Transformation Processing Smackdown
Spark vs Hive vs Pig

Lester Martin

DevNexus 2017

Connection before Content

Lester Martin – Hadoop/Spark Trainer & Consultant

lester.martin@gmail.com

http://lester.website *(links to blog, twitter, github, LI, FB, etc)*

Agenda

- **Present Frameworks**
- **File Formats**
- **Source to Target Mappings**
- **Data Quality**
- **Data Profiling**
- **Core Processing Functionality**
- **Custom Business Logic**
- **Mutable Data Concerns**
- **Performance**

! DS/ML

Lots of Code !

Standard Disclaimers Apply

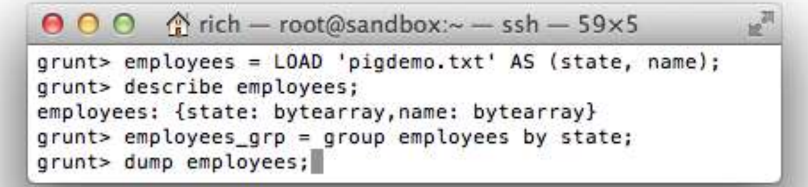
Wide Topic – Multiple Frameworks – Limited Time, *so…*

- **Simple use cases**
  - Glad to enhance https://github.com/lestermartin/oss-transform-processing-comparison
- **ALWAYS 2+ ways to skin a cat**
  - Especially with Spark ;-)
- **CLI, not GUI, tools**
  - Others in that space such as Talend, Informatica & Syncsort
- **ALL code compiles in PPT ;-)**
- ***Won't explain all examples!!!***

Apache Pig – http://pig.apache.org

- **A high-level data-flow scripting language (Pig Latin)**
- **Run as standalone scripts or use the interactive shell**
- **Executes on Hadoop**
- **Uses lazy execution**

Grunt shell

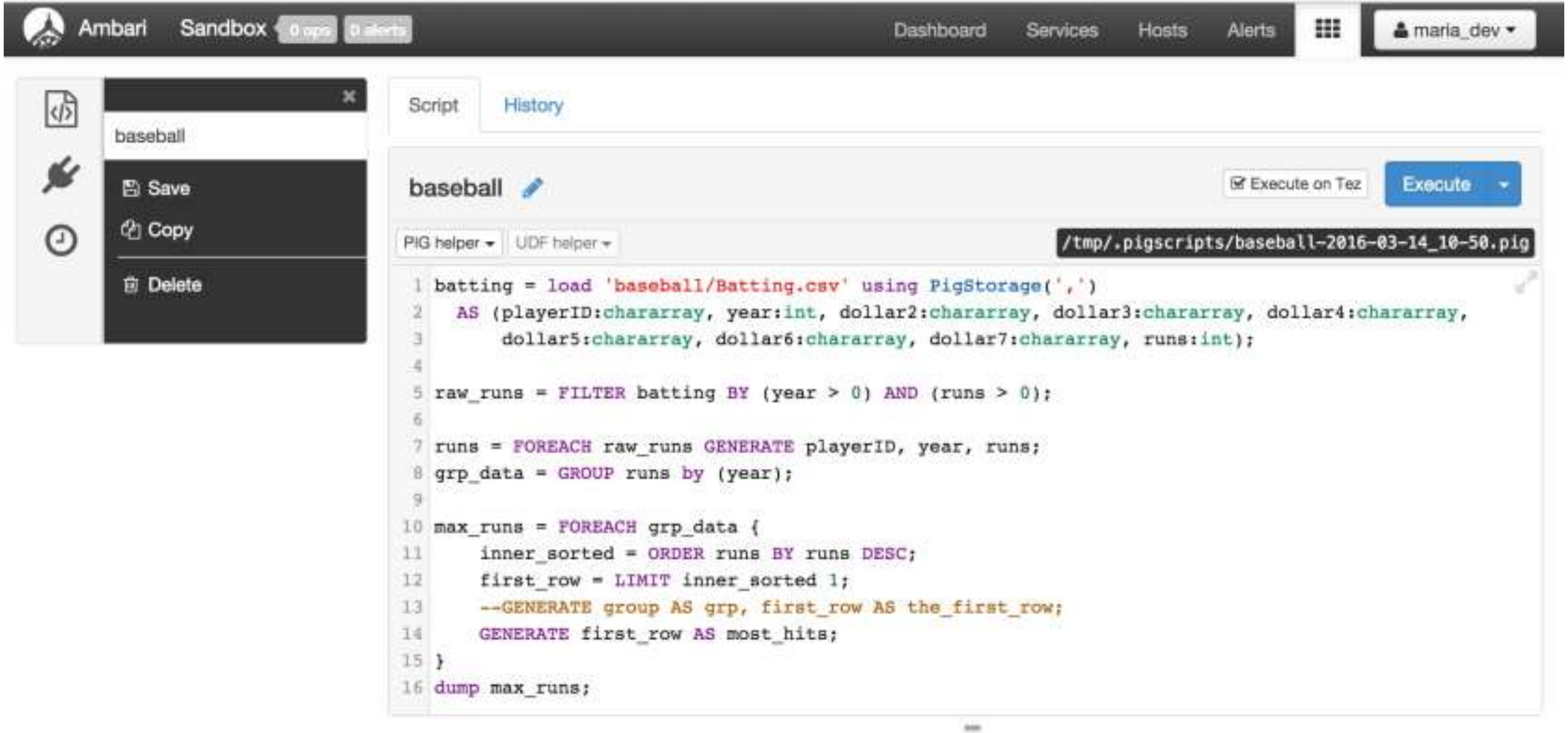


```
grunt> employees = LOAD 'pigdemo.txt' AS (state, name);
grunt> describe employees;
employees: {state: bytearray,name: bytearray}
grunt> employees_grp = group employees by state;
grunt> dump employees;
```

Simple and Novel Commands

Pig Command	Description
LOAD	Read data from file system
STORE	Write data to file system
FOREACH	Apply expression to each record and output 1+ records
FILTER	Apply predicate and remove records that do not return true
GROUP/COGROUP	Collect records with the same key from one or more inputs
JOIN	Joint 2+ inputs based on a key; various join algorithms exist
ORDER	Sort records based on a key
DISTINCT	Remove duplicate records
UNION	Merge two data sets
SPLIT	Split data into 2+ more sets based on filter conditions
STREAM	Send all records through a user provided executable
SAMPLE	Read a random sample of the data
LIMIT	Limit the number of records

Executing Scripts in Ambari Pig View



/tmp/.pigscripts/baseball-2016-03-14_10-50.pig

```
batting = load 'baseball/Batting.csv' using PigStorage(',')
  AS (playerID:chararray, year:int, dollar2:chararray, dollar3:chararray, dollar4:chararray,
      dollar5:chararray, dollar6:chararray, dollar7:chararray, runs:int);

raw_runs = FILTER batting BY (year > 0) AND (runs > 0);

runs = FOREACH raw_runs GENERATE playerID, year, runs;
grp_data = GROUP runs by (year);

max_runs = FOREACH grp_data {
    inner_sorted = ORDER runs BY runs DESC;
    first_row = LIMIT inner_sorted 1;
    --GENERATE group AS grp, first_row AS the_first_row;
    GENERATE first_row AS most_hits;
}
dump max_runs;
```

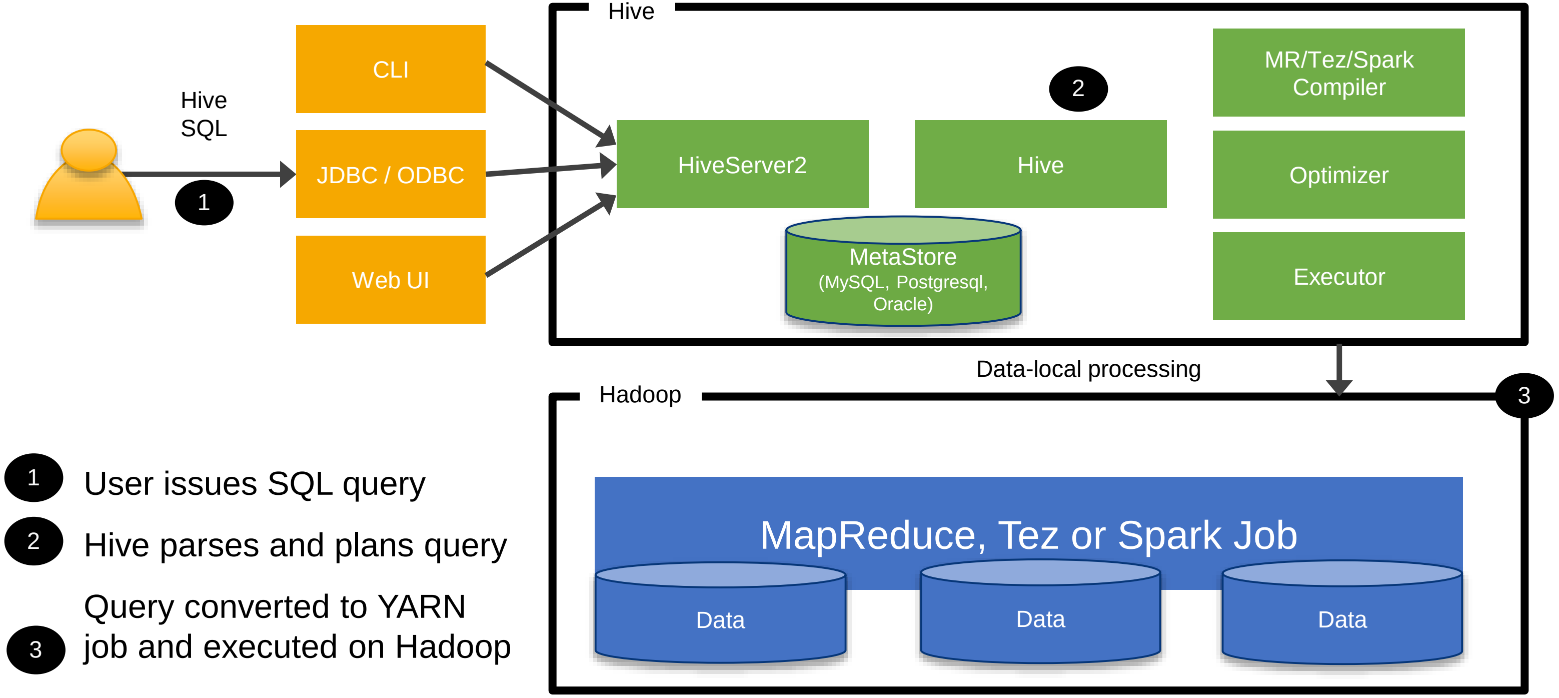
Apache Hive – http://hive.apache.org

- Data warehouse system for Hadoop
- Create schema/table definitions that point to data in HDFS
- Treat your data in Hadoop as tables
- SQL 92
- Interactive queries at scale

Hive’s Alignment with SQL

SQL Datatypes	SQL Semantics
INT	SELECT, LOAD, INSERT from query
TINYINT/SMALLINT/BIGINT	Expressions in WHERE and HAVING
BOOLEAN	GROUP BY, ORDER BY, SORT BY
FLOAT	CLUSTER BY, DISTRIBUTE BY
DOUBLE	Sub-queries in FROM clause
STRING	GROUP BY, ORDER BY
BINARY	ROLLUP and CUBE
TIMESTAMP	UNION
ARRAY, MAP, STRUCT, UNION	LEFT, RIGHT and FULL INNER/OUTER JOIN
DECIMAL	CROSS JOIN, LEFT SEMI JOIN
CHAR	Windowing functions (OVER, RANK, etc.)
VARCHAR	Sub-queries for IN/NOT IN, HAVING
DATE	EXISTS / NOT EXISTS
	INTERSECT, EXCEPT

Hive Query Process

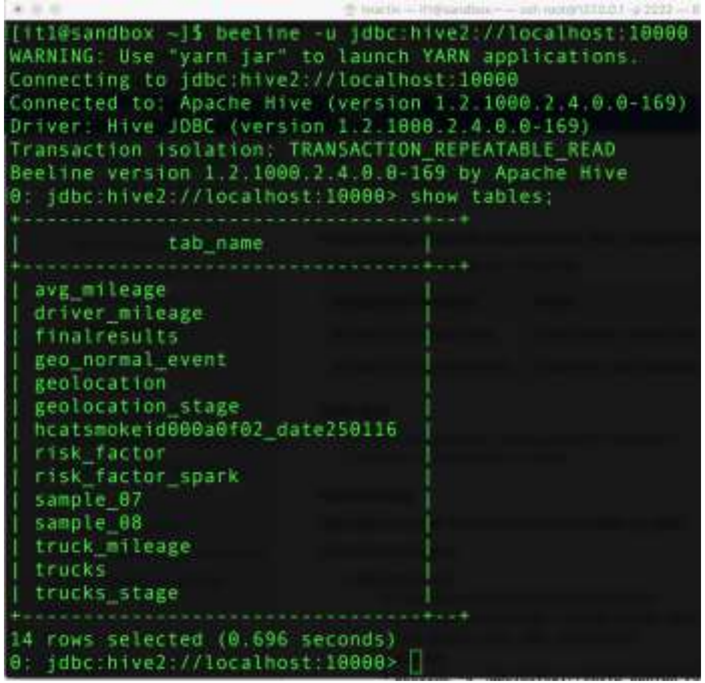


1 User issues SQL query

2 Hive parses and plans query

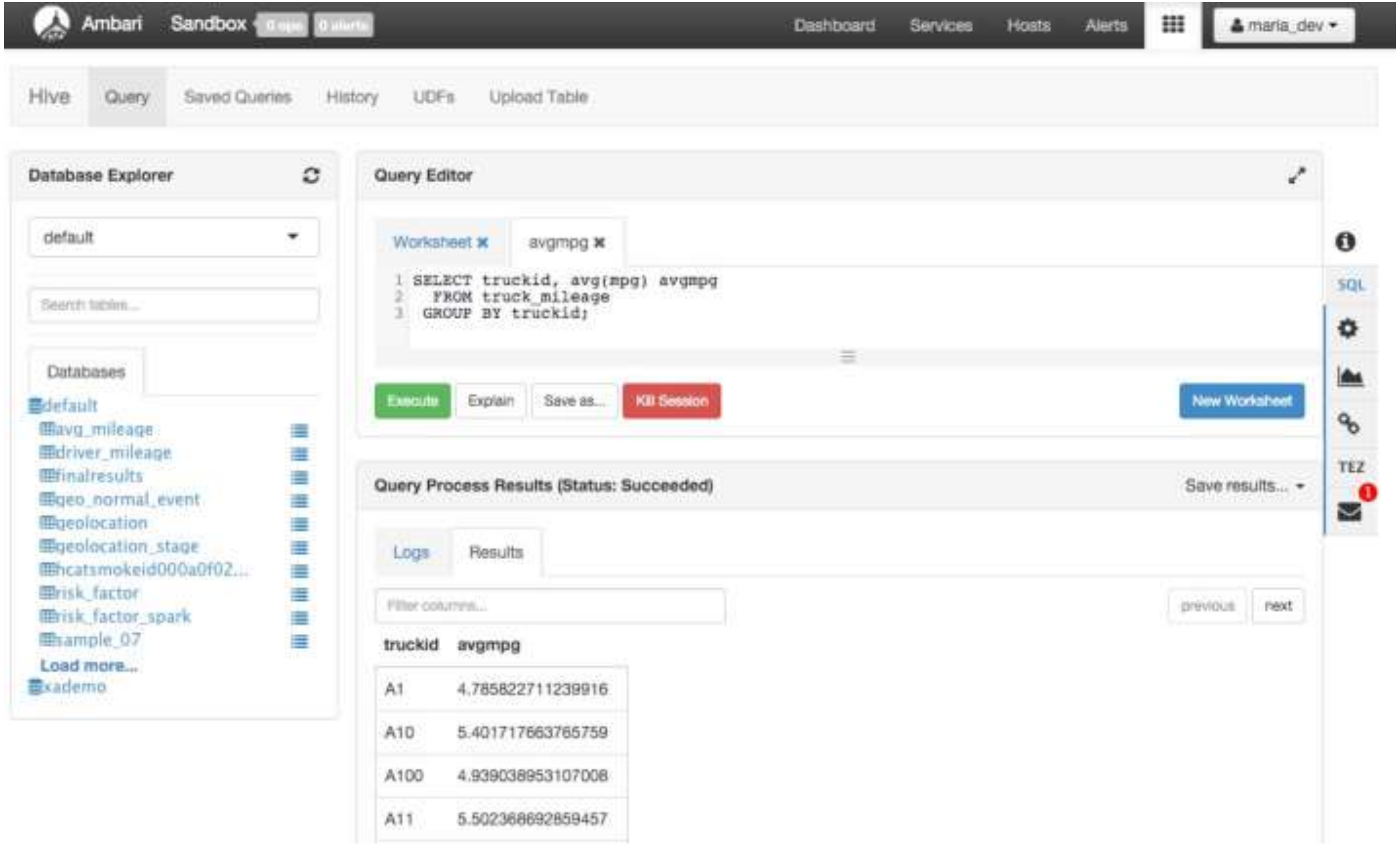
3 Query converted to YARN job and executed on Hadoop

Submitting Hive Queries – *CLI and GUI Tools*



```
[it1@sandbox ~]$ beeline -u jdbc:hive2://localhost:10000
WARNING: Use "yarn jar" to launch YARN applications.
Connecting to jdbc:hive2://localhost:10000
Connected to: Apache Hive (version 1.2.1000.2.4.0.0-169)
Driver: Hive JDBC (version 1.2.1000.2.4.0.0-169)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.1000.2.4.0.0-169 by Apache Hive
0: jdbc:hive2://localhost:10000> show tables;
+-----------------------------------+--+
|             tab_name              |
+-----------------------------------+--+
| avg_mileage                       |
| driver_mileage                    |
| finalresults                      |
| geo_normal_event                  |
| geolocation                       |
| geolocation_stage                 |
| hcatsmokeid000a0f02_date250116    |
| risk_factor                       |
| risk_factor_spark                 |
| sample_07                         |
| sample_08                         |
| truck_mileage                     |
| trucks                            |
| trucks_stage                      |
+-----------------------------------+--+
14 rows selected (0.696 seconds)
0: jdbc:hive2://localhost:10000>
```

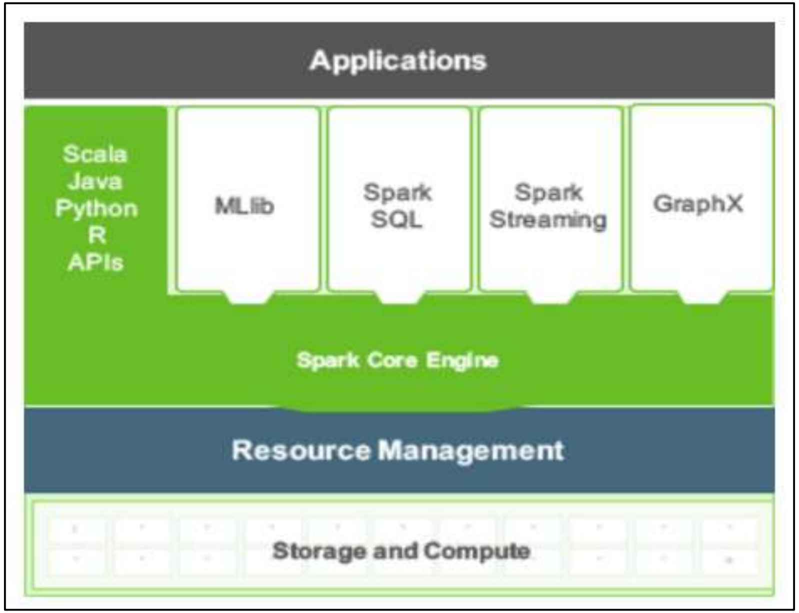
Submitting Hive Queries – *Ambari Hive View*



```
SELECT truckid, avg(mpg) avgmpg
  FROM truck_mileage
 GROUP BY truckid;
```

truckid	avgmpg
A1	4.785822711239916
A10	5.401717663765759
A100	4.939038953107008
A11	5.502368692859457

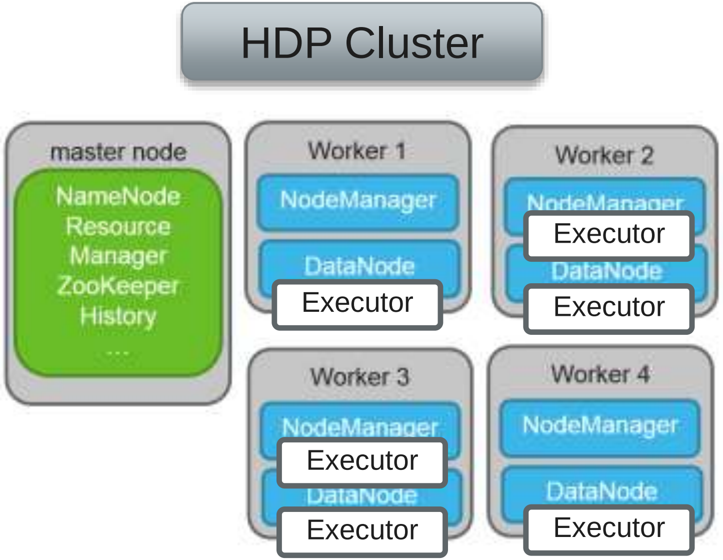
Apache Spark – http://spark.apache.org



- A data access engine for fast, large-scale data processing
- Designed for iterative in-memory computations and interactive data mining
- Provides expressive multi-language APIs for Scala, Java, R and Python
- Data workers can use built-in libraries to rapidly iterate over data for:
  - ETL
  - Machine learning
  - SQL workloads
  - Stream processing
  - Graph computations

Spark Executors & *Cluster Deployment Options*

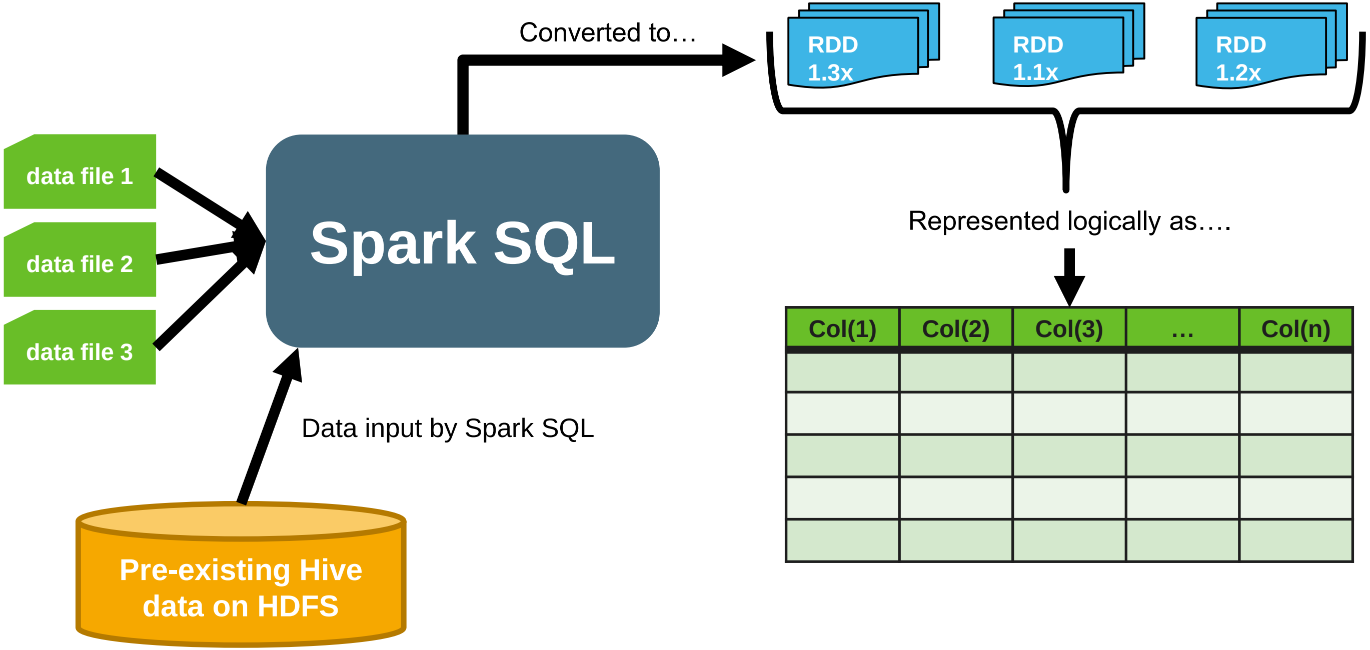
- Responsible for all application workload processing
  - The "workers" of a Spark application
  - Includes the `SparkContext` serving as the "master"
    - Schedules tasks
    - Pre-created in shells & notebooks
- **Exist for the life of the application**
- Standalone mode and cluster options
  - YARN
  - Mesos



Spark SQL Overview

- A module built on top of Spark Core
- Provides a programming abstraction for distributed processing of large-scale structured data in Spark
- Data is described as a DataFrame with rows, columns and a schema
- Data manipulation and access is available with two mechanisms
  - SQL Queries
  - DataFrames API

The DataFrame Visually



data file 1

data file 2

data file 3

Spark SQL

Pre-existing Hive data on HDFS

Data input by Spark SQL

Converted to…

RDD 1.3x

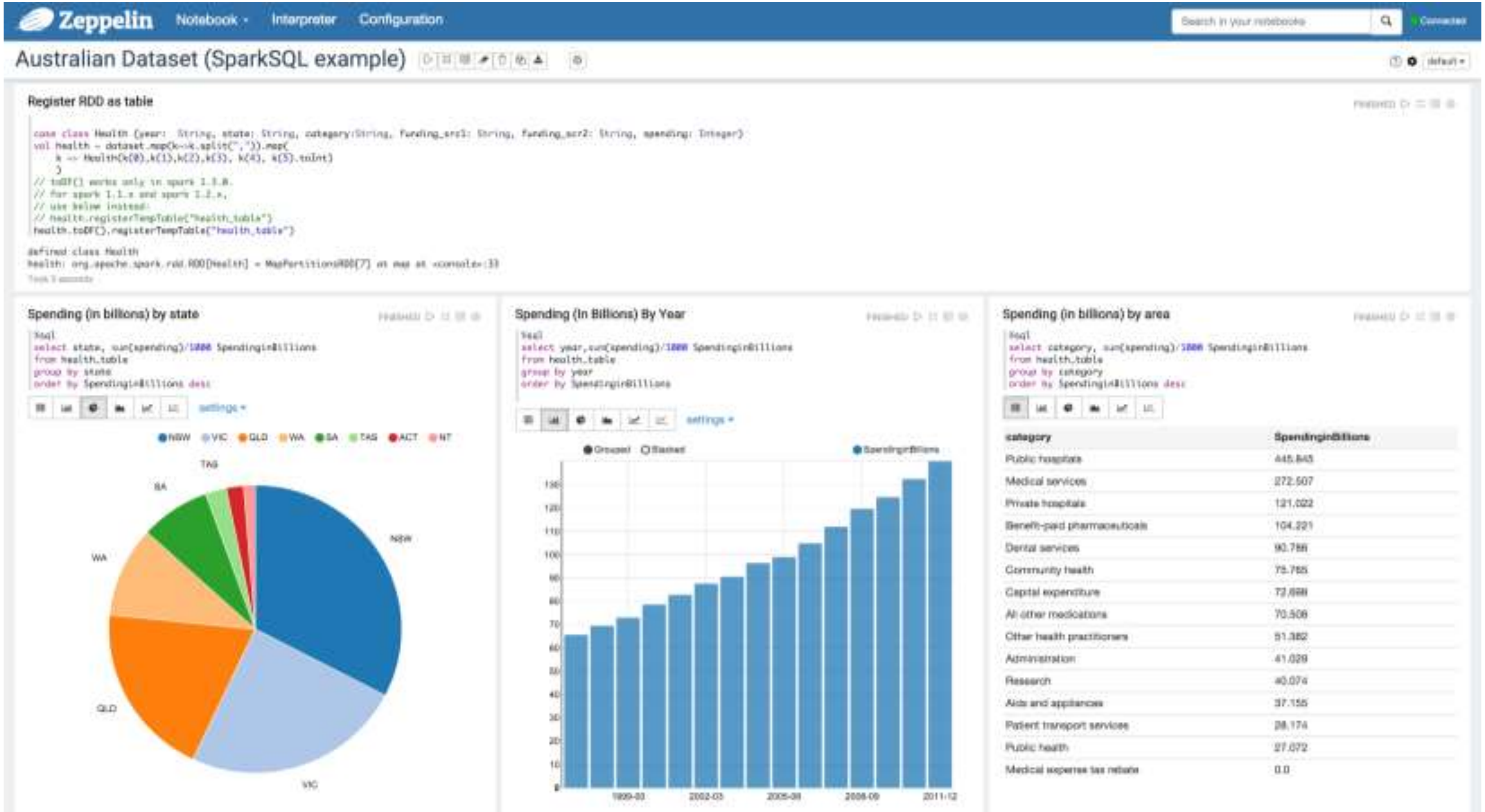
RDD 1.1x

RDD 1.2x

Represented logically as….

Col(1)	Col(2)	Col(3)	…	Col(n)

Apache Zeppelin – http://zeppelin.apache.org



Australian Dataset (SparkSQL example)

Register RDD as table

```
case class Health (year:  String, state: String, category:String, funding_src1: String, funding_scr2: String, spending: Integer)
val health = dataset.map(k=>k.split(",")).map(
    k => Health(k(0),k(1),k(2),k(3), k(4), k(5).toInt)
    )
// toDF() works only in spark 1.3.0.
// for spark 1.1.x and spark 1.2.x,
// use below instead:
// health.registerTempTable("health_table")
health.toDF().registerTempTable("health_table")
```

```
defined class Health
health: org.apache.spark.rdd.RDD[Health] = MapPartitionsRDD[7] at map at <console>:33
Took 3 seconds
```

Spending (in billions) by state

```
%sql
select state, sum(spending)/1000 SpendinginBillions
from health_table
group by state
order by SpendinginBillions desc
```

Spending (In Billions) By Year

```
%sql
select year,sum(spending)/1000 SpendinginBillions
from health_table
group by year
order by SpendinginBillions
```

Spending (in billions) by area

```
%sql
select category, sum(spending)/1000 SpendinginBillions
from health_table
group by category
order by SpendinginBillions desc
```

category	SpendinginBillions
Public hospitals	445.845
Medical services	272.507
Private hospitals	121.022
Benefit-paid pharmaceuticals	104.221
Dental services	90.786
Community health	75.765
Capital expenditure	72.698
All other medications	70.508
Other health practitioners	51.382
Administration	41.029
Research	40.074
Aids and appliances	37.155
Patient transport services	28.174
Public health	27.072
Medical expense tax rebate	0.0

Still Based on MapReduce Principles

```
sc.textFile("/some-hdfs-data") \
 .flatMap(lambda line: line.split(" ")) \
 .map(lambda line: (word, 1))) \
 .reduceByKey(lambda a,b : a+b, \
    numPartition=3) \
 .collect()
```

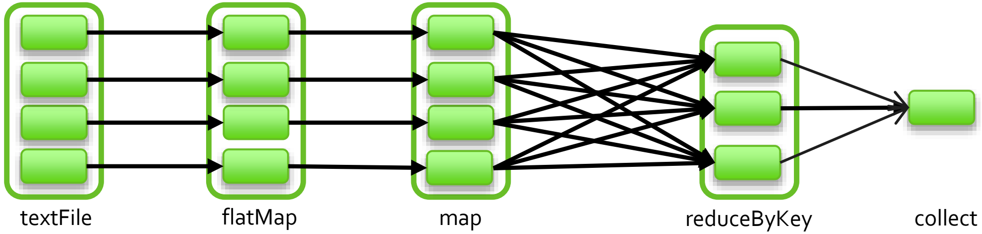
RDD[String]

RDD[List[String]]

RDD[(String, Int)]

RDD[(String, Int)]

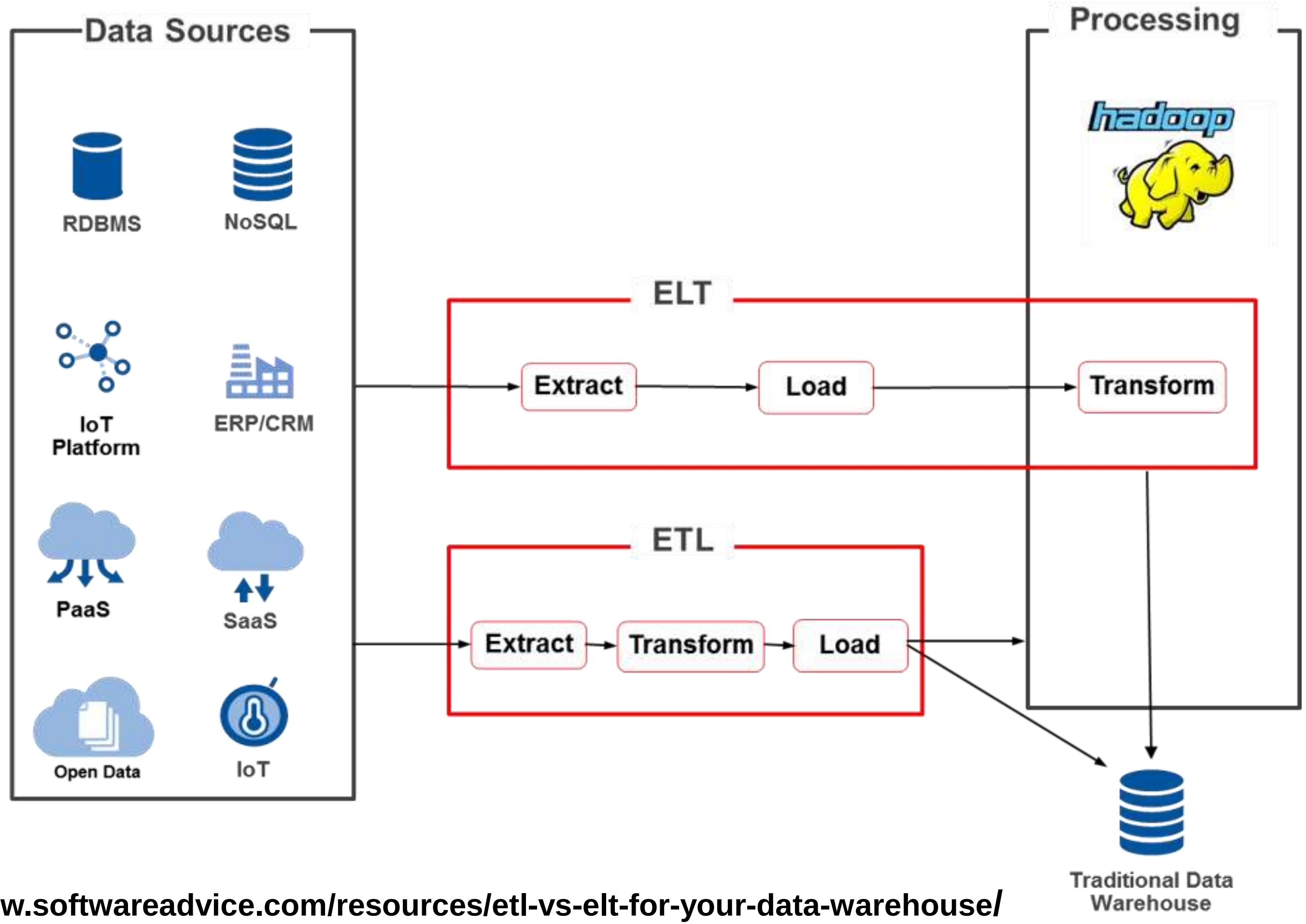
Array[(String, Int)]



ETL Requirements

- Read/write multiple file formats and persistent stores
- Source-to-target mappings
- Data profiling
- Data quality
- Common processing functionality
- Custom business rules injection
- Merging changed records
- Error handling
- Alerts / notifications
- Logging
- Lineage & job statistics
- Administration
- Reusability
- Performance & scalability
- Source code management

ETL vs ELT



Source: http://www.softwareadvice.com/resources/etl-vs-elt-for-your-data-warehouse/

File Formats

The ability to read & write many different file formats is critical

- **Delimited values (comma, tab, etc)**
- **XML**
- **JSON**
- **Avro**
- **Parquet**
- **ORC**
- **Esoteric formats such as EBCDIC and compact RYO solutions**

File Formats: Delimited Values

Delimited datasets are very common place in Big Data clusters

Simple example file: `catalog.del`

```
Programming Pig|Alan Gates|23.17|2016
Apache Hive Essentials|Dayong Du|39.99|2015
Spark in Action|Petar Zecevic|41.24|2016
```

Pig Code for Delimited File

```
book_catalog = LOAD
     '/otpc/ff/del/data/catalog.del’
  USING PigStorage('|')
  AS (title:chararray, author:chararray,
      price:float, year:int);

DESCRIBE book_catalog;
DUMP book_catalog;
```

Pig Output for Delimited File

```
book_catalog: {title: chararray,author:
chararray,price: float,year: int}


(Programming Pig,Alan Gates,23.17,2016)
(Apache Hive Essentials,Dayong
Du,39.99,2015)
(Spark in Action,Petar Zecevic,41.24,2016)
```

Hive Code for Delimited File

```
CREATE EXTERNAL TABLE book_catalog_pipe(
       title string, author string,
       price float, year int)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '|'
    STORED AS TEXTFILE
    LOCATION '/otpc/ff/del/data';
```

Hive Output for Delimited File Schema

```
desc book_catalog_pipe;


| col_name  | data_type  | comment  |
+-----------+------------+----------+--+
| title     | string     |          |
| author    | string     |          |
| price     | float      |          |
| year      | int        |          |
```

Hive Output for Delimited File Contents

```
SELECT * FROM book_catalog_pipe;


| title     | author  | price | year |
+-----------+---------+-------+------+--+
| Progra... | Alan... | 23.17 | 2016 |
| Apache... | Dayo... | 39.99 | 2015 |
| Spark ... | Peta... | 41.24 | 2016 |
```

Spark Code for Delimited File

```
val catalogRDD = sc.textFile(
    "hdfs:///otpc/ff/del/data/catalog.del")
case class Book(title: String, author:
String, price: Float, year: Int)
val catalogDF = catalogRDD
    .map(b => b.split('|'))
    .map(b => Book(b(0), b(1), b(2).toFloat,
                   b(3).toInt))
    .toDF()
```

Spark Output for Delimited File Schema

```
catalogDF.printSchema()


root
 |-- title: string (nullable = true)
 |-- author: string (nullable = true)
 |-- price: float (nullable = false)
 |-- year: integer (nullable = false)
```

Spark Output for Delimited File Contents

```
catalogDF.show()


|         title|     author|price|year|
+--------------+-----------+-----+----+
|Programming...| Alan Gates|23.17|2016|
|Apache Hive...|  Dayong Du|39.99|2015|
|Spark in Ac...|Petar Ze...|41.24|2016|
```

File Formats: XML

Simple example file: `catalog.xml`

```
<CATALOG>
  <BOOK>
    <TITLE>Programming Pig</TITLE>
    <AUTHOR>Alan Gates</AUTHOR>
    <PRICE>23.17</PRICE>
    <YEAR>2016</YEAR>
  </BOOK>
  <!-- other 2 BOOKs not shown -->
</CATALOG>
```

Pig Code for XML File

```
raw = LOAD '/otpc/ff/xml/catalog.xml'
   USING XMLLoader('BOOK') AS (x:chararray);

formatted = FOREACH raw GENERATE
         XPath(x, 'BOOK/TITLE')  AS title,
         XPath(x, 'BOOK/AUTHOR') AS author,
 (float) XPath(x, 'BOOK/PRICE')  AS price,
   (int) XPath(x, 'BOOK/YEAR')   AS year;
```

Hive Code for XML File

```
CREATE EXTERNAL TABLE book_catalog_xml(str string)
LOCATION '/otpc/ff/xml/flat';


CREATE TABLE book_catalog STORED AS ORC AS
   SELECT xpath_string(str,'BOOK/TITLE')  AS title,
          xpath_string(str,'BOOK/AUTHOR') AS author,
          xpath_float( str,'BOOK/PRICE')  AS price,
          xpath_int(   str,'BOOK/YEAR')   AS year
     FROM book_catalog_xml;
```

Spark Code for XML File

```
val df = sqlContext
          .read
          .format("com.databricks.spark.xml")
          .option("rowTag", "BOOK")
          .load("/otpc/ff/xml/catalog.xml")
```

File Formats: JSON

Simple example file: `catalog.json`

```
{"title":"Programming Pig", "author":"Alan Gates",
    "price":23.17, "year":2016}
{"title":"Apache Hive Essentials", "author":"Dayong Du",
    "price":39.99, "year":2015}
{"title":"Spring in Action", "author":"Petar Zecevic",
     "price":41.24, "year":2016}
```

Pig Code for JSON File

```
book_catalog =
  LOAD '/otpc/ff/json/data/catalog.json’
     USING JsonLoader('title:chararray,
                       author:chararray,
                       price:float,
                       year:int');
```

Hive Code for JSON File

```
CREATE EXTERNAL TABLE book_catalog_json(
        title string, author string,
        price float, year int)
    ROW FORMAT SERDE 'o.a.h.h.d.JsonSerDe’
    STORED AS TEXTFILE
    LOCATION '/otpc/ff/json/data';
```

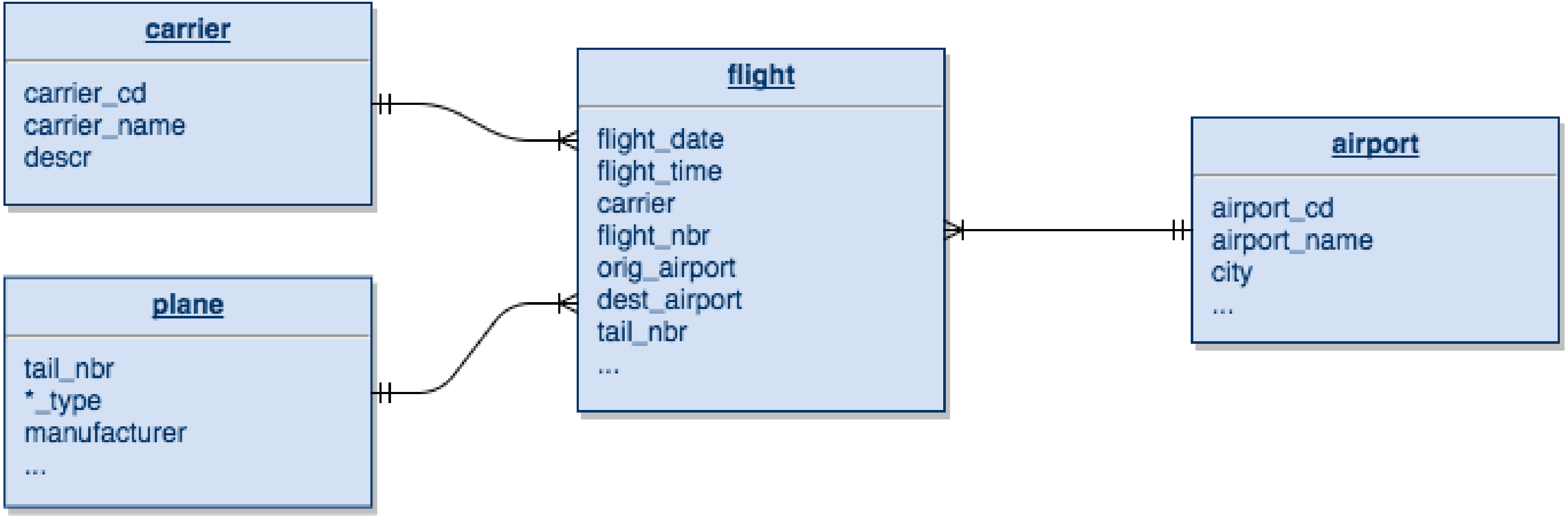
Spark Code for JSON File

```
val df = sqlContext
         .read
         .format("json")
         .load("/otpc/ff/json/catalog.json")
```

WINNER: File Formats

Data Set for Examples

Reliance on Hive Metastore for Pig and Spark SQL



Source to Target Mappings

Classic ETL need to map one dataset to another; includes these scenarios

Column Presence	Action
Source and Target	Move data from source column to target column (could be renamed, cleaned, transformed, etc)
Source, not in Target	Ignore this column
Target, not in Source	Implies a hard-coded or calculated value will be inserted or updated

Source to Target Mappings Use Case

Create new dataset from `airport_raw`

- **Change column names**
  - `airport_code` to `airport_cd`
  - `airport` to `name`
- **Carry over as named**
  - `city`, `state`, `country`
- **Exclude**
  - `latitude` and `longitude`
- **Hard-code new field**
  - `gov_agency` as `'FAA'`

Pig Code for Data Mapping

```
src_airport = LOAD 'airport_raw’
     USING o.a.h.h.p.HCatLoader();
tgt_airport = FOREACH src_airport GENERATE
  airport_code AS airport_cd,
  airport AS name, city, state, country,
  'FAA' AS gov_agency:chararray;
DESCRIBE tgt_airport;
DUMP tgt_airport;
```

Pig Output for Data Mapping

```
target_airport: {airport_cd: chararray,
name: chararray, city: chararray, state:
chararray, country: chararray, gov_agency:
chararray}
```

```
(00M,Thigpen,BaySprings,MS,USA,FAA)
(00R,LivingstonMunicipal,Livingston,TX,USA,FAA)
(00V,MeadowLake,ColoradoSprings,CO,USA,FAA)
```

Hive Code for Data Mapping

```
CREATE TABLE tgt_airport STORED AS ORC AS
  SELECT airport_code AS airport_cd,
         airport AS name,
         city, state, country,
         'FAA' AS gov_agency
    FROM airport_raw;
```

Hive Output for Mapped Schema

```
| col_name   | data_type   | comment  |
+------------+------------+----------+--+
| airport_cd | string      |          |
| name       | string      |          |
| city       | string      |          |
| state      | string      |          |
| country    | string      |          |
| gov_agency | string      |          |
```

Hive Output for Mapped Contents

```
SELECT * FROM tgt_airport;


| _cd | name    | city    | st | cny | g_a |
+-----+---------+---------+----+-----+-----+
| 00M | Thig... | BayS... | MS | USA | FAA |
| 00R | Livi... | Livi... | TX | USA | FAA |
| 00V | Mead... | Colo... | CO | USA | FAA |
```

Spark Code for Data Mapping

```
val airport_target = hiveContext
  .table("airport_raw")
  .drop("lat").drop("long")
  .withColumnRenamed("airport_code",
                     "airport_cd")
  .withColumnRenamed("airport", "name")
  .withColumn("gov_agency", lit("FAA"))
```

Spark Output for Mapped Schema

```
root
 |-- airport_cd: string (nullable = true)
 |-- name: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- country: string (nullable = true)
 |-- gov_agency: string (nullable = false)
```

Spark Output for Mapped Contents

```
airport_target.show()


|_cd|         name|         city|st|cny|g_a|
+---+-------------+-------------+--+---+---+
|00M|      Thigpen|   BaySprings|MS|USA|FAA|
|00R|Livingston...|Livingston...|TX|USA|FAA|
|00V|   MeadowLake|ColoradoSp...|CO|USA|FAA|
```

WINNER: Source to Target Mapping

Data Quality

DQ is focused on detecting/correcting/enhancing input data

- **Data type conversions / casting**
- **Numeric ranges**
- **Currency validation**
- **String validation**
  - Leading / trailing spaces
  - Length
  - Formatting (ex: SSN and phone #s)
- **Address validation / standardization**
- **Enrichment**

Numeric Validation Use Case

Validate `latitude` and `longitude` values from `airport_raw`

- **Convert them from string to float**
- **Verify these values are within normal ranges**

Attribute	Min	Max
`latitude`	-90	+90
`longitude`	-180	+180

Pig Code for Numeric Validation

```
src_airport = LOAD 'airport_raw’ USING HCatLoader();
aprt_cnvrtd = FOREACH src_airport GENERATE
    airport_cd, (float) latitude, (float) longitude;

ll_not_null = FILTER aprt_cnvrtd BY
     ( NOT ( (latitude IS NULL) OR
             (longitude IS NULL) ) );
valid_airports = FILTER ll_not_null BY
    (latitude <= 70) AND (longitude >= -170);
```

Hive Code for Numeric Validation

```
CREATE TABLE airport_stage STORED AS ORC AS
    SELECT airport_code,
           CAST(latitude AS float),
           CAST(longitude AS float)
      FROM airport_raw;
CREATE TABLE airport_final STORED AS ORC AS
    SELECT * FROM airport_stage
     WHERE latitude  BETWEEN  -80 AND 70
       AND longitude BETWEEN -170 AND 180;
```

Spark Code for Numeric Validation

```
val airport_validated = hiveContext
  .table("airport_raw")
  .selectExpr("airport_code",
        "cast(latitude as float) latitude",
        "cast(longitude as float) longitude")
  .filter("latitude is not null")
  .filter("longitude is not null")
  .filter("latitude <= 70")
  .filter("longitude >= -170")
```

String Validation Use Case

Validate `city` values from `airport_raw`

- **Trim any leading / trailing spaces**
- **Truncate any characters beyond the first 30**

Pig Code for String Validation

```
src_airport = LOAD 'airport_raw’
     USING HCatLoader();


valid_airports = FOREACH src_airport
  GENERATE
    airport_code, airport,
    SUBSTRING(TRIM(city),0,29) AS city,
    state, country;
```

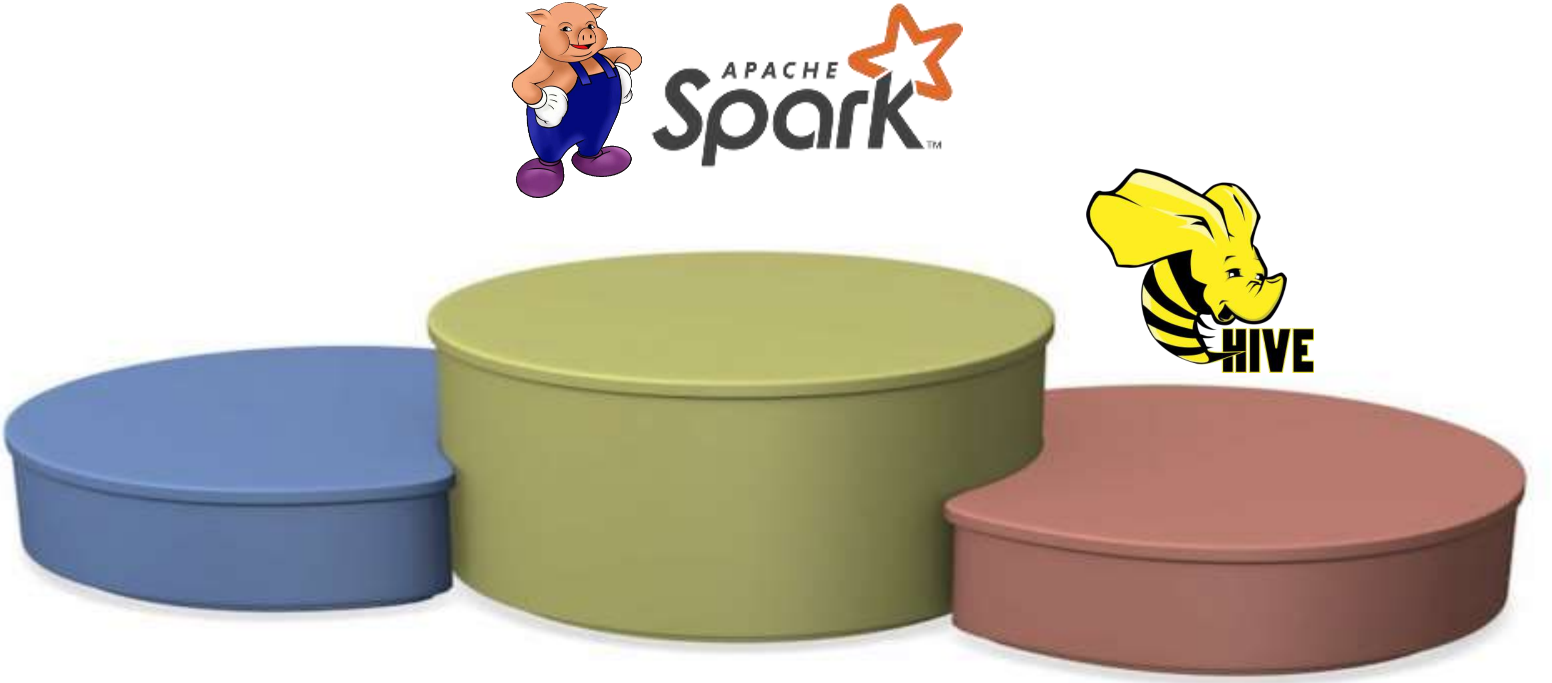
Hive Code for String Validation

```
CREATE TABLE airport_final STORED AS ORC AS
    SELECT airport_code, airport,
           SUBSTR(TRIM(city),1,30) AS city,
           state, country
      FROM airport_raw;
```

Spark Code for String Validation

```
val airport_validated = hiveContext
  .table("airport_raw")
  .withColumnRenamed("city", "city_orig")
  .withColumn("city", substring(
                  trim($"city_orig"),1,30))
  .drop("city_orig")
```

WINNER: Data Quality



Data Profiling

Technique used to examine data for different purposes such as determining accuracy and completeness – drives DQ improvements

- **Numbers of records – including null counts**
- **Avg / max lengths**
- **Distinct values**
- **Min / max values**
- **Mean**
- **Variance**
- **Standard deviation**

Data Profiling with Pig

Coupled with Apache DataFu generates statistics such as the following

```
Column Name: sales_price

Row Count: 163794

Null Count: 0             Total Value: 21781793

Distinct Count: 1446      Mean Value: 132.98285040966093

Highest Value: 70589      Variance: 183789.18332067598

Lowest Value: 1           Standard Deviation: 428.7064069041609
```

Data Profiling with Hive

Column-level statistics for all data types

```
|col_name|min|max|nulls|dist_ct|
+--------+---+---+-----+-------+
|air_time|  0|757|    0|    316|
```

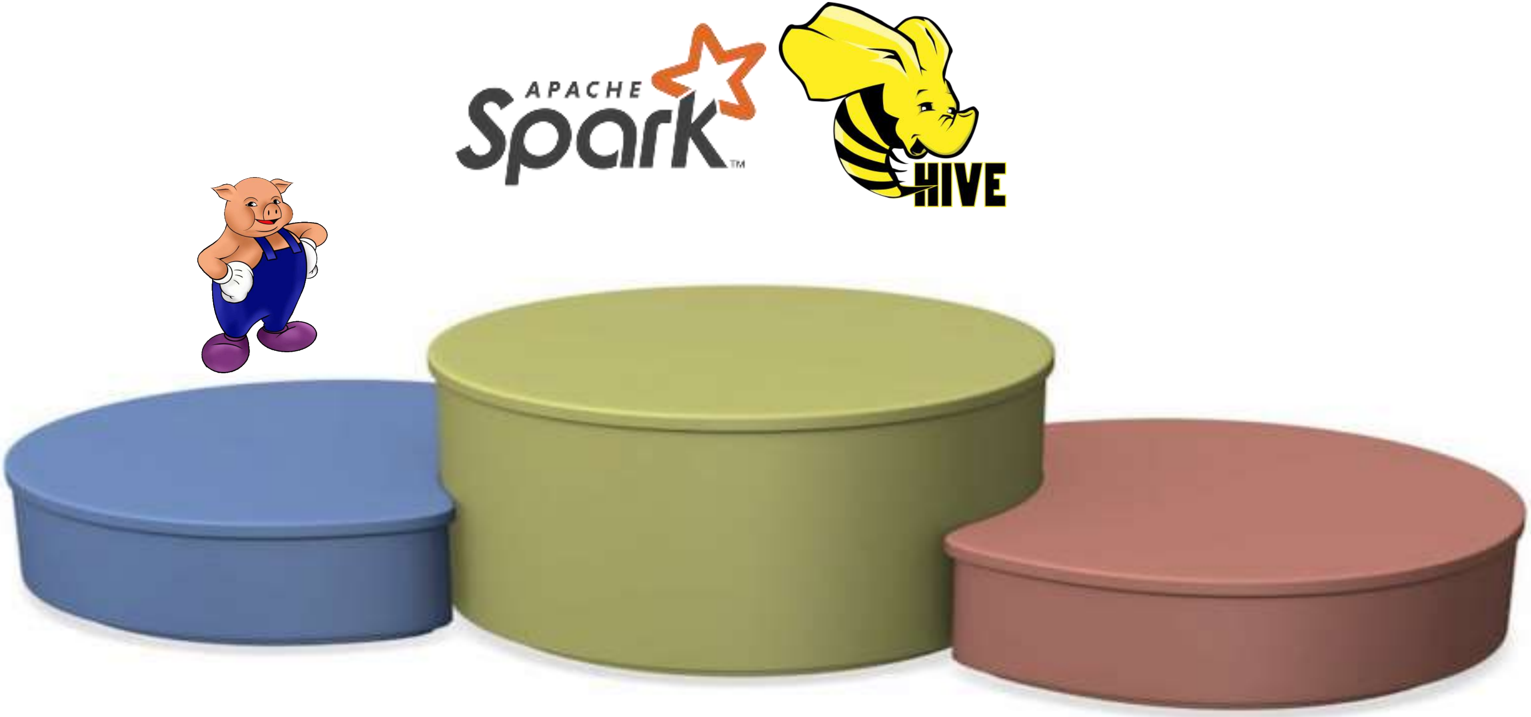
```
|col_name|dist_ct|avgColLn|mxColLn|
+--------+-------+--------+-------+
| city   |   2535|   8.407|     32|
```

Data Profiling with Spark

Inherent statistics for numeric data types

```
|summary|         air_time|
+-------+-----------------+
|  count|          2056494|
|   mean|103.9721783773743|
| stddev|67.42112792270458|
|    min|                0|
|    max|              757|
```

WINNER: Data Profiling



Core Processing Functionality

Expected features to enable data transformation, cleansing & enrichment

- **Filtering / splitting**
- **Sorting**
- **Lookups / joining**
- **Union / distinct**
- **Aggregations / pivoting**
- **SQL support**
- **Analytical functions**

Filtering Examples; Pig, Hive & Spark

```
tx_arprt = FILTER arprt BY state == 'TX';
```

```
SELECT * FROM arprt WHERE state = 'TX';
```

```
val txArprt = hiveContext
              .table("arprt")
              .filter("state = 'TX'")
```

Sorting Examples; Pig, Hive & Spark

```
srt_flight = ORDER flight BY dep_delay DESC,
     unique_carrier, flight_num;
```

```
SELECT * FROM flight ORDER BY dep_delay DESC,
     unique_carrier, flight_num;
```

```
val longestDepartureDelays = hiveContext
    .table("flight").sort($"dep_delay".desc,
              $"unique_carrier", $"flight_num")
```

Joining with Pig

```
jnRslt = JOIN flights BY tail_num,
               planes  BY tail_number;
prettier = FOREACH with_year GENERATE
    flights::flight_date AS flight_date,
    flights::tail_num    AS tail_num,
    -- plus other 17 "flights" attribs
    planes::year         AS plane_built;
    -- ignore other 8 "planes" attribs
```

Joining with Hive

```
SELECT F.*,
       P.year AS plane_built
  FROM flight F,
       plane P
 WHERE F.tail_num = P.tail_number;
```

Joining with Spark

```
val flights = hiveContext.table("flight")
val planes = hiveContext.table("plane")
  .select("tail_number", "year")
  .withColumnRenamed("year", "plane_built")
val augmented_flights = flights
  .join(planes)
  .where($"tail_num" === $"tail_number")
  .drop("tail_number")
```

Pig Code for Distinct

```
planes  = LOAD 'plane' USING HCatLoader();
rotos = FILTER planes BY
    aircraft_type == 'Rotorcraft';
makers = FOREACH rotos GENERATE
    manufacturer;

distinct_makers = DISTINCT makers;
DUMP distinct_rotor_makers;
```

Pig Output for Distinct

```
(BELL)

(SIKORSKY)

(AGUSTA SPA)

(AEROSPATIALE)

(COBB INTL/DBA ROTORWAY INTL IN)
```

Hive Code for Distinct

```
SELECT DISTINCT(manufacturer)
  FROM plane
 WHERE aircraft_type = 'Rotorcraft';
```

Hive Output for Distinct

```
| manufacturer                    |
+---------------------------------+
| AEROSPATIALE                    |
| AGUSTA SPA                      |
| BELL                            |
| COBB INTL/DBA ROTORWAY INTL IN  |
| SIKORSKY                        |
```

Spark Code for Distinct

```
val rotor_makers = hiveContext
     .table("plane")
     .filter("aircraft_type = 'Rotorcraft'")
     .select("manufacturer")
     .distinct()
```

Spark Output for Distinct

```
|        manufacturer|
+--------------------+
|                BELL|
|            SIKORSKY|
|          AGUSTA SPA|
|        AEROSPATIALE|
|COBB INTL/DBA ROT...|
```

Pig Code for Aggregation

```
flights = LOAD 'flight' USING HCatLoader();
reqd_cols = FOREACH flights GENERATE
    origin, dep_delay;
by_orig = GROUP reqd_cols BY origin;
avg_delay = FOREACH by_orig GENERATE
    group AS origin,
    AVG(reqd_cols.dep_delay) AS avg_dep_delay;
srtd_delay = ORDER avg_delay BY avg_dep_delay DESC;
top5_delay = LIMIT srtd_delay 5;
DUMP top5_delay;
```

Pig Output for Aggregation

```
(PIR,49.5)

(ACY,35.916666666666664)

(ACK,25.558333333333334)

(CEC,23.40764331210191)

(LMT,23.40268456375839)
```

Hive Code for Aggregation

```
SELECT origin,
       AVG(dep_delay) AS avg_dep_delay
  FROM flight
 GROUP BY origin
 ORDER BY avg_dep_delay DESC
 LIMIT 5;
```

Hive Output for Aggregation

```
| origin  |    avg_dep_delay    |
+---------+---------------------+
| PIR     | 49.5                |
| ACY     | 35.916666666666664  |
| ACK     | 25.558333333333334  |
| CEC     | 23.40764331210191   |
| LMT     | 23.40268456375839   |
```

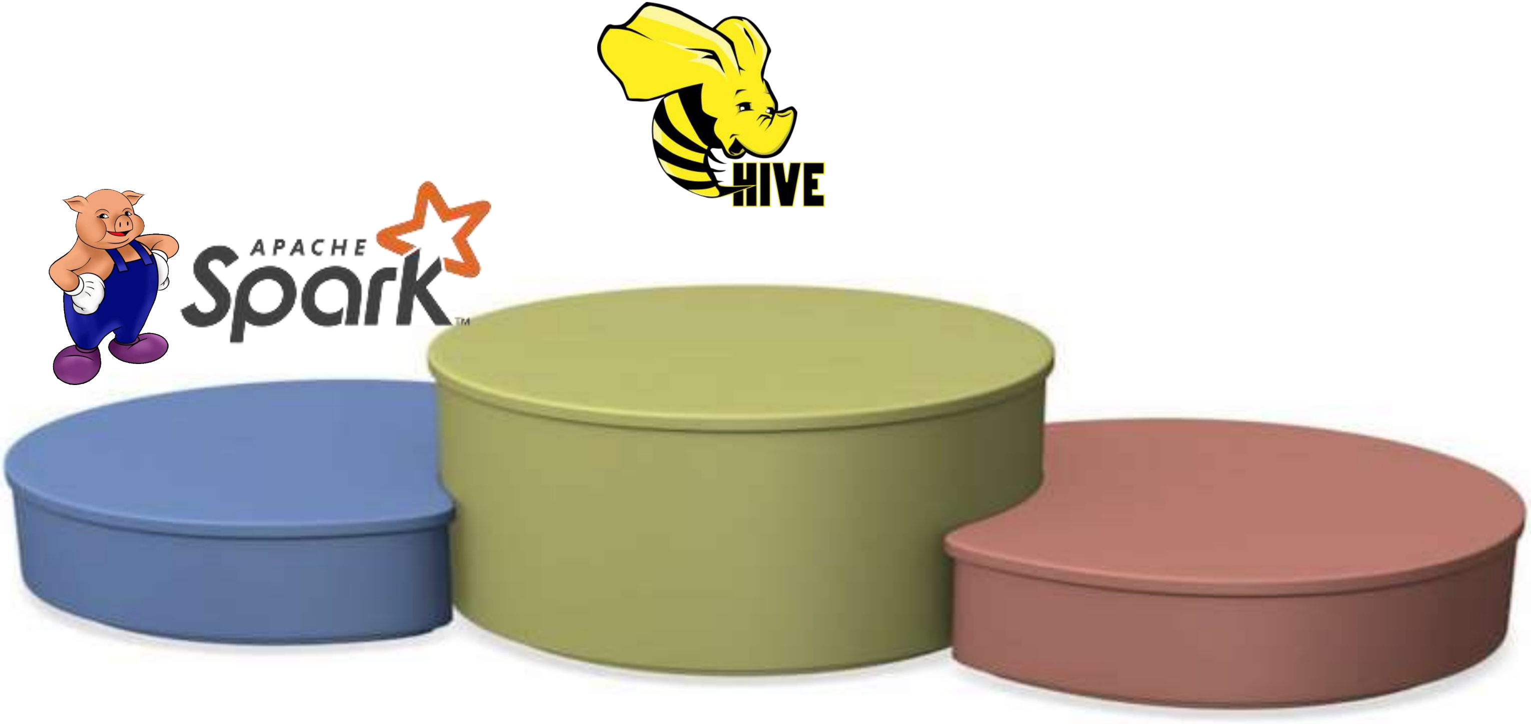
Spark Code for Aggregation

```
val sorted_orig_timings = hiveContext
   .table("flight")
   .select("origin", "dep_delay")
   .groupBy("origin").avg()
   .withColumnRenamed("avg(dep_delay)",
                      "avg_dep_delay")
   .sort($"avg_dep_delay".desc)
sorted_orig_timings.show(5)
```

Spark Output for Aggregation

```
|origin|     avg_dep_delay|
+------+------------------+
|   PIR|              49.5|
|   ACY|35.916666666666664|
|   ACK|25.558333333333334|
|   CEC| 23.40764331210191|
|   LMT| 23.40268456375839|
```

WINNER: Core Processing Functionality



Custom Business Logic

Implemented via User Defined Functions (UDF)

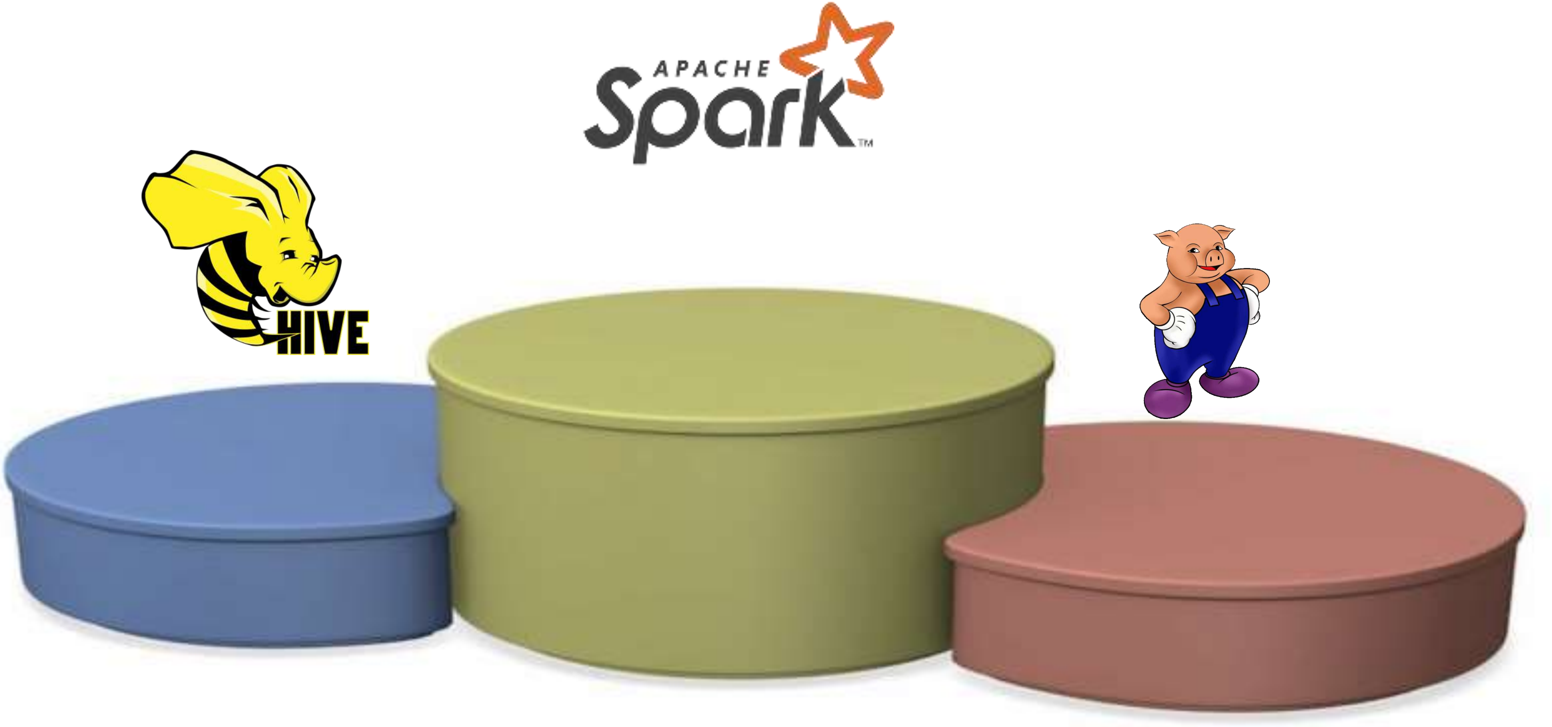
- **Pig and Hive**
  - Write Java and compile to a JAR
  - Register JAR
- **Hive can administratively pre-register UDFs at the database level**
- **Spark can wrap functions at runtime**

```
+----------+-------+
|      date|product|
+----------+-------+
|2015-03-12|toaster|
|2015-04-12|   iron|
|2014-12-31| fridge|
|2015-02-03|    cup|
+----------+-------+
```

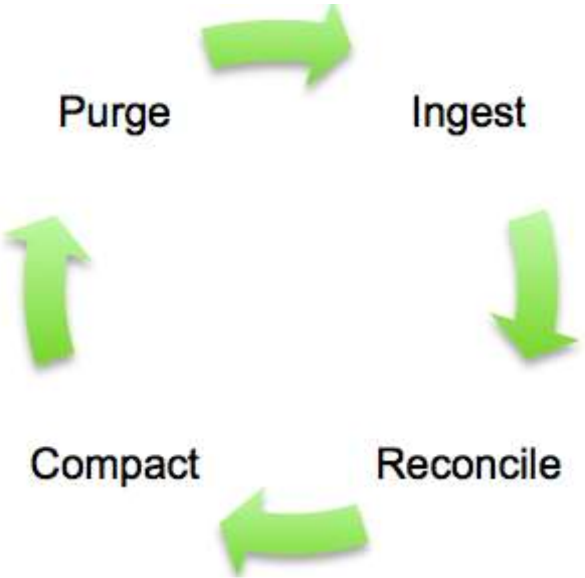
```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

get_year = udf(lambda x: int(x[:4]), IntegerType())

df1.select(get_year(df1["date"]).alias("year"),

                                 df1["product"])

       .collect()
```

```
+----+-------+
|year|product|
+----+-------+
|2015|toaster|
|2015|   iron|
|2014| fridge|
|2015|    cup|
+----+-------+
```

WINNER: Custom Business Logic



Mutable Data – *Merge & Replace*



Ingest – bring over the incremental data

Reconcile – perform the merge

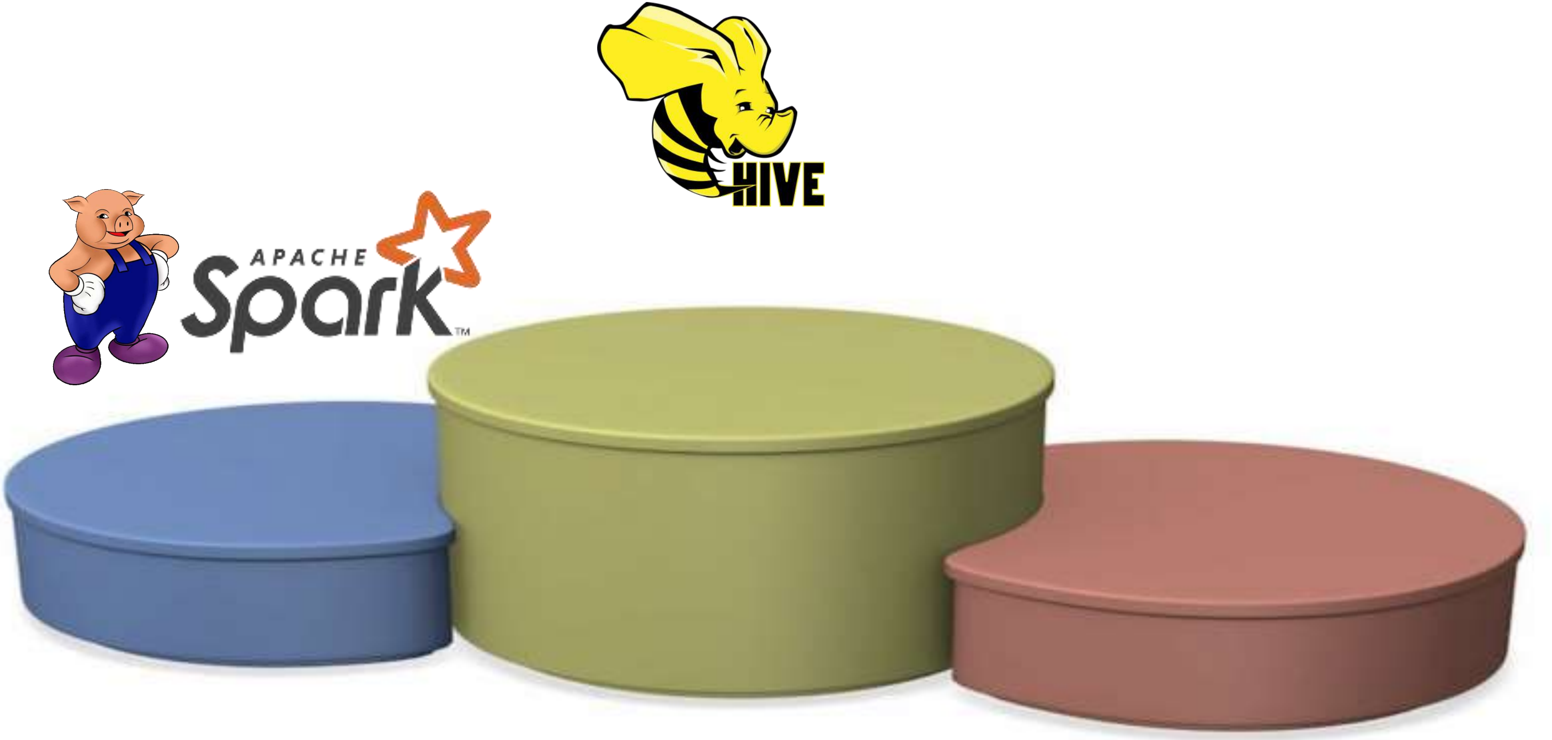
Compact – replace the existing data with the newly merged content

Purge – cleanup & prepare to repeat

See my preso and video on this topic

- http://www.slideshare.net/lestermartin/mutable-data-in-hives-immutable-world
- https://www.youtube.com/watch?v=EUz6Pu1lBHQ

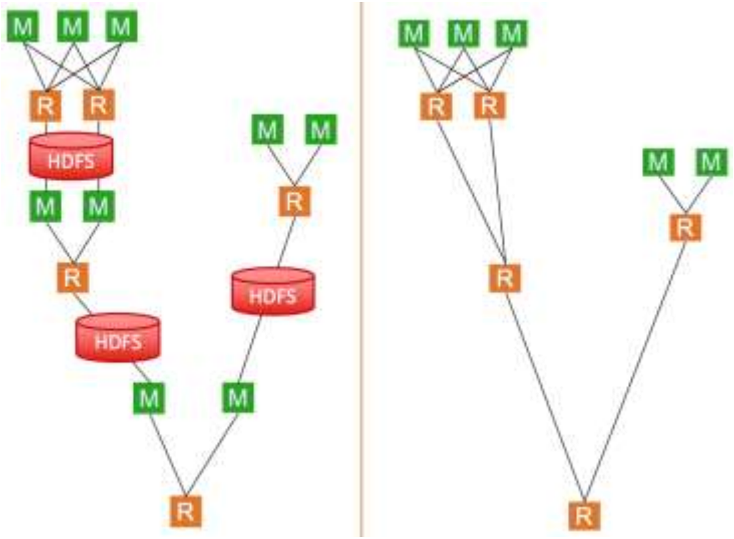
WINNER: Mutable Data



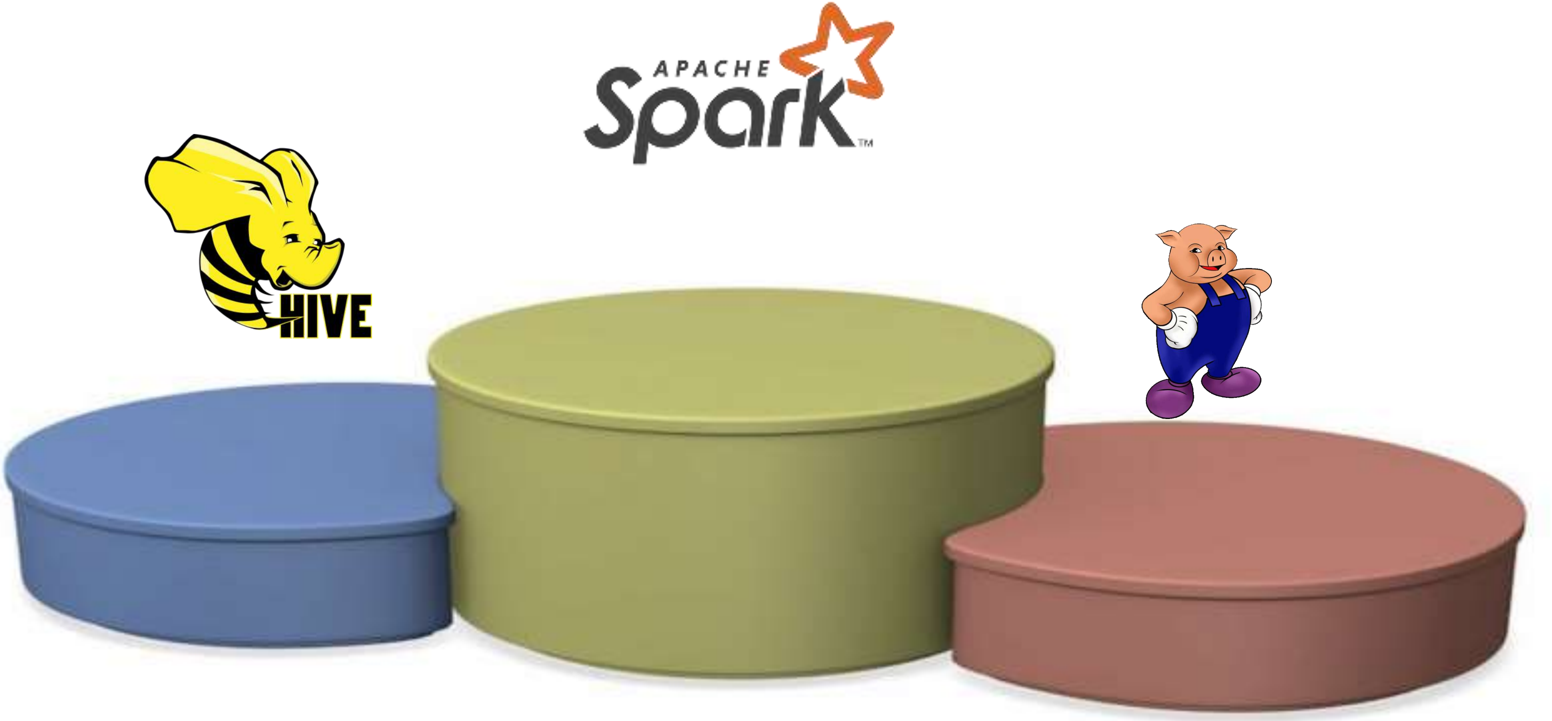
Performance

Scalability is based on size of cluster

- **Tez and Spark has MR optimizations**
  - Can link multiple maps and reduces together without having to write intermediate data to HDFS
  - Every reducer does not require a map phase
- **Hive and Spark SQL have query optimizers**
- **Spark has the edge**
  - Caching data to memory can avoid extra reads from disk
  - **Resources dedicated for entire life of the application**
  - Scheduling of tasks from 15-20s to 15-20ms



WINNER: Performance



Recommendations

Review ALL THREE frameworks back at “your desk”

Decision Criteria…

- Existing investments
- Forward-looking beliefs
- Adaptability & current skills
- It’s a “matter of style”
- Polyglot programming is NOT a bad thing!!

Share your findings via blogs and local user groups

Questions?

Lester Martin – Hadoop/Spark Trainer & Consultant

lester.martin@gmail.com

http://lester.website *(links to blog, twitter, github, LI, FB, etc)*

THANKS FOR YOUR TIME!!