Module - 7

What is HIVE?

What is Hive?

- Data Warehouse system built on top of Hadoop
  - Takes advantage of Hadoop distributed processing power

- Facilitates easy data summarization, ad-hoc queries, analysis of large datasets stored in Hadoop

- Hive provides a SQL interface (HQL) for data stored in Hadoop
  - Familiar, Widely known syntax
  - Data Definition Language and Data Manipulation Language

- HQL queries implicitly translated to one or more Hadoop MapReduce job(s) for execution
  - Saves you from having to write the MapReduce programs!
  - Clear separation of defining the *what* (you want) vs. the *how* (to get it)

- Hive provides mechanism to project structure onto Hadoop datasets
  - Catalog ("metastore") maps file structure to a tabular form

What HIVE is NOT….

What Hive is not…

- **Hive is not a full database - but it fits alongside your RDBMS.**
- Is not a real-time processing system
  - Best for heavy analytics and large aggregations – Think Data Warehousing.
  - Latencies are often much higher than RDBMS
  - Schema on Read
    - Fast loads and flexibility – at the cost of query time
    - Use RDBMS for fast run queries.
- Not SQL-92 compliant
  - Does not provide row level inserts, updates or deletes
  - Doesn't support transactions
  - Limited subquery support
- Query optimization still a work in progress
- See HBase for rapid queries and row-level updates and transactions

Real World use cases

Real world use cases

- **CNET: "We use Hive for data mining, internal log analysis and ad hoc queries."**
- **Digg: "We use Hive for data mining, internal log analysis, R&D, and reporting/analytics."**
- **Grooveshark: "We use Hive for user analytics, dataset cleaning, and machine learning R&D."**
- **Papertrail: "We use Hive as a customer-facing analysis destination for our hosted syslog and app log management service."**
- **Scribd: "We use hive for machine learning, data mining, ad-hoc querying, and both internal and user-facing analytics."**
- **VideoEgg: "We use Hive as the core database for our data warehouse where we track and analyze all the usage data of the ads across our network."**

Hive vs JAVA & PIG

Hive versus Java and Pig

Java

- Word Count MapReduce example
  - Lists words and number of occurrences in a document
  - Takes 63 lines of Java code to write this.
  - Hive solution only takes 7 easy lines of code!

Pig

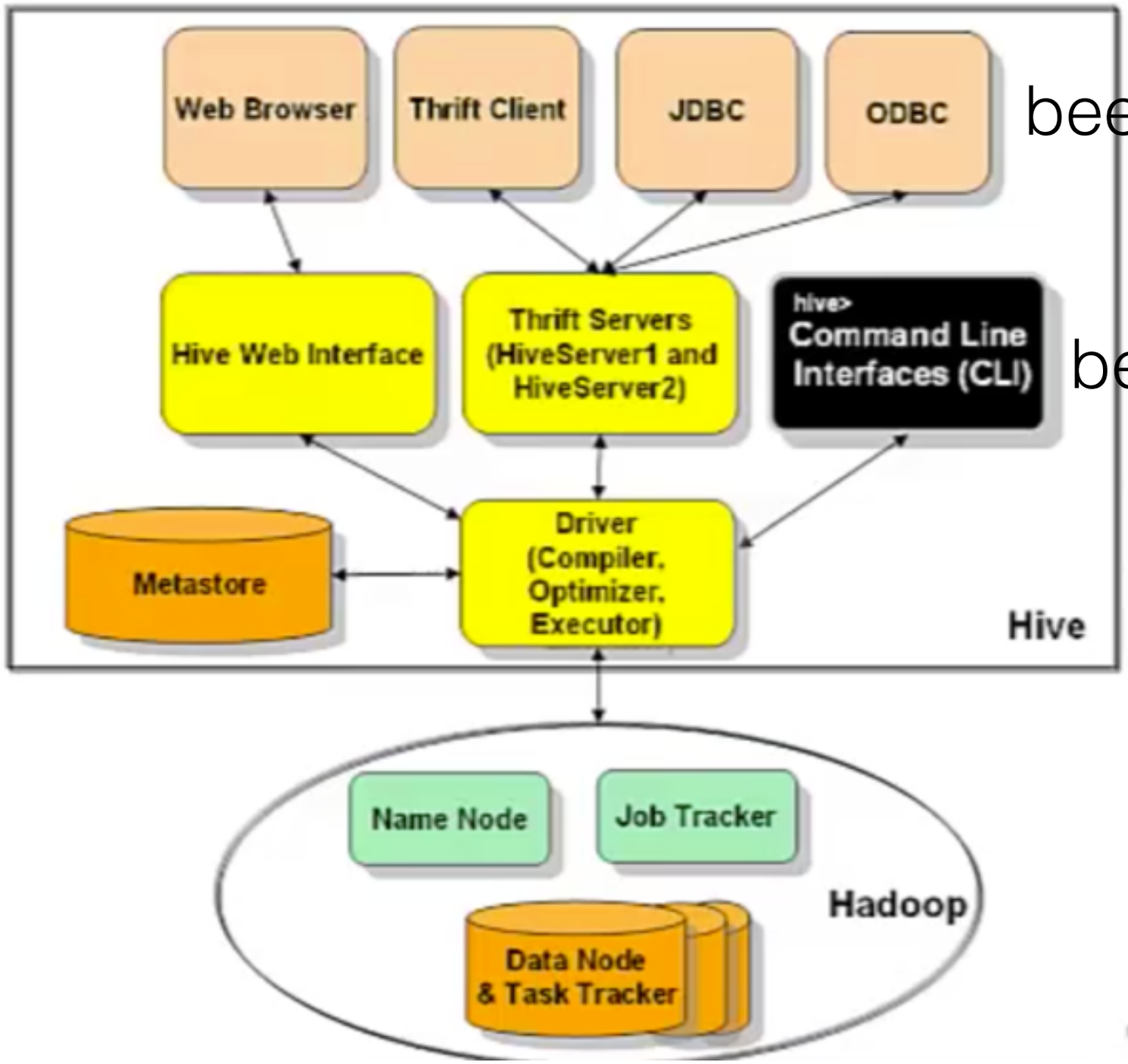
- High level programming language ("data flow language")
  - Higher learning curve for SQL programmers
- Good for ETL, not as good for ad-hoc querying
- Powerful transformation capabilities
- Often used in combination with Hive

Hive Directory Structure

Hive Directory Structure

- Lib directory
  - $HIVE_HOME/lib
  - Location of Hive JAR files
  - Contain the actual Java code that implement the Hive functionality
- Bin directory
  - $HIVE_HOME/bin
  - Location of Hive Scripts/Services
- Conf directory
  - HIVE_HOME/conf
  - Location of configuration files

Hive Components



beeline-connect

beeline-connect

Hive Data Units

Hive Data Units

- Organization of Hive data (order of granularity)

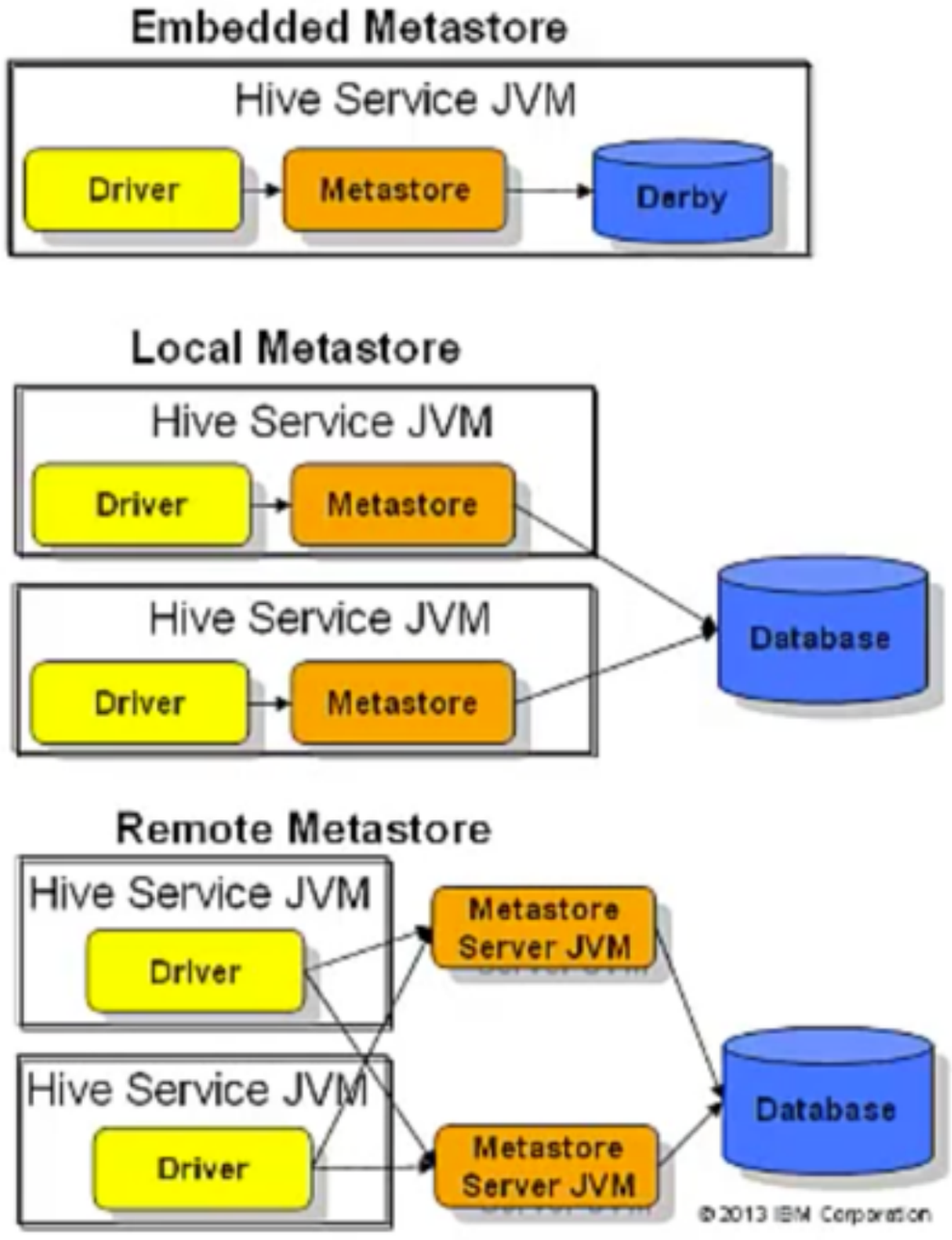
```
Database
  ->Table
    ->Partition
      ->Buckets
```

  - **Databases**: Namespaces that separate tables and other data units from naming confliction.
  - **Tables**: Homogeneous units of data which have the same schema.
  - **Partitions**: A virtual column which defines how data is stored on the file system based on its values. Each table can have one or more partitions (and one or more levels of partition).
  - **Buckets** (or **Clusters**): In each partition, data can be divided into buckets based on the hash value of a column in the table (useful for sampling, join optimization).
  - Note that it is not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.

Hive Metastore

Metastore

- 2 pieces – Service & Datastore
- Stores Hive Metadata in 1 of 3 configs:
  - **Embedded:** in-process metastore, in-process database
  - **Local:** in-process metastore, out-of-process database
  - **Remote:** out-of-process metastore, out-of-process database
- If metastore not configured - Derby database is used
  - Derby metastore allows only one user at a time
- Can be configured to use a wide variety of storage options (DB2, MySQL, Oracle, XML files, etc.) for more robust metastore



Complex Data Types

Complex Data Types

- Complex Types can be built up from primitive types and other composite types.
- **Arrays** - Indexable lists containing elements of the same type.
  - Format: ARRAY<data_type>
  - Literal syntax example: array('user1', 'user2')
  - Accessing Elements: [n] notation where n is an index into the array. E.g. *arrayname[0]*

- **Structs** - Collection of elements of different types.
  - Format: STRUCT<col_name : data_type, ...>
  - Literal syntax example: struct('Jake', '213')
  - Accessing Elements: DOT (.) notation. E.g. *structname.firstname*

- **Maps** – Collection of key-value tuples.
  - Format: MAP<primitive_type, data_type>
  - Literal syntax example: map('business_title', 'CEO', 'rank', '1')
  - Accessing Elements: ['element name'] notation. E.g. *mapname['business_title']*

- **Union** – at any one point can hold exactly one of their specified data types.
  - Format: UNIONTYPE<data_type, data_type, ...>
  - Accessing Elements: Use the create_union UDF (see Hive docs for more info)

Hive - CLI

CLI (Command Line Interface)

- Most common way to interact with Hive
- From the shell you can
  - Perform queries, DML, and DDL
  - View and manipulate table metadata
  - Retrieve query explain plans (execution strategy)

Beeline CLI

- The Hive Beeline shell and original CLI are located in **$HIVE_HOME**/bin/hive

Original Hive CLI

```
$ $HIVE_HOME/bin/hive
2013-01-14 23:36:52.153 GMT : Connection obtained for host: master-
Logging initialized using configuration in file:/opt/ibm/biginsight
Hive history file=/var/ibm/biginsights/hive/query/biadmin/hive_job

hive> show tables;
mytab1
mytab2
mytab3
OK
Time taken: 2.987 seconds
hive> quit;
```

Hive commands

Database Show/Describe

- List the Database(s) in the Hive system

```
hive> SHOW DATABASES;
```

- Show some basic information about a Database

```
hive> DESCRIBE DATABASE mydatabase;
```

- Show more detailed information about a Database

```
hive> DESCRIBE DATABASE EXTENDED mydatabase;
```

Few Examples

```
CREATE  TABLE IF NOT EXISTS employee1 (eid int, name String, salary
String, destination String)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ","
LINES TERMINATED BY "\n"
STORED AS TEXTFILE;
```

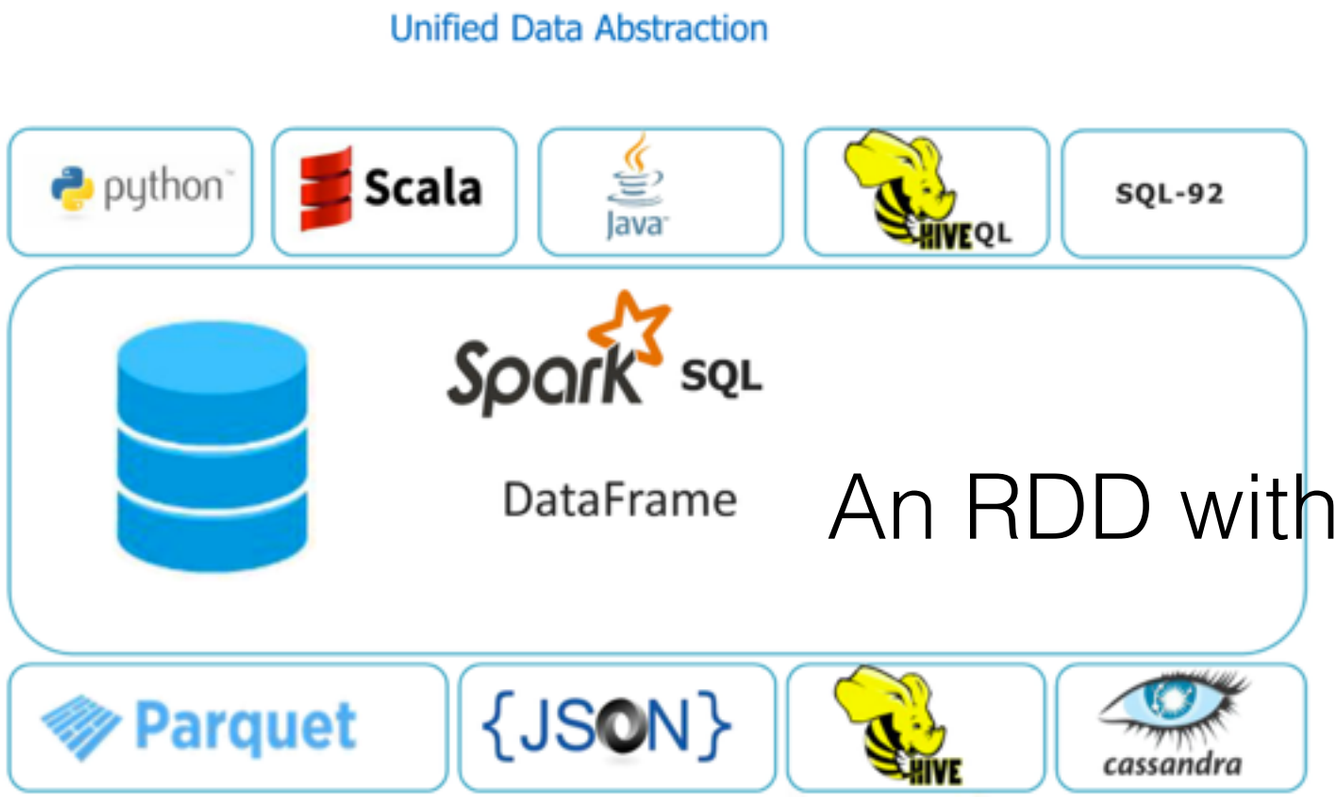
```
LOAD DATA LOCAL INPATH './foo.txt' OVERWRITE INTO TABLE employee1;
```

```
select * from employee;
select * from employee where salary > 30000;
```

SPARK SQL

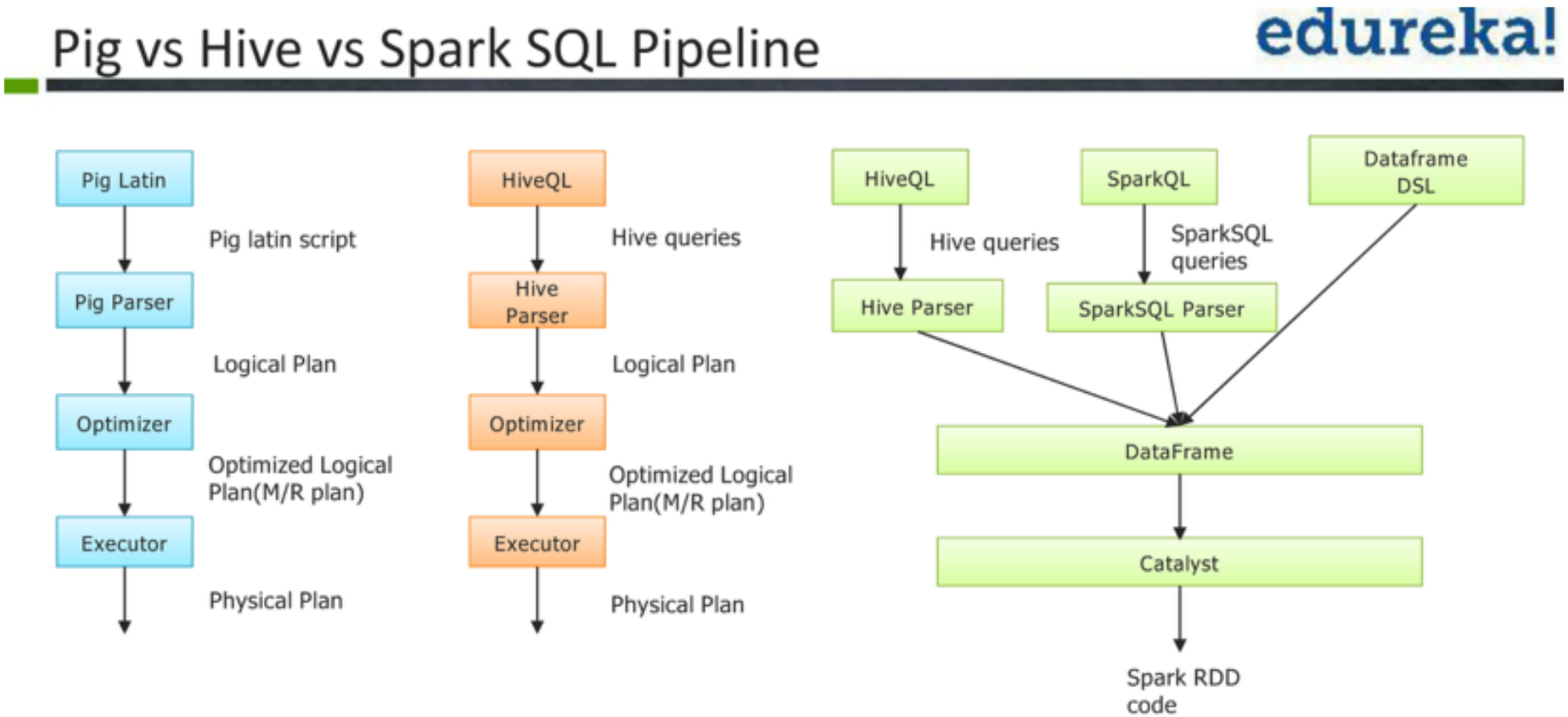
Spark SQL

edureka!



An RDD with Schema

PIG vs HIVE vs SPARK QL



SQL Context & Hive Context

→ Entry point for all SQL functionality

→ Wraps/extends existing SparkContext

SQL Context :

```
scala> import org.apache.spark.sql._
scala> var sqlContext = new SQLContext(sc)
                        or
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

HiveContext:

```
scala> import org.apache.spark.sql.hive._
scala> val hc = new HiveContext(sc)
                      or
scala> val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

DATAFRAMES

→ Distributed collection of structured data in Spark

→ DataFrame = RDD + Schema -> SchemaRDD

→ Similar to tables in RDBMS and data frames in R/Python

→ Sources can be structured data files, tables in Hive, external databases, or existing RDDs

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json("file:///home/edureka/spark-1.5.2/examples/src/main/resources/people.json")
```

```
df.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

```
val people = sc.textFile("file:///home/edureka/spark-1.5.2/examples/src/main/resources/people.txt").toDF

people.show()
```

```
+-----------+
|         _1|
```

Creating DataFrames

Creating Dataframe from RDD

```
val peopledf = sc.textFile("file:///home/edureka/spark-1.5.2/examples/
src/main/resources/people.txt").toDF
```

Creating Dataframe from JSON file

```
val peoplejsondf = sqlContext.read.json("file:///home/edureka/
spark-1.5.2/examples/src/main/resources/people.json")
```

```
val employeejsondf = sqlContext.read.json("file:///home/edureka/
employee.json")
```

Print Schema of a Dataframe

```
employeejsondf.printSchema()
```

Register the Dataframe as Table

```
employeejsondf.registerTempTable("employee")
```

Creating DataFrames

Run SQL Queries on the Json File

```
val emp_data = sqlContext.sql("Select name, address.city,
address.state FROM employee")

emp_data.collect().foreach(println)
```

```
val state = sqlContext.sql("SELECT name, address.city from employee
where address.state = ‘California'")

state.collect().foreach(println)
```

Creating DataFrames

Playing with Parquet Files

employeejsondf.saveAsParquetFile("file:///home/edureka/emp.parquet")

val parquetfile = sqlContext.parquetFile(“file:///home/edureka/emp.parquet")

parquetfile.registerTempTable("parquetfile")

val par_state = sqlContext.sql("SELECT name, address.city from parquetfile where address.state = ‘California’")

val par_state = sqlContext.sql("SELECT name, address.city from parquetfile where address.state = ‘California'")
par_state.collect().foreach(println)

par_state.saveAsParquetFile(“hdfs://localhost:8020/Parquet_Employee”)

1) Create a dataframe : sc.textFile().toDF (from RDD)
sqlContext.read.json()
2) registerTempTable(“employee”)
3) Run Queries => Another DataFrame
4) DataFrame.collect() ==> to get the results of data frame

other ops:
5) DataFrame.show()
6) DataFrame.printSchema()
7) Save DataFrame as TextFile or Parquet into local file system or HDFS

Integrating Hive with Spark

Copy hive-site.xml to spark conf folder, This will ensure sparks integrates with hive upon start

```
<configuration>
<property>
  <name>hive.metastore.warehouse.dir</name>
  <value>/user/hive/warehouse</value>
  <description>location of default database for the warehouse</description>
</property>
<property>
  <name>hive.metastore.uris</name>
  <value></value>
  <description>Thrift URI for the remote metastore. Used by metastore client to connect to remote metastore.</description>
</property>
<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:derby:;databaseName=/home/edureka/metastore_db;create=true</value>
  <description>JDBC connect string for a JDBC metastore</description>
</property>

<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>org.apache.derby.jdbc.EmbeddedDriver</value>
  <description>Driver class name for a JDBC metastore</description>
</property>
</configuration>
```

Integrating Hive with Spark

Create\Load\Select HIVE table from Sql Context

```
sqlContext.sql("create table if not exists customer (key INT, value STRING) row format delimited fields terminated by ‘,'")

sqlContext.sql("LOAD DATA LOCAL INPATH 'file:///home/edureka/customers.txt' INTO TABLE customer”)

sqlContext.sql("SELECT * FROM CUSTOMER WHERE value='Andy'").collect()
sqlContext.sql("select * from customer").collect().foreach(println)
```

Drop HIVE table from Sql Context

```
sqlContext.sql("drop table customer")
```

Contents of directory /user/hive/warehouse

Goto : /user/hive/warehouse go

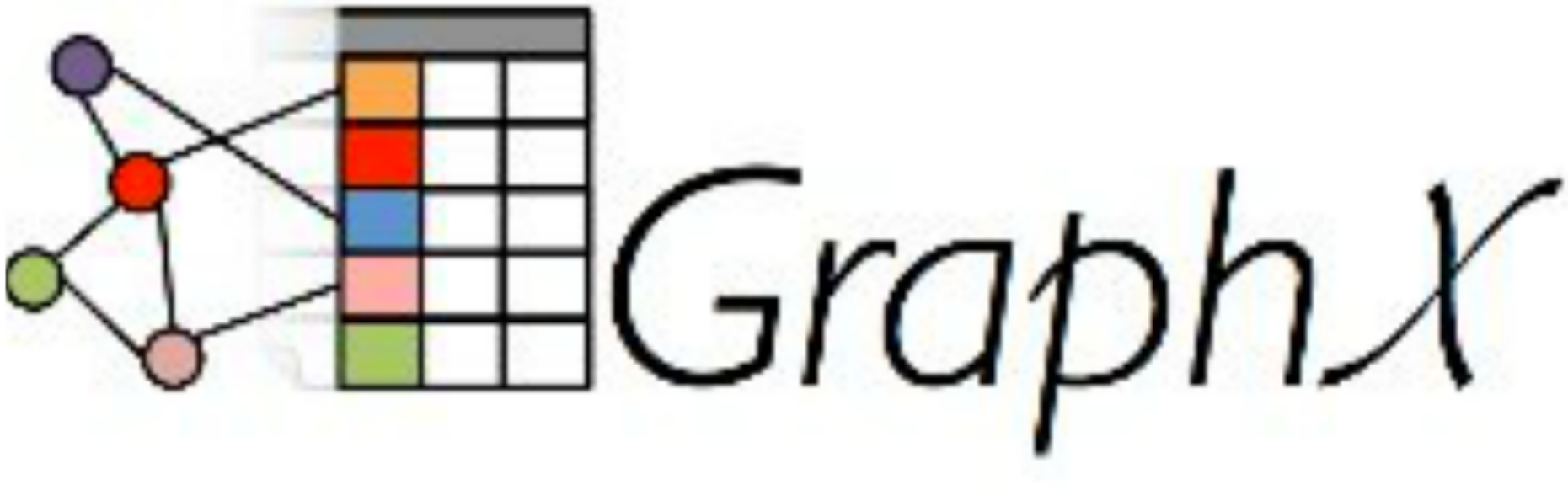
Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
employee	dir				2017-02-11 16:43	rwxr-xr-x	edureka	supergroup
pokes	dir				2017-02-11 12:57	rwxr-xr-x	edureka	supergroup
test.db	dir				2015-04-06 12:34	rwxr-xr-x	edureka	supergroup

Go back to DFS home

(1,val_1),(2,val_2)

1,val_1
2,val_2
3,val_3



neo4j
Giraph

GraphX

→ GraphX is a new component in Spark for graphs and graph-parallel computation

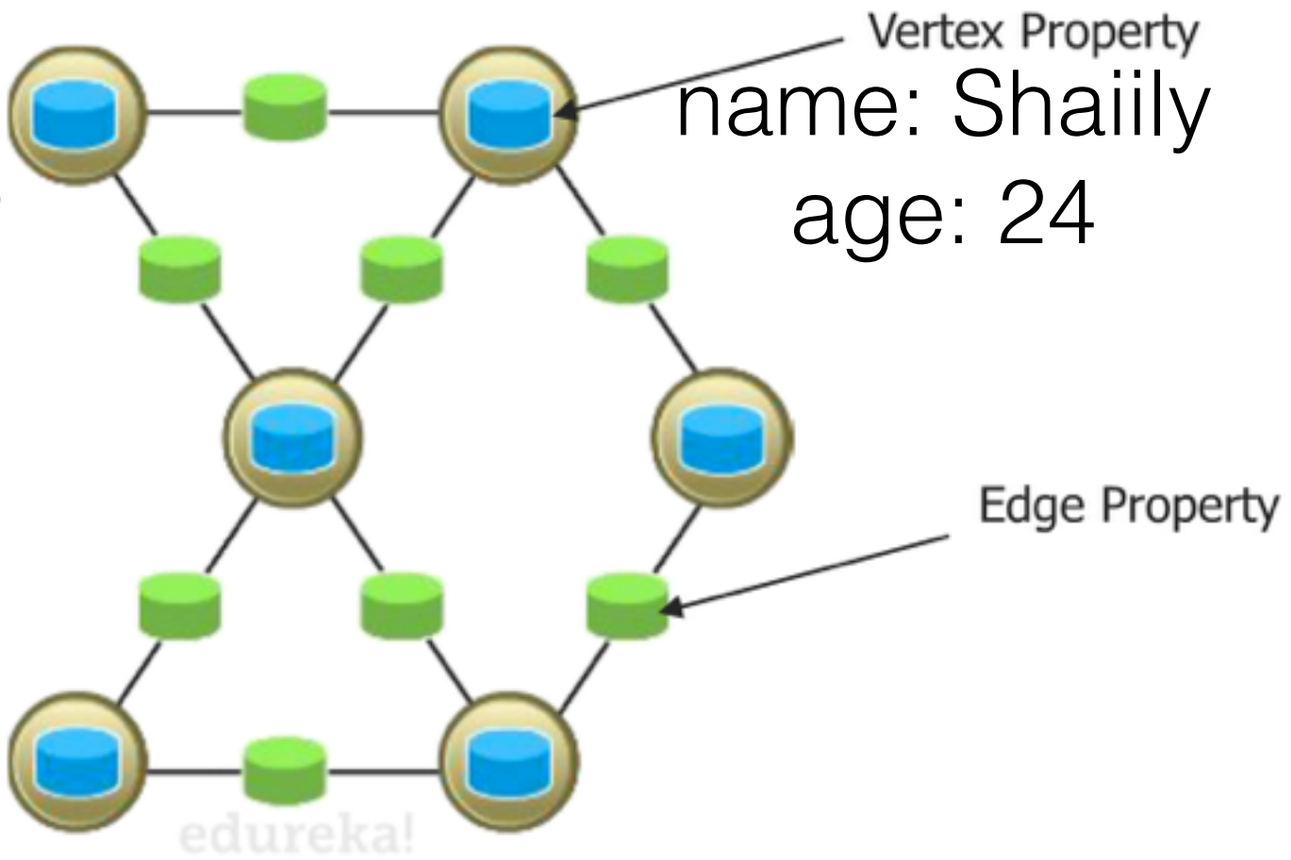
→ GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages)

→ GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks

Property Graph

→The property graph is a directed multigraph with properties attached to each vertex and edge

name: Vindo
age: 28



v1,v2,7

Relationship score = 1 to 10

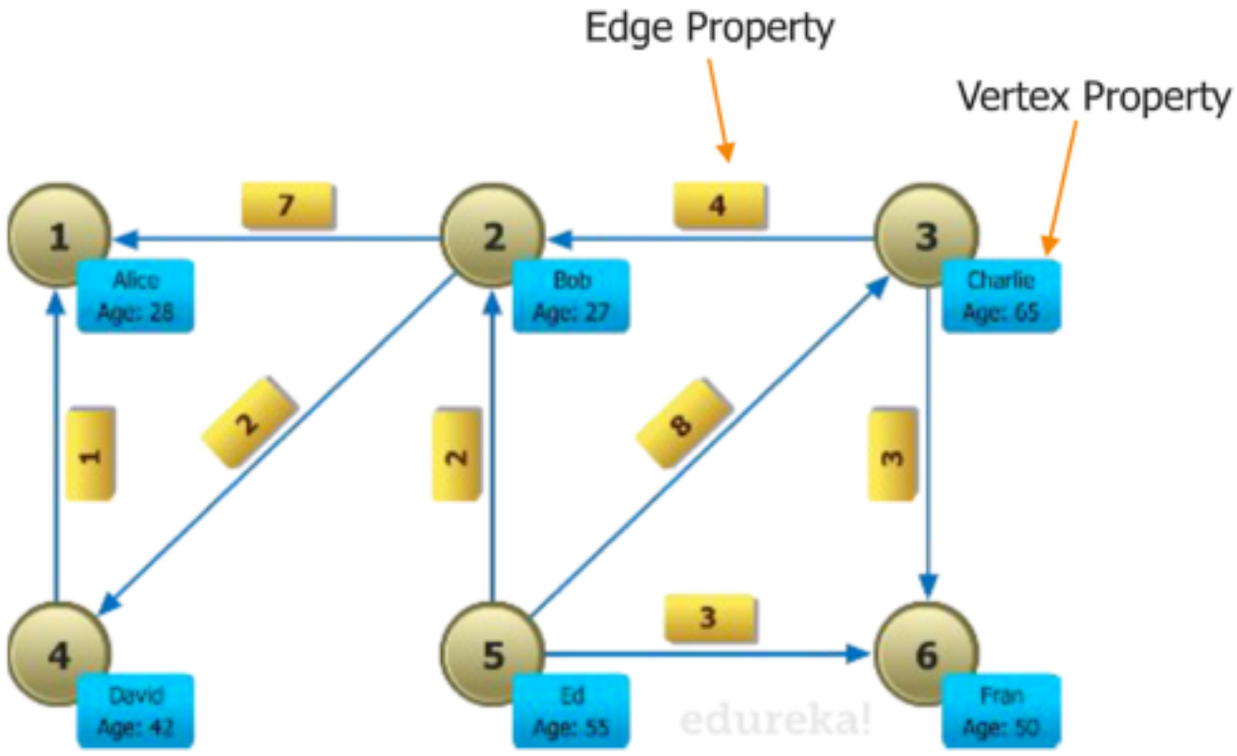
Property Graph

Vertex Array

```
val vertexarray = Array(
    (1L,("Alice",28)),
    (2L,("Bob",27)),
    (3L,("charlie",65)),
    (4L,("David",57)),
    (5L,("Ed",45)),
    (6L,("Frank",67))
    )
```

Edge Array

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

val edgearray = Array(
    Edge(1L,2L,7),
    Edge(2L,4L,2),
    Edge(3L,2L,4),
    Edge(3L,6L,3),
    Edge(4L,1L,1),
    Edge(5L,2L,2),
    Edge(5L,3L,8),
    Edge(5L,6L,3)
    )
```



Creating a Graph RDD from vertices and edges

Creating a Vertices RDD of vertices

```
val vertexRDD:RDD[(Long,(String,Int))] = sc.parallelize(vertexarray)
```

Creating an Edge RDD of edges

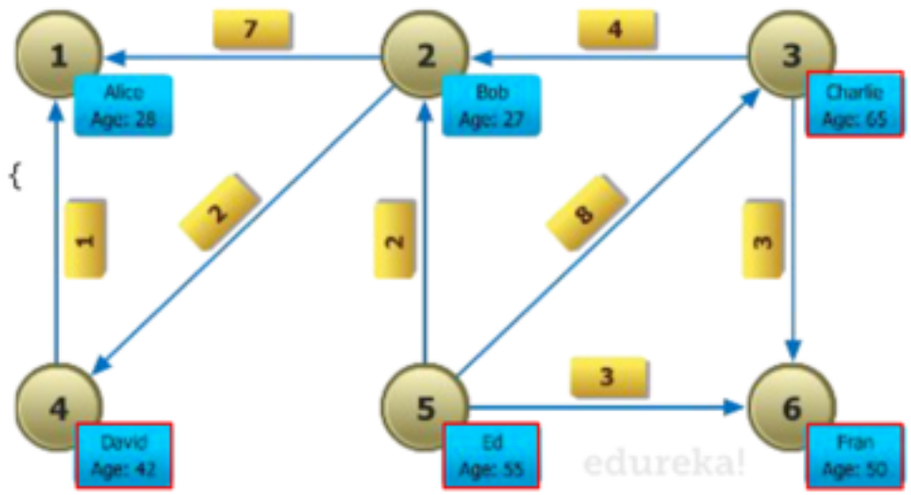
```
val edgeRDD:RDD[Edge[Int]] = sc.parallelize(edgearray)
```

Creating a Graph RDD from Vertices & Edges RDDs

```
val graph:Graph[(String,Int),Int] = Graph(vertexRDD,edgeRDD)
```

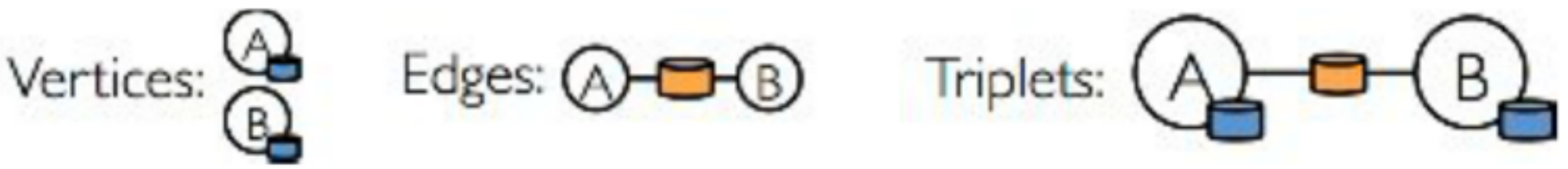
Finding Name and Age of people having age > 30

```
graph.vertices.filter{ case(id,(name,age)) => age > 30}.collect.foreach {
      case(id,(name,age)) => println(s"$name is of $age")
}
```



Triplets

→ The triplet view logically joins the vertex and edge properties yielding an RDD[EdgeTriplet[VD, ED]] containing instances of the EdgeTriplet class



Whats in a Triplet?????

```
for(x <- graph.triplets.collect) {println(x)}

 for(x <- graph.triplets.collect) {println(x.srcAttr)}

 for(x <- graph.triplets.collect) {println(x.dstAttr)}
```

So let's find who likes whom in the group:

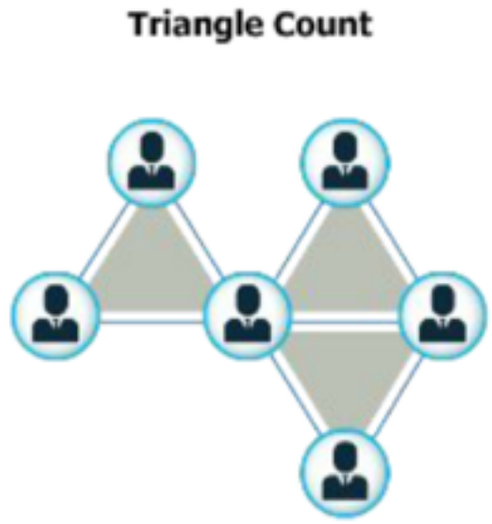
Definition: A person likes another person if they have a relationship score of more than 5 between them

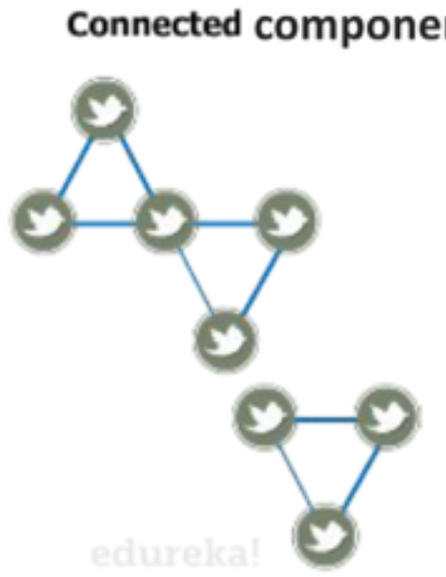
```
for(triplet <- graph.triplets.filter(t => t.attr > 5).collect) {
        println(s"${triplet.srcAttr._2} likes ${triplet.dstAttr._2}")
}
```

Graph Algorithm

→ Built-in algorithms

» PageRank, Connected Components, Triangle Count, ...





BroadCast Variables

Broadcast Variables

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Spark actions are executed through a set of stages, separated by distributed “shuffle” operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

Create BroadCast Variable

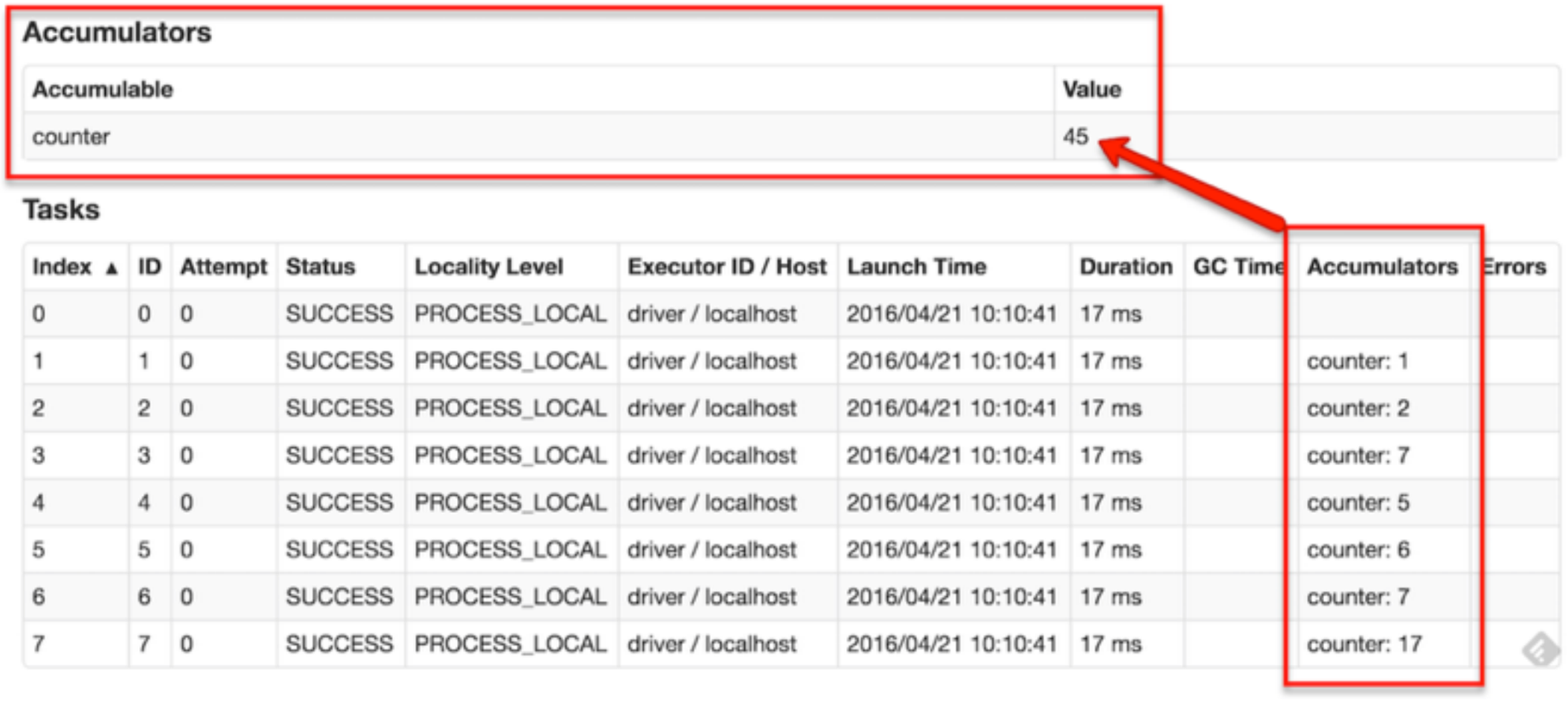
```
val broadcastVar = sc.broadcast(Array(1, 2, 3))

broadcastVar.value
```

Accumulators

Accumulators

Accumulators are variables that are only "added" to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

As a user, you can create named or unnamed accumulators. As seen in the image below, a named accumulator (in this instance `counter`) will display in the web UI for the stage that modifies that accumulator. Spark displays the value for each accumulator modified by a task in the "Tasks" table.



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

```
val accum = sc.longAccumulator("My Accumulator")
```

Spark MLLIB

–Johnny Appleseed