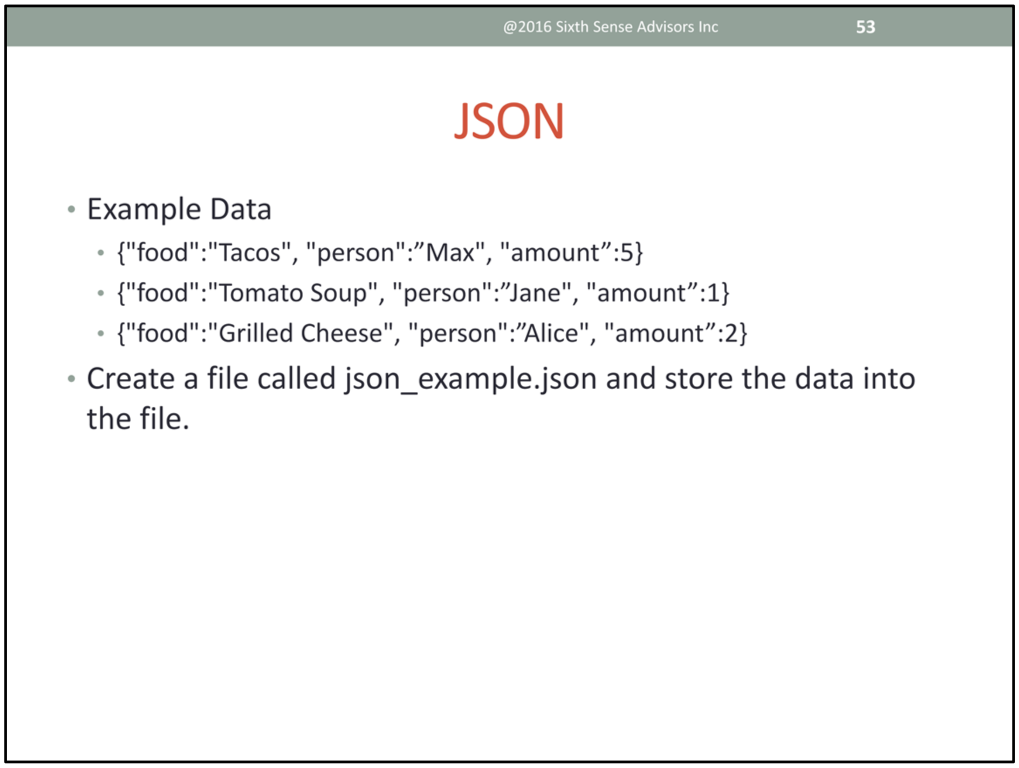
Pig Latin

- User Defined Function (UDF) - Pig allows the use of User-Defined Functions in other languages including Java, Ruby, Python, and Javascript.
  - **from** pig_util **import** outputSchema
  - **@outputSchema(**'score:int'**)**
  - **def** calculate_score**(**gold**,** silver**,** bronze**):**
  - **return 3 *** gold **+ 2 *** silver **+** bronze
- Register the UDF first
  - **REGISTER** 'olympic_udfs.py' **USING** streaming_python **AS** udf;
- Example
  - athlete_score = **FOREACH** athletes **GENERATE** athlete, udf.calculate_score(gold, silver, bronze) **as** score;

Word Count using Pig

```
Lines=LOAD ‘input/hadoop.log’ AS (line: chararray);
Words = FOREACH Lines GENERATE  FLATTEN(TOKENIZE(line)) AS
word;
Groups = GROUP Words BY word;
Counts = FOREACH Groups GENERATE  group, COUNT(Words);
Results = ORDER Words BY Counts DESC;
Top5 = LIMIT Results 5;
STORE Top5 INTO /output/top5words;
```

ADVANCED PIG



JSON

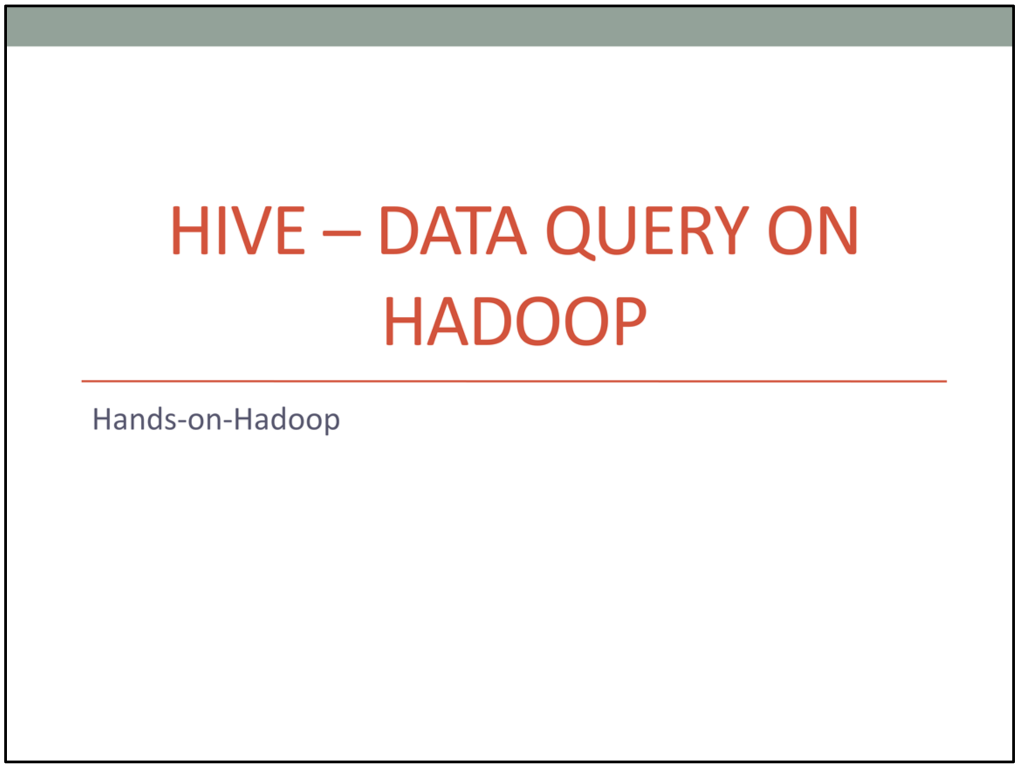
- Example Data
  - {"food":"Tacos", "person":”Max", "amount”:5}
  - {"food":"Tomato Soup", "person":”Jane", "amount”:1}
  - {"food":"Grilled Cheese", "person":”Alice", "amount”:2}
- Create a file called json_example.json and store the data into the file.

Nested Data in JSon

- JSON and Pig both support nesting data. We can store bags of data and tuples in JSON and read the data into Pig.
- Pig expects tuples to be stored in JSON as dictionaries and bags as lists of dictionaries.
- Example
  - "recipe":”Tostada","ingredients":[{"name":”Chicken"},{"name":”Salsa"},{"name":"Cheese"}],"inventor":{"name":”Jane","age”:18}}
    - In this dataset, the ingredients bag is stored as a list of dictionaries ([{"name":”Chicken"},{"name":”Salsa"},{"name":"Cheese"}]). Similarly, the inventor tuple is stored as a dictionary ({"name":”Jane","age”:18}).
  - {"recipe":"TomatoSoup","ingredients":[{"name":"Tomatoes"},{"name":"Milk"}],"inventor":{"name":”Sarah","age”:16}}
  - Copy the example data into a file nested_data.json

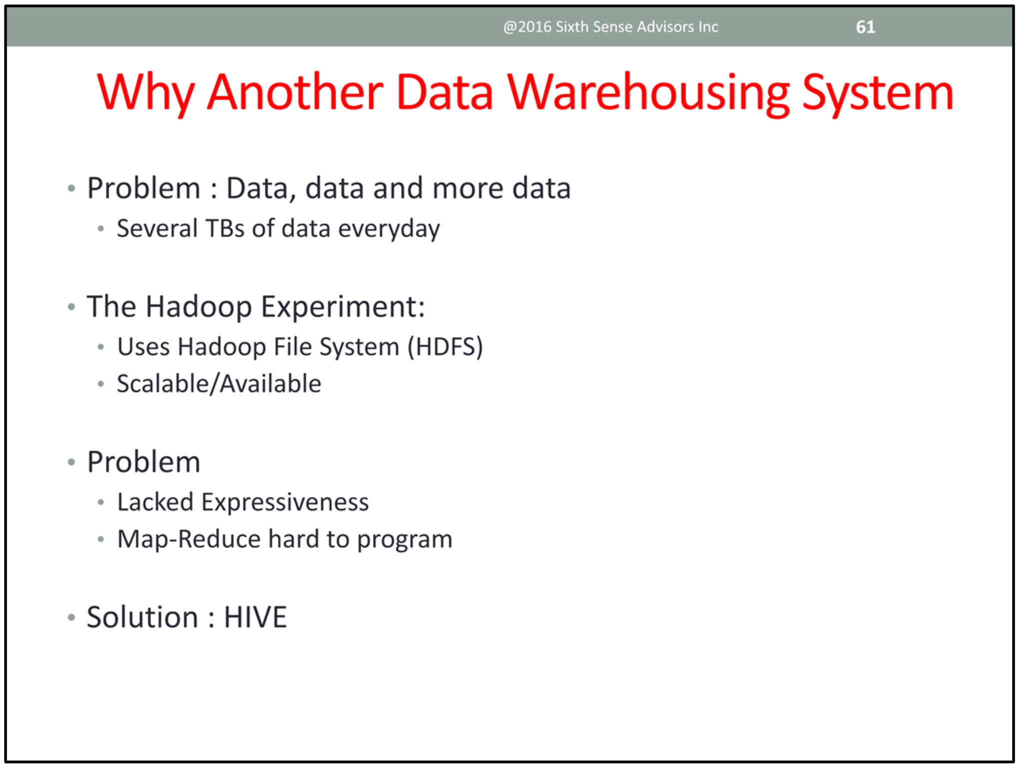
JsonStorage

- Example
  - STORE data_table INTO 'data_table.json' USING JsonStorage();
  - In HDFS the output directory is created with files representing the data is created from the store function.
  - Pig also creates an intermediate file in the folder called ".pig_schema" that explicitly specifies the schema of the output data, which is used for future operations:
    - {"fields":[{"name":"col1","type":55,"description":"autogenerated from Pig Field Schema","schema":null}],"version":0,"sortKeys":[],"sortKeyOrders":[]}



HIVE – DATA QUERY ON HADOOP

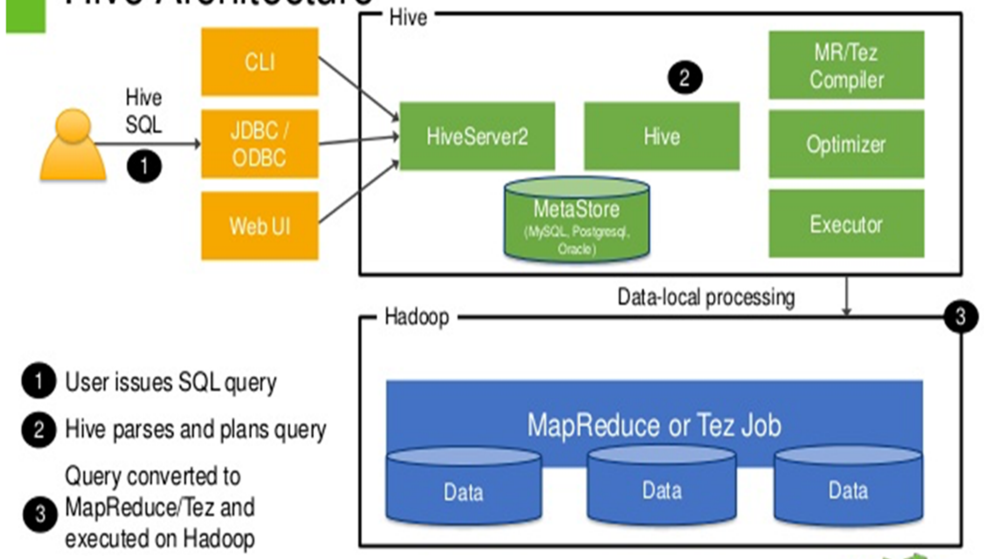
Hands-on-Hadoop



Why Another Data Warehousing System

- Problem : Data, data and more data
  - Several TBs of data everyday
- The Hadoop Experiment:
  - Uses Hadoop File System (HDFS)
  - Scalable/Available
- Problem
  - Lacked Expressiveness
  - Map-Reduce hard to program
- Solution : HIVE

Hive Architecture



1 User issues SQL query

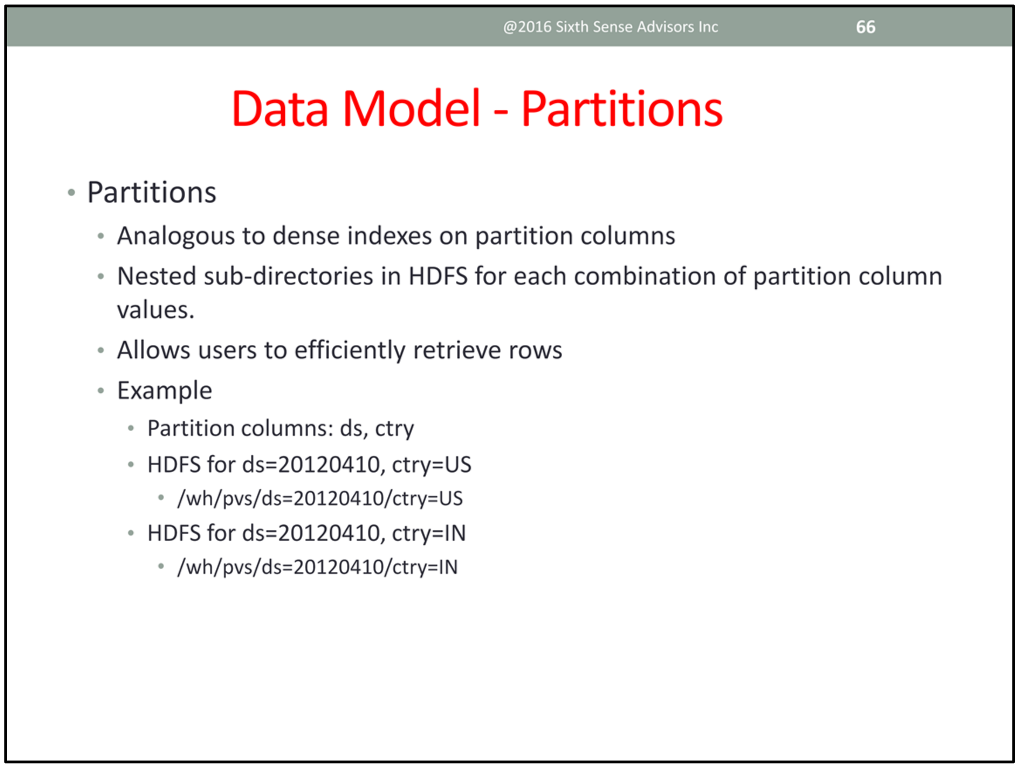
2 Hive parses and plans query

3 Query converted to MapReduce/Tez and executed on Hadoop

Hortonworks

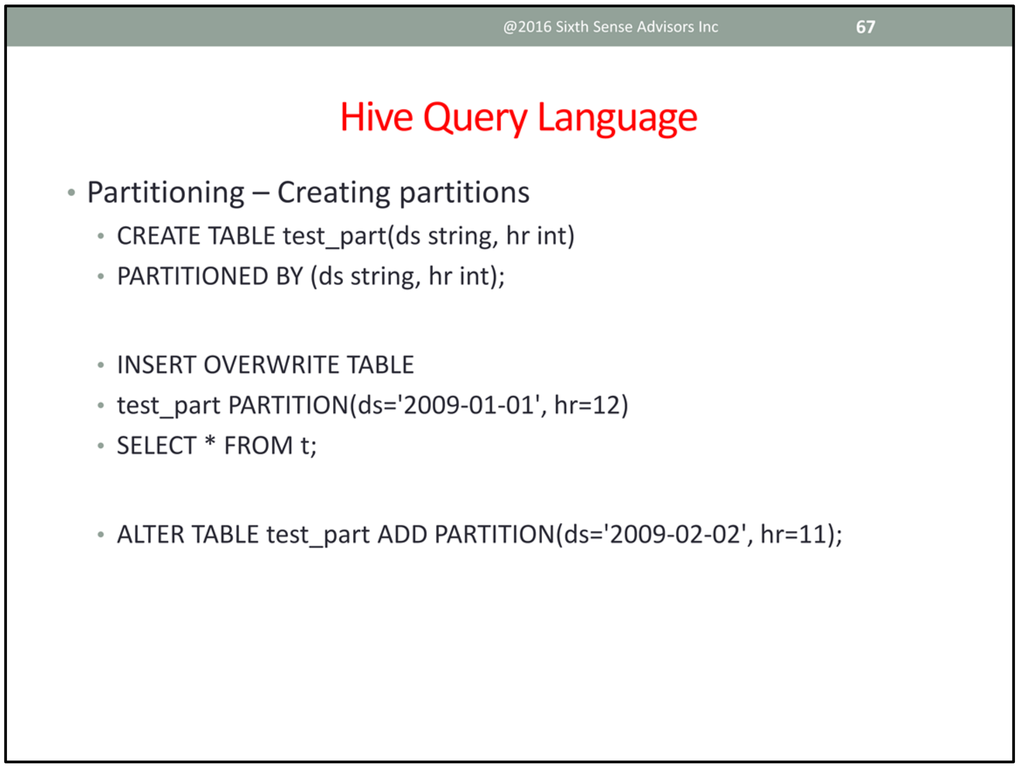
Type System

- Primitive types
  - Integers:TINYINT, SMALLINT, INT, BIGINT.
  - Boolean: BOOLEAN.
  - Floating point numbers: FLOAT, DOUBLE .
  - String: STRING.
- Complex types
  - Structs: {a INT; b INT}.
  - Maps: M['group'].
  - Arrays: ['a', 'b', 'c'], A[1] returns 'b'.



Data Model - Partitions

- Partitions
  - Analogous to dense indexes on partition columns
  - Nested sub-directories in HDFS for each combination of partition column values.
  - Allows users to efficiently retrieve rows
  - Example
    - Partition columns: ds, ctry
    - HDFS for ds=20120410, ctry=US
      - /wh/pvs/ds=20120410/ctry=US
    - HDFS for ds=20120410, ctry=IN
      - /wh/pvs/ds=20120410/ctry=IN

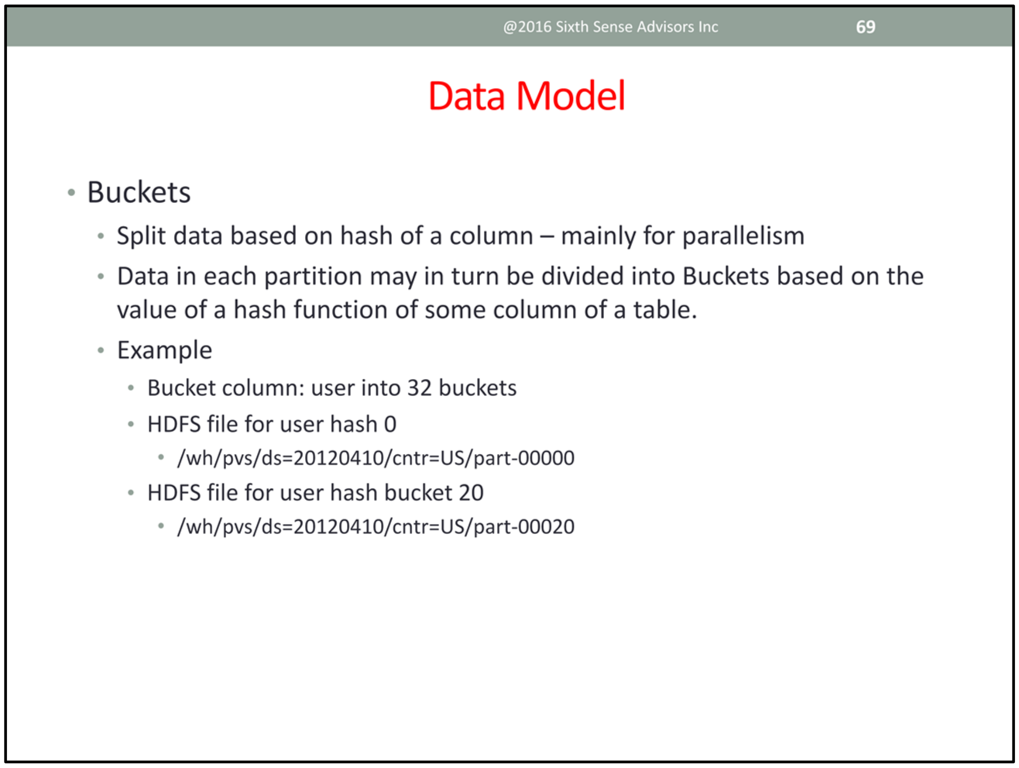


Hive Query Language

- Partitioning – Creating partitions
  - CREATE TABLE test_part(ds string, hr int)
  - PARTITIONED BY (ds string, hr int);

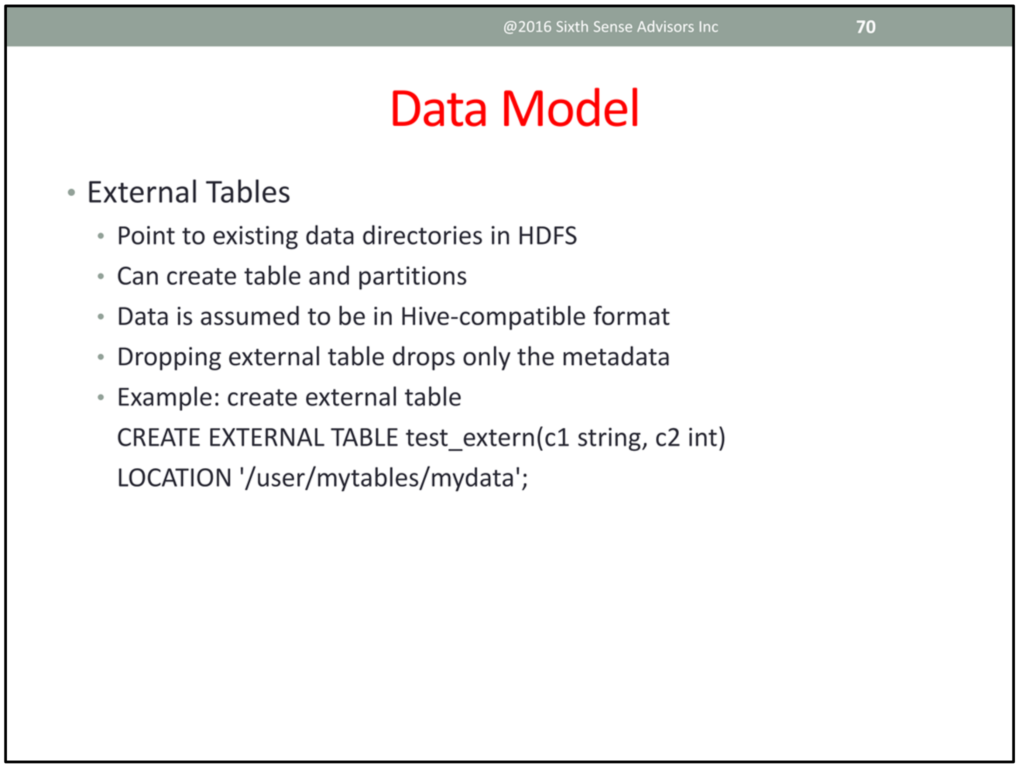
  - INSERT OVERWRITE TABLE
  - test_part PARTITION(ds='2009-01-01', hr=12)
  - SELECT * FROM t;

  - ALTER TABLE test_part ADD PARTITION(ds='2009-02-02', hr=11);



Data Model

- Buckets
  - Split data based on hash of a column – mainly for parallelism
  - Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of a table.
  - Example
    - Bucket column: user into 32 buckets
    - HDFS file for user hash 0
      - /wh/pvs/ds=20120410/cntr=US/part-00000
    - HDFS file for user hash bucket 20
      - /wh/pvs/ds=20120410/cntr=US/part-00020



Data Model

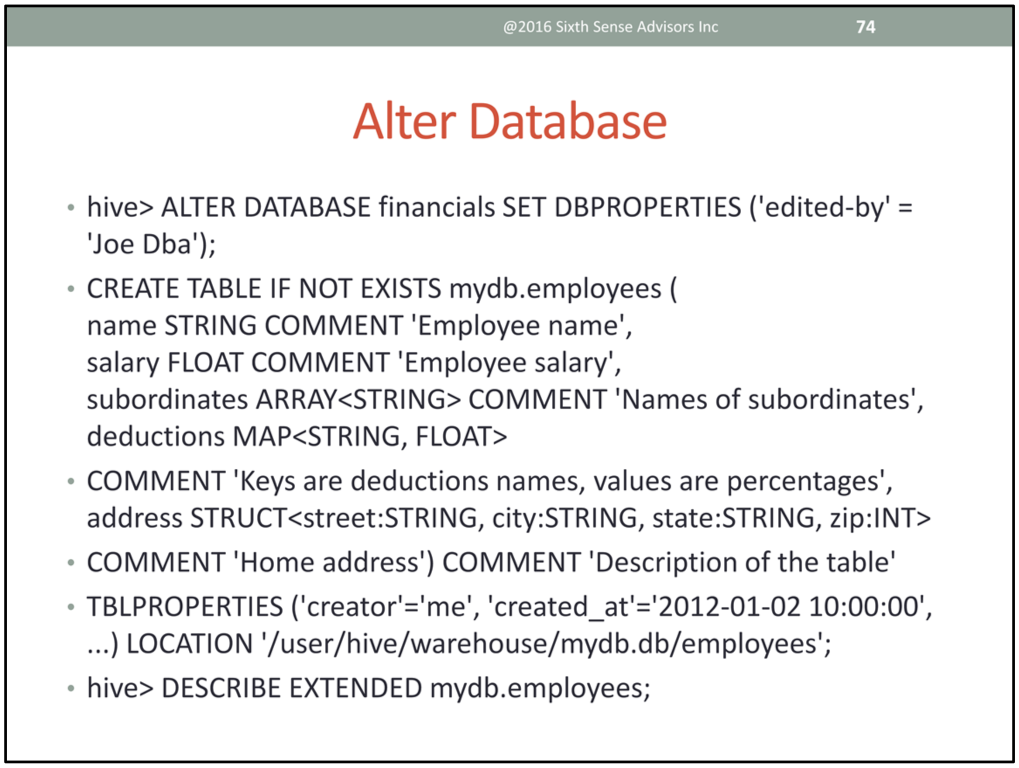
- External Tables
  - Point to existing data directories in HDFS
  - Can create table and partitions
  - Data is assumed to be in Hive-compatible format
  - Dropping external table drops only the metadata
  - Example: create external table

```
CREATE EXTERNAL TABLE test_extern(c1 string, c2 int)
LOCATION '/user/mytables/mydata';
```

Hive File Formats

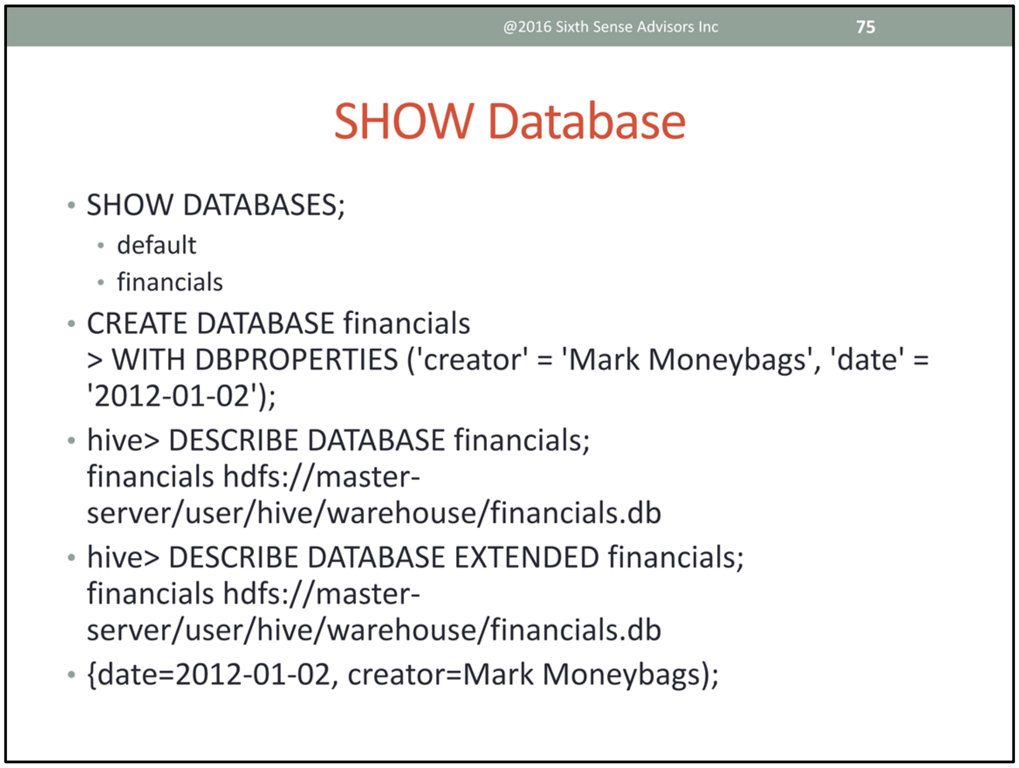
- Hive lets users store different file formats
- Helps in performance improvements
- SQL Example:

```
CREATE TABLE dest1(key INT, value STRING)
STORED AS
INPUTFORMAT
'org.apache.hadoop.mapred.SequenceFileInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.mapred.SequenceFileOutputFormat'
```



Alter Database

- hive> ALTER DATABASE financials SET DBPROPERTIES ('edited-by' = 'Joe Dba');
- CREATE TABLE IF NOT EXISTS mydb.employees (
  name STRING COMMENT 'Employee name',
  salary FLOAT COMMENT 'Employee salary',
  subordinates ARRAY<STRING> COMMENT 'Names of subordinates',
  deductions MAP<STRING, FLOAT>
- COMMENT 'Keys are deductions names, values are percentages',
  address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
- COMMENT 'Home address') COMMENT 'Description of the table'
- TBLPROPERTIES ('creator'='me', 'created_at'='2012-01-02 10:00:00', ...) LOCATION '/user/hive/warehouse/mydb.db/employees';
- hive> DESCRIBE EXTENDED mydb.employees;



SHOW Database

- SHOW DATABASES;
  - default
  - financials
- CREATE DATABASE financials
  > WITH DBPROPERTIES ('creator' = 'Mark Moneybags', 'date' = '2012-01-02');
- hive> DESCRIBE DATABASE financials;
  financials hdfs://master-server/user/hive/warehouse/financials.db
- hive> DESCRIBE DATABASE EXTENDED financials;
  financials hdfs://master-server/user/hive/warehouse/financials.db
- {date=2012-01-02, creator=Mark Moneybags);

Create External Table

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (
exchange STRING,
symbol STRING,
ymd STRING,
price_open FLOAT,
price_high FLOAT,
price_low FLOAT,
price_close FLOAT,
volume INT,
price_adj_close FLOAT)
CLUSTERED BY (exchange, symbol)
SORTED BY (ymd ASC)
INTO 96 BUCKETS
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/stocks';
```

Create Table

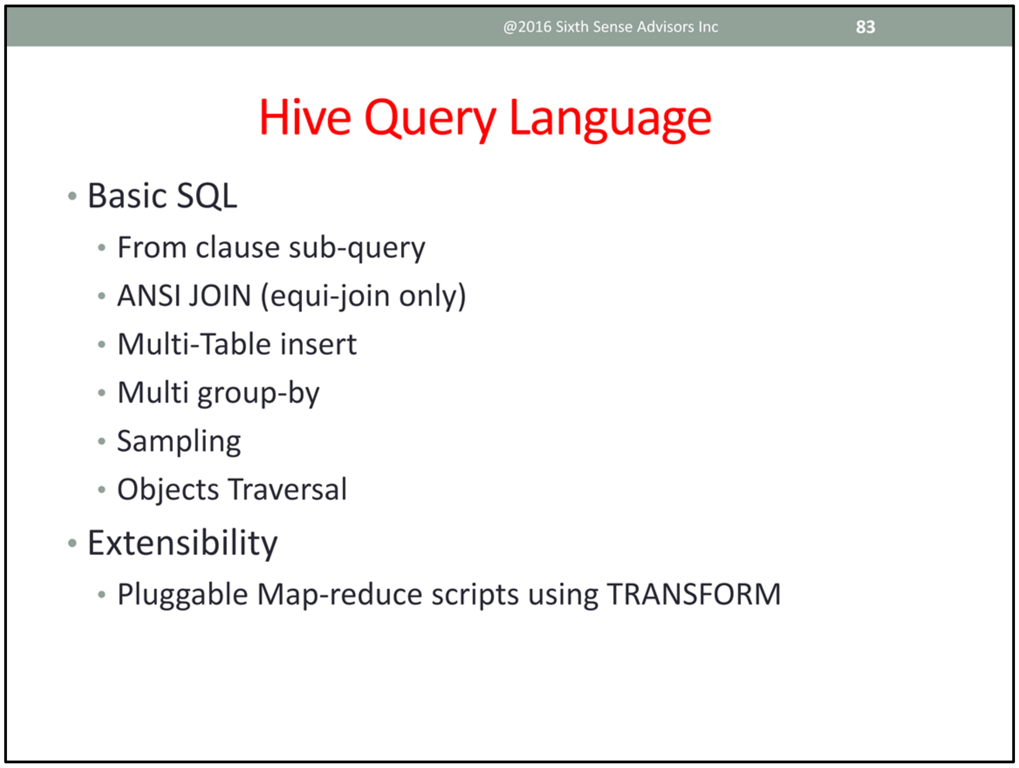
- CREATE TABLE employees (
  name STRING,
  salary FLOAT,
  subordinates ARRAY<STRING>,
  deductions MAP<STRING, FLOAT>,
  address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>)
  PARTITIONED BY (country STRING, state STRING);

Alter Table

- ALTER TABLE log_messages
  CHANGE COLUMN hms hours_minutes_seconds INT
  COMMENT 'The hours, minutes, and seconds part of the timestamp' AFTER severity;
- ALTER TABLE log_messages ADD COLUMNS (
  app_name STRING COMMENT 'Application name', session_id LONG COMMENT 'The current session id');

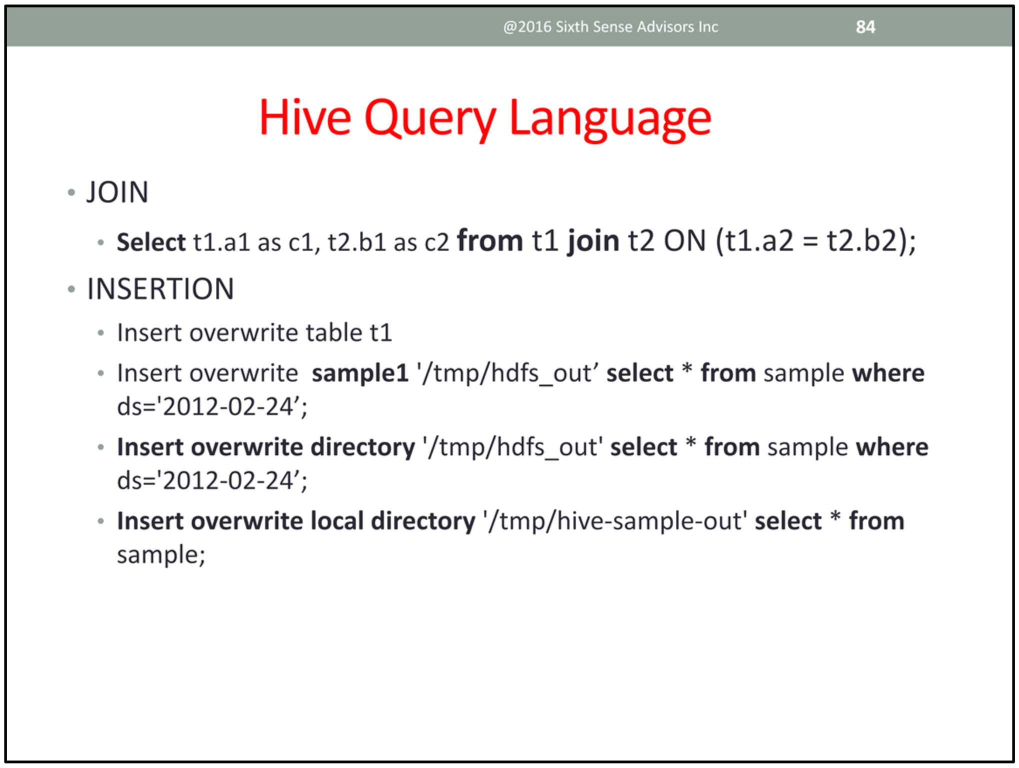
Alter Table

- ALTER TABLE log_messages REPLACE COLUMNS (hours_mins_secs INT COMMENT 'hour, minute, seconds from timestamp', severity STRING COMMENT 'The message severity' message STRING COMMENT 'The rest of the message');
- ALTER TABLE log_messages PARTITION(year = 2012, month = 1, day = 1) SET FILEFORMAT SEQUENCEFILE;



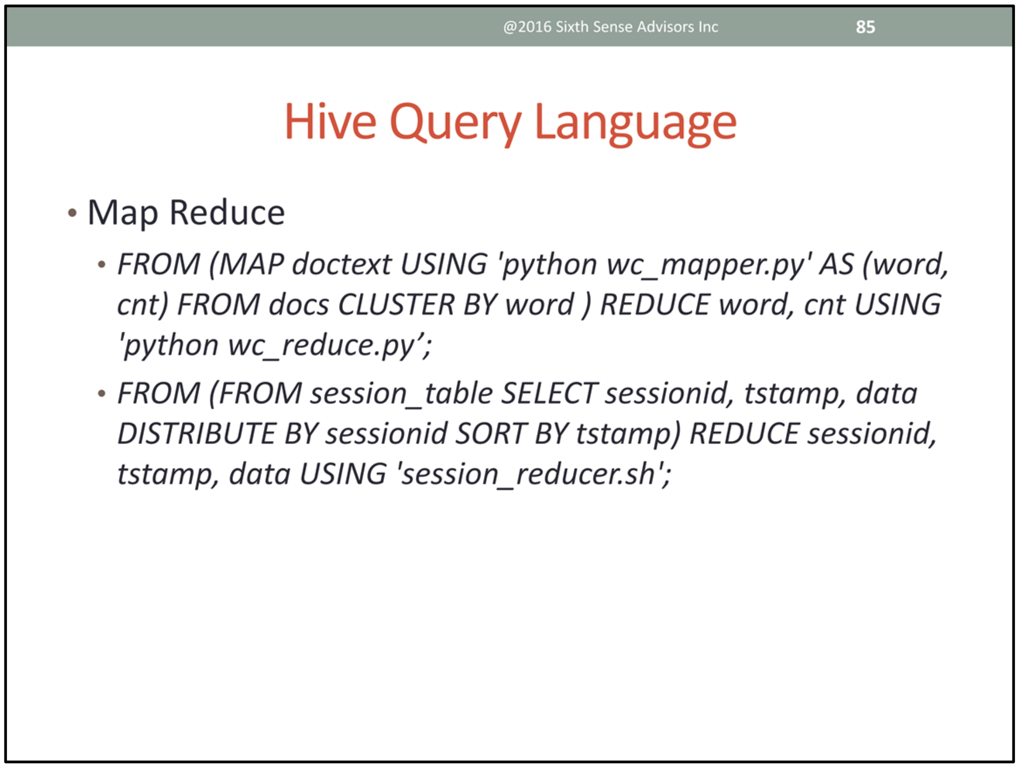
Hive Query Language

- Basic SQL
  - From clause sub-query
  - ANSI JOIN (equi-join only)
  - Multi-Table insert
  - Multi group-by
  - Sampling
  - Objects Traversal
- Extensibility
  - Pluggable Map-reduce scripts using TRANSFORM



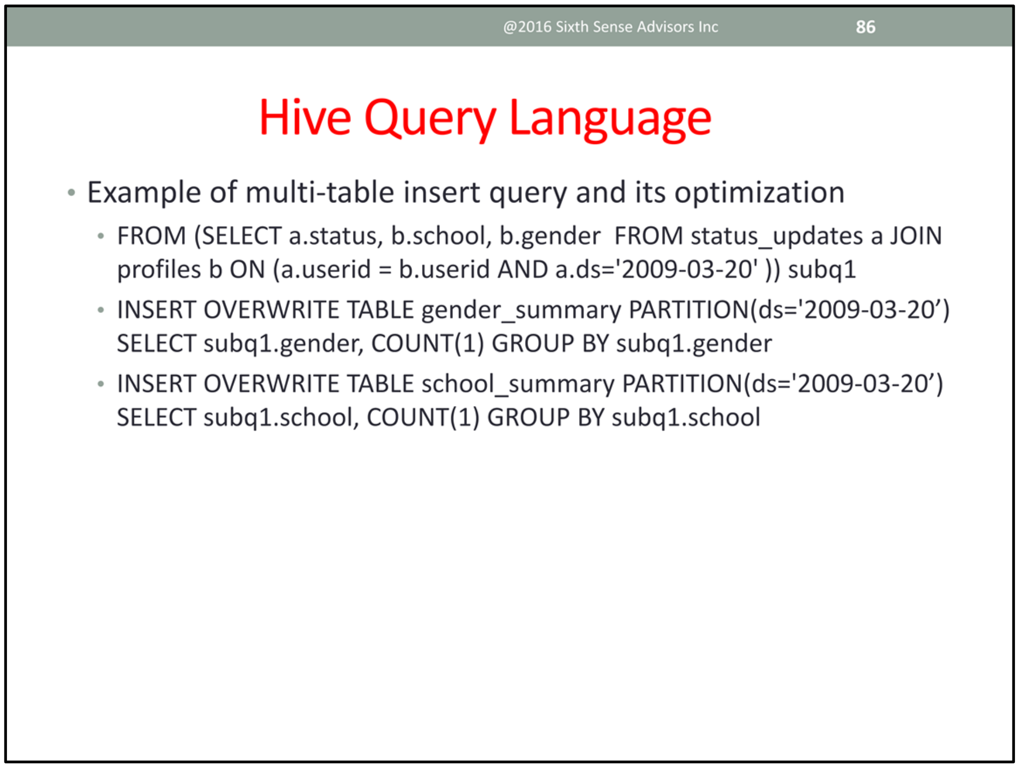
Hive Query Language

- JOIN
  - **Select** t1.a1 as c1, t2.b1 as c2 **from** t1 **join** t2 ON (t1.a2 = t2.b2);
- INSERTION
  - Insert overwrite table t1
  - Insert overwrite **sample1** '/tmp/hdfs_out’ **select** * **from** sample **where** ds='2012-02-24’;
  - **Insert overwrite directory** '/tmp/hdfs_out' **select** * **from** sample **where** ds='2012-02-24’;
  - **Insert overwrite local directory** '/tmp/hive-sample-out' **select** * **from** sample;



Hive Query Language

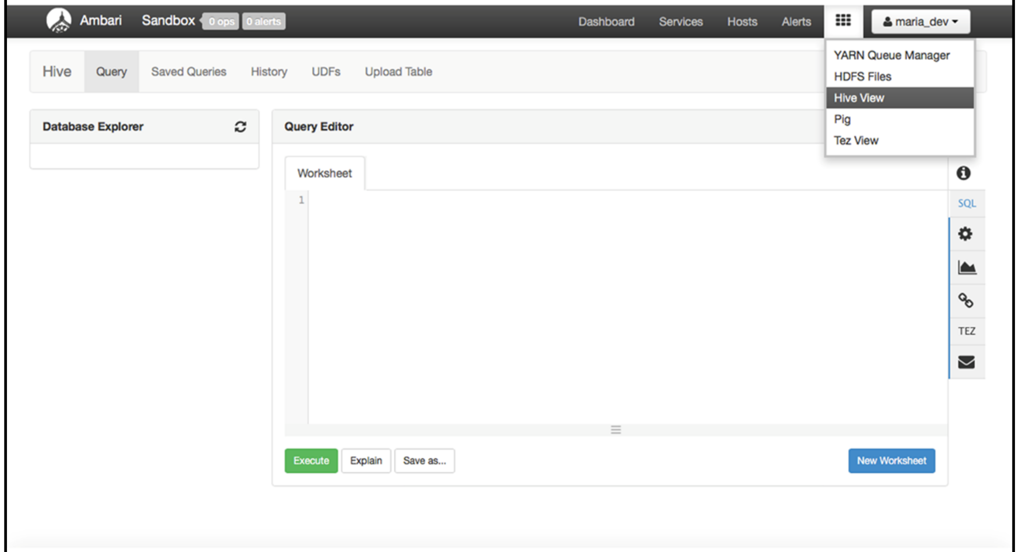
- Map Reduce
  - *FROM (MAP doctext USING 'python wc_mapper.py' AS (word, cnt) FROM docs CLUSTER BY word) REDUCE word, cnt USING 'python wc_reduce.py’;*
  - *FROM (FROM session_table SELECT sessionid, tstamp, data DISTRIBUTE BY sessionid SORT BY tstamp) REDUCE sessionid, tstamp, data USING 'session_reducer.sh';*



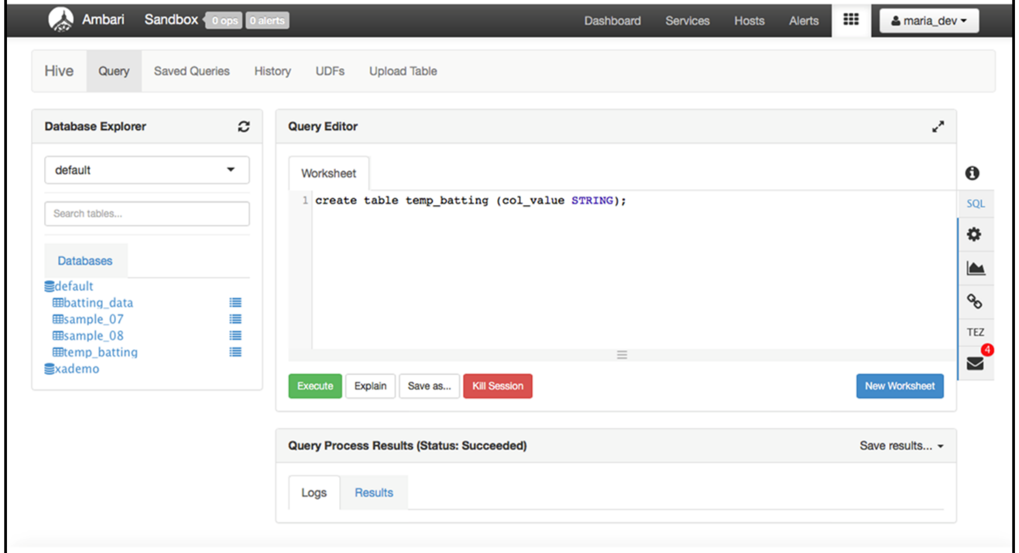
Hive Query Language

- Example of multi-table insert query and its optimization
  - FROM (SELECT a.status, b.school, b.gender FROM status_updates a JOIN profiles b ON (a.userid = b.userid AND a.ds='2009-03-20')) subq1
  - INSERT OVERWRITE TABLE gender_summary PARTITION(ds='2009-03-20’) SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender
  - INSERT OVERWRITE TABLE school_summary PARTITION(ds='2009-03-20’) SELECT subq1.school, COUNT(1) GROUP BY subq1.school

Hive Query UI Launch

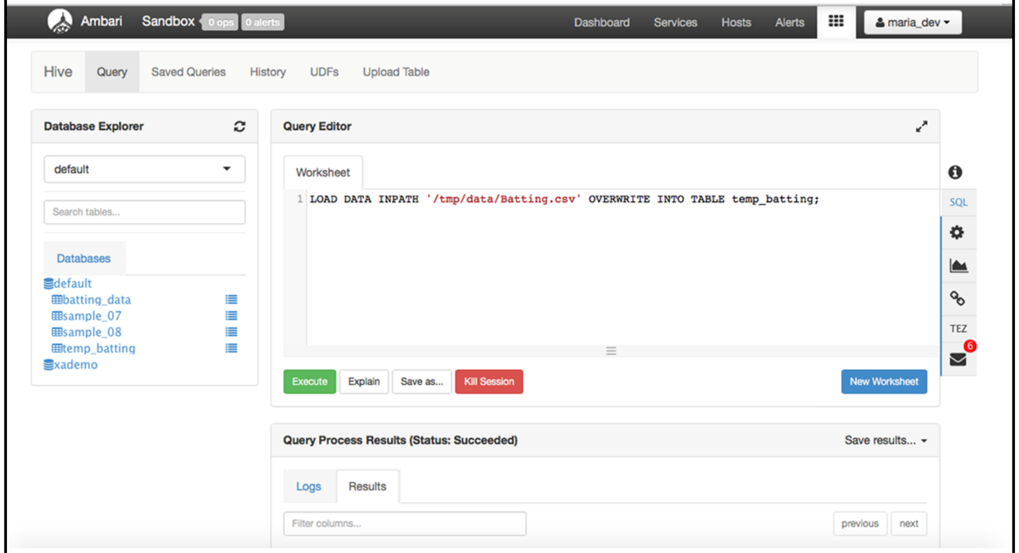


Hive Query Execution



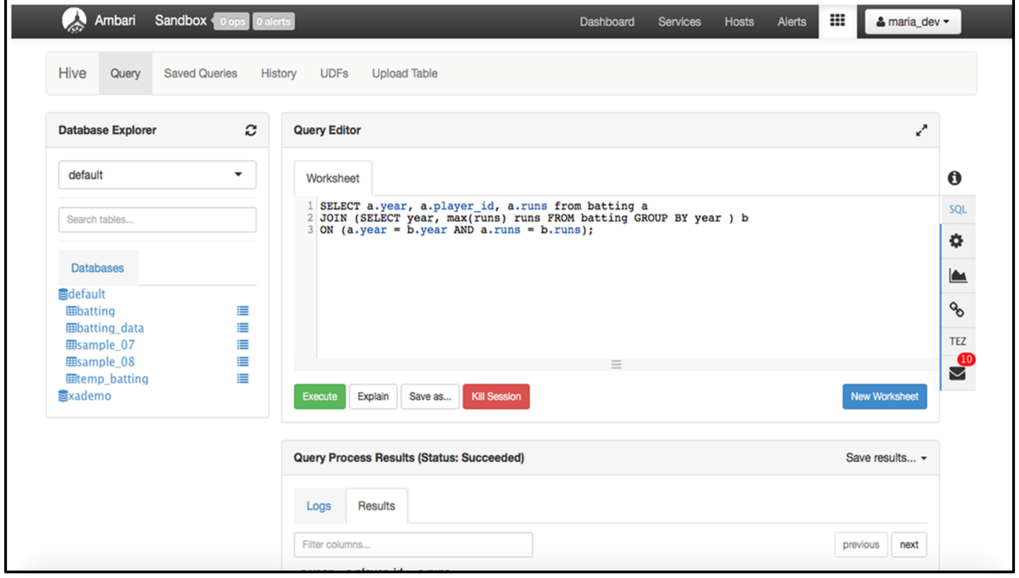
```
create table temp_batting (col_value STRING);
```

Hive Query Execution

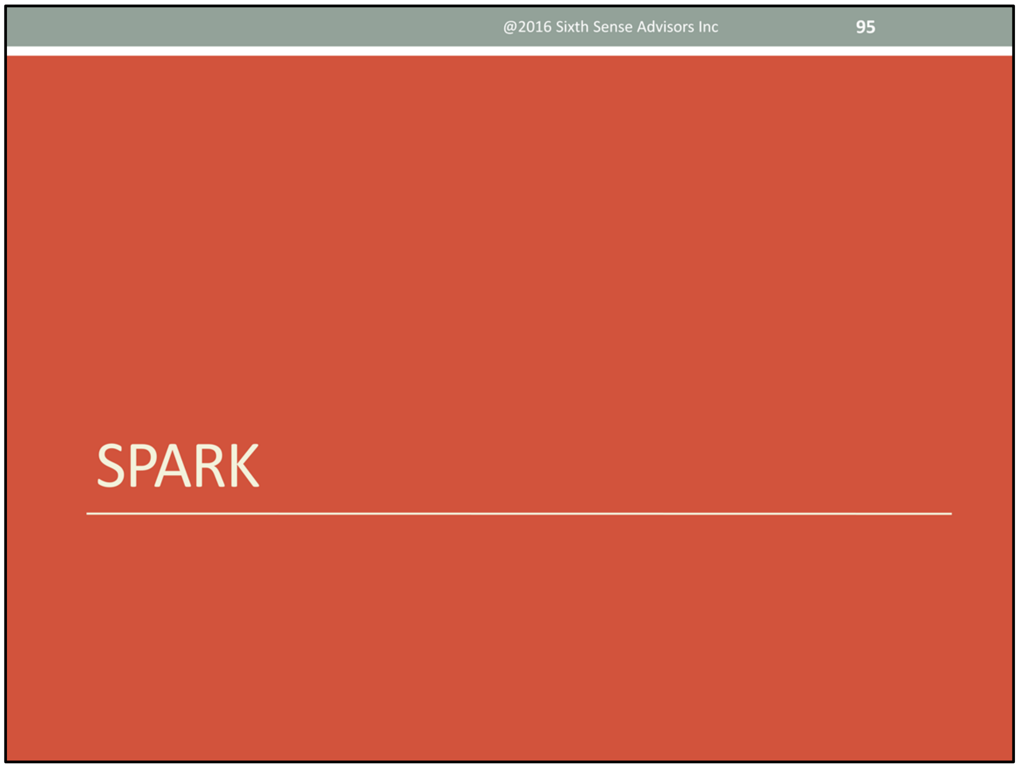


```
LOAD DATA INPATH '/tmp/data/Batting.csv' OVERWRITE INTO TABLE temp_batting;
```

Hive Query Execution



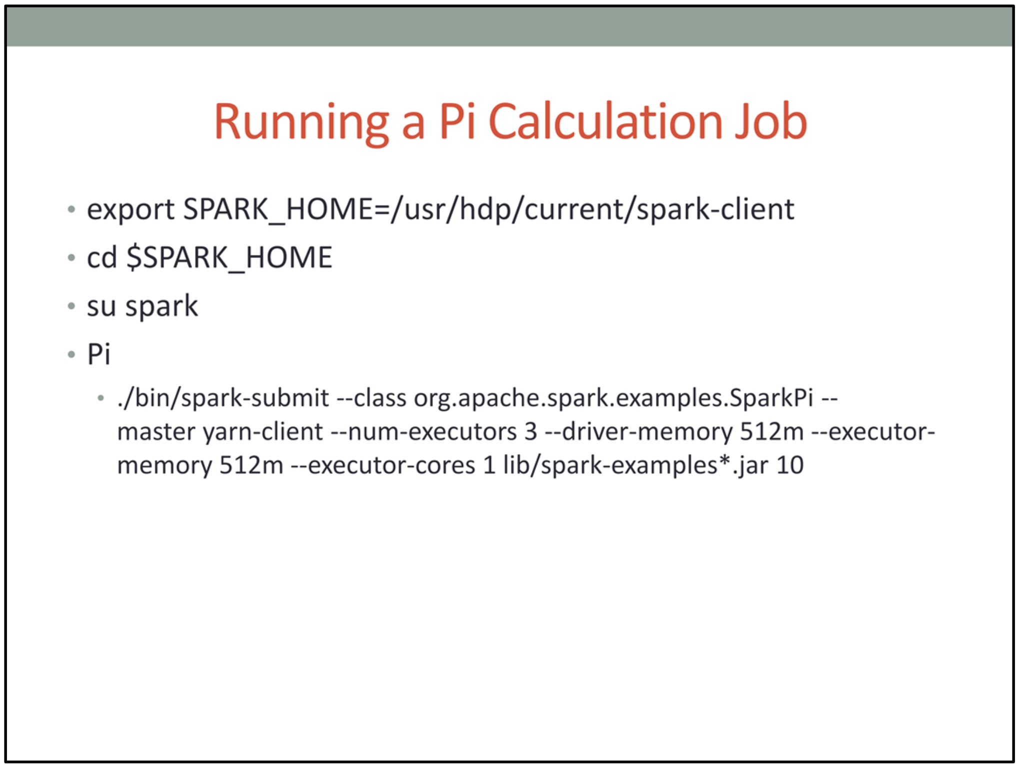
```
SELECT a.year, a.player_id, a.runs from batting a
JOIN (SELECT year, max(runs) runs FROM batting GROUP BY year ) b
ON (a.year = b.year AND a.runs = b.runs);
```



SPARK

Starting Spark

- http://127.0.0.1:4200 - Sandbox login
- Change password: sparkclass



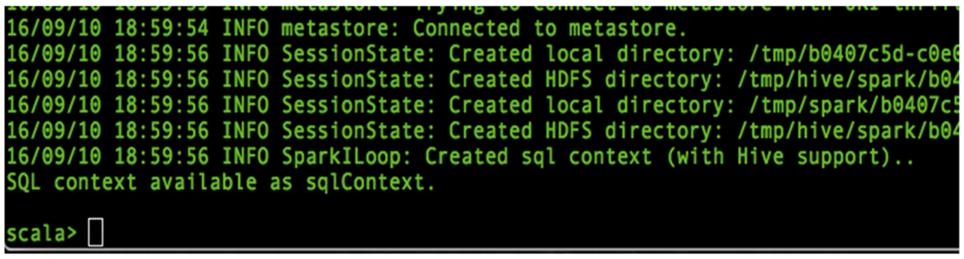
Running a Pi Calculation Job

- `export SPARK_HOME=/usr/hdp/current/spark-client`
- `cd $SPARK_HOME`
- `su spark`
- Pi

  -
    ```
    ./bin/spark-submit --class org.apache.spark.examples.SparkPi --
    master yarn-client --num-executors 3 --driver-memory 512m --executor-
    memory 512m --executor-cores 1 lib/spark-examples*.jar 10
    ```

Wordcount Job

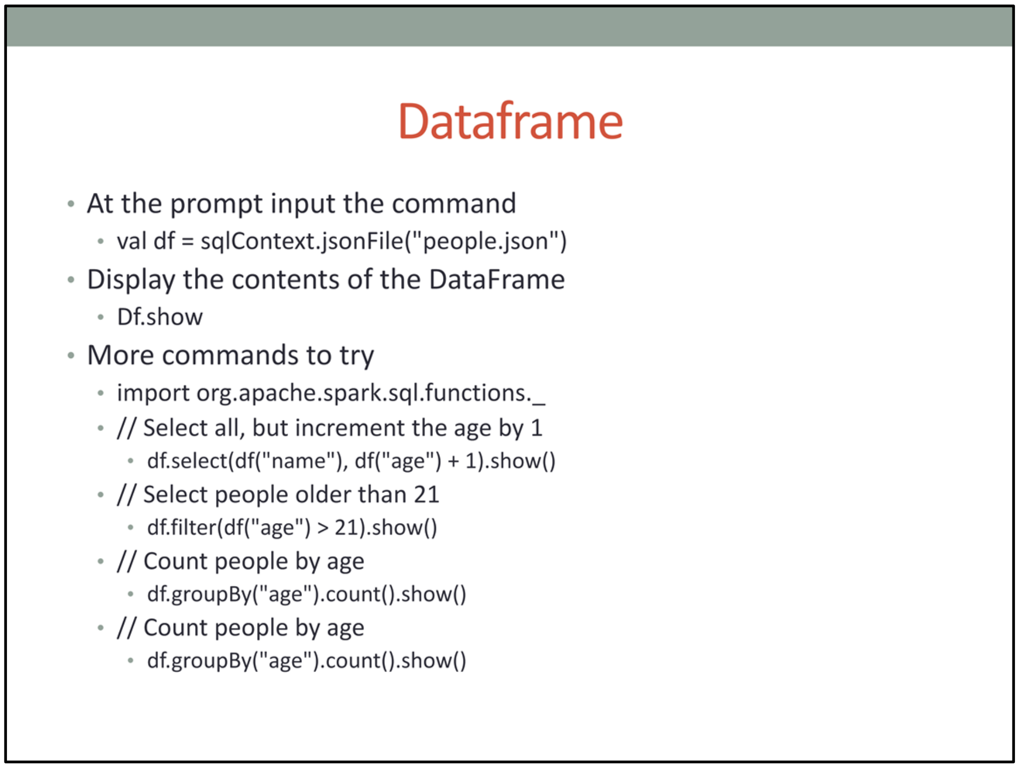
- Wordcount
  - hadoop fs -copyFromLocal /etc/hadoop/conf/log4j.properties /tmp/data
- Spark shell
  - ./bin/spark-shell --master yarn-client --driver-memory 512m --executor-memory 512m



```
16/09/10 18:59:54 INFO metastore: Connected to metastore.
16/09/10 18:59:56 INFO SessionState: Created local directory: /tmp/b0407c5d-c0e
16/09/10 18:59:56 INFO SessionState: Created HDFS directory: /tmp/hive/spark/b04
16/09/10 18:59:56 INFO SessionState: Created local directory: /tmp/spark/b0407c5
16/09/10 18:59:56 INFO SessionState: Created HDFS directory: /tmp/hive/spark/b04
16/09/10 18:59:56 INFO SparkILoop: Created sql context (with Hive support)..
SQL context available as sqlContext.

scala> 
```

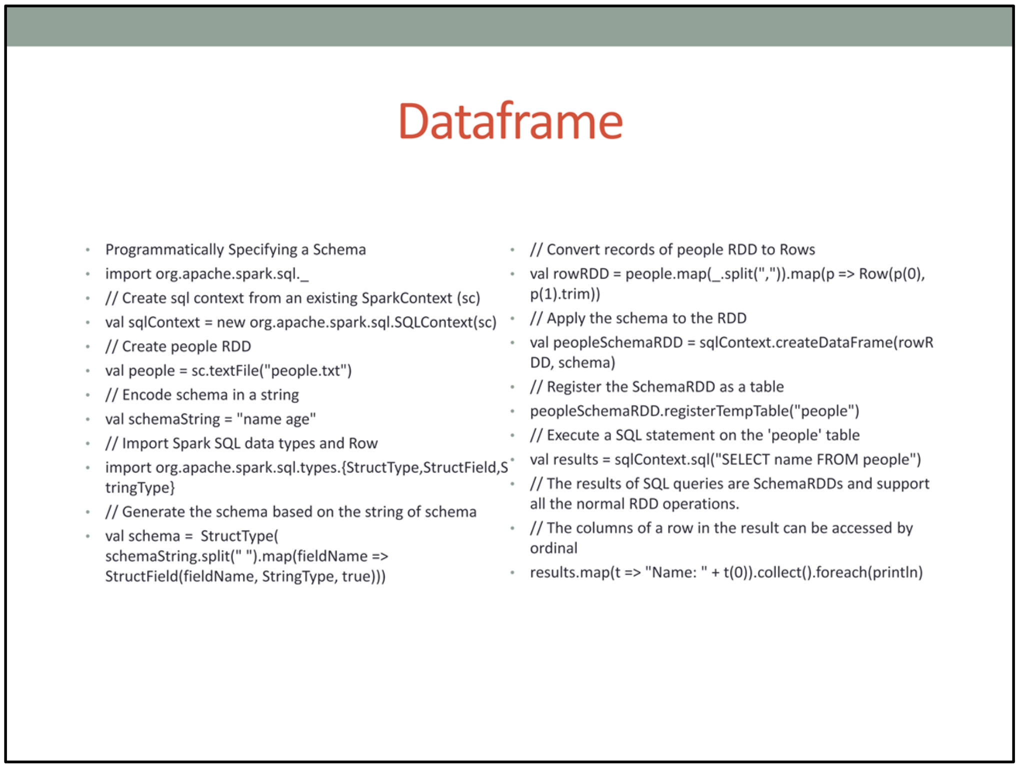
Wordcount - 2

- val lines = sc.textFile(“/tmp/data/filename.txt”)
- **val** counts = lines.flatMap(line **=>** line.split(" ")).map(word **=>** (word, 1)).reduceByKey(_ + _)



Dataframe

- At the prompt input the command
  - val df = sqlContext.jsonFile("people.json")
- Display the contents of the DataFrame
  - Df.show
- More commands to try
  - import org.apache.spark.sql.functions._
  - // Select all, but increment the age by 1
    - df.select(df("name"), df("age") + 1).show()
  - // Select people older than 21
    - df.filter(df("age") > 21).show()
  - // Count people by age
    - df.groupBy("age").count().show()
  - // Count people by age
    - df.groupBy("age").count().show()



Dataframe

- Programmatically Specifying a Schema
- import org.apache.spark.sql._
- // Create sql context from an existing SparkContext (sc)
- val sqlContext = new org.apache.spark.sql.SQLContext(sc)
- // Create people RDD
- val people = sc.textFile("people.txt")
- // Encode schema in a string
- val schemaString = "name age"
- // Import Spark SQL data types and Row
- import org.apache.spark.sql.types.{StructType,StructField,StringType}
- // Generate the schema based on the string of schema
- val schema = StructType(schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))
- // Convert records of people RDD to Rows
- val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
- // Apply the schema to the RDD
- val peopleSchemaRDD = sqlContext.createDataFrame(rowRDD, schema)
- // Register the SchemaRDD as a table
- peopleSchemaRDD.registerTempTable("people")
- // Execute a SQL statement on the 'people' table
- val results = sqlContext.sql("SELECT name FROM people")
- // The results of SQL queries are SchemaRDDs and support all the normal RDD operations.
- // The columns of a row in the result can be accessed by ordinal
- results.map(t => "Name: " + t(0)).collect().foreach(println)

SparkSQL

- At prompt on beeline
  - show tables;
  - Will show you tables available
- Type Ctrl+C to exit beeline.
- **Stop Thrift Server**
  - ./sbin/stop-thriftserver.sh