Class 4

Hadoop-Related Tools

CSCI 6830

BIG DATA ANALYTICS
WITH HADOOP AND R

Apache Tez

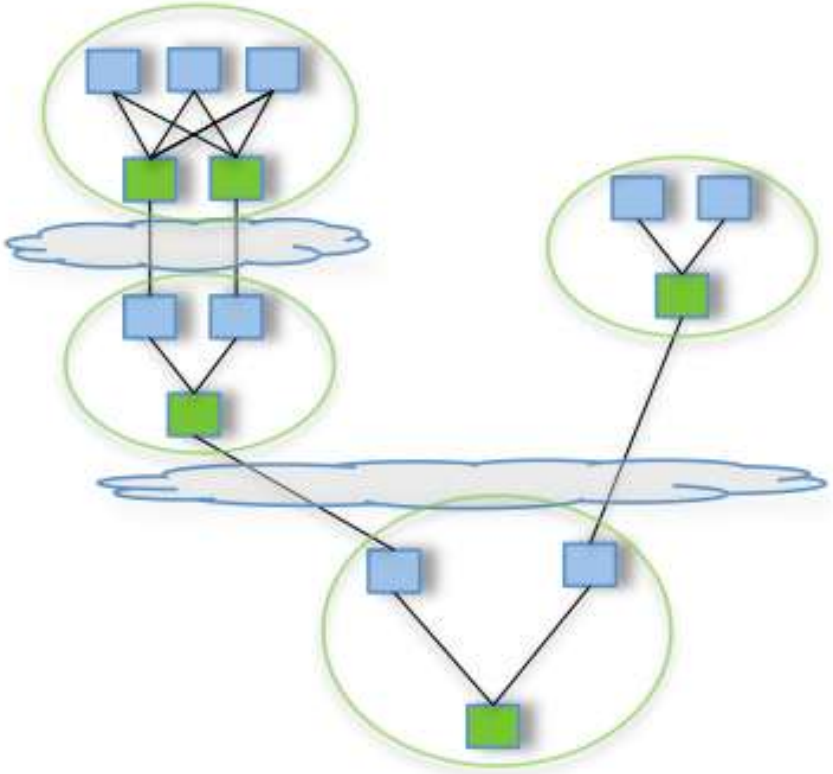
APPLICATION FRAMEWORK FOR A COMPLEX DIRECTED-ACYCLIC-GRAPH OF TASKS FOR PROCESSING DATA.

Apache Tez

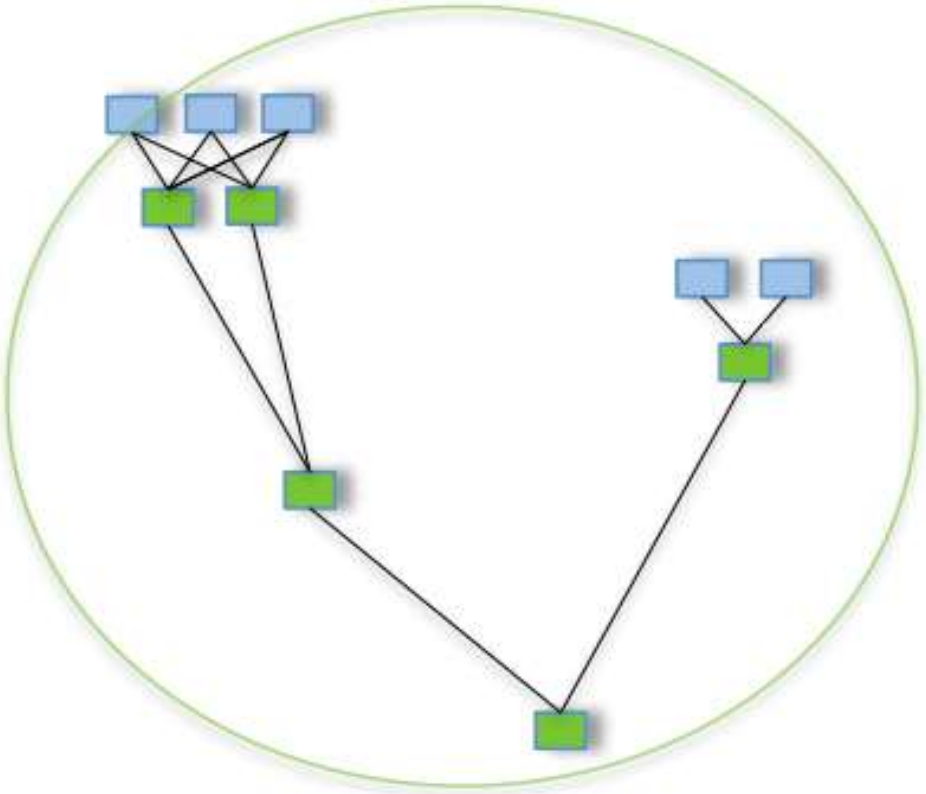
- Expressive dataflow definition APIs
- Flexible Input-Processor-Output runtime model
- Data types do not matter
- Simplifying deployment
- Performance gains over Map Reduce
- Optimal resource management
- Plan reconfiguration at runtime
- Dynamic physical data flow decisions

Apache Tez

- Tez can be used to process data, that earlier took multiple MR jobs, now it is a single Tez job



Pig/Hive - MR



Pig/Hive - Tez

Apache Tez

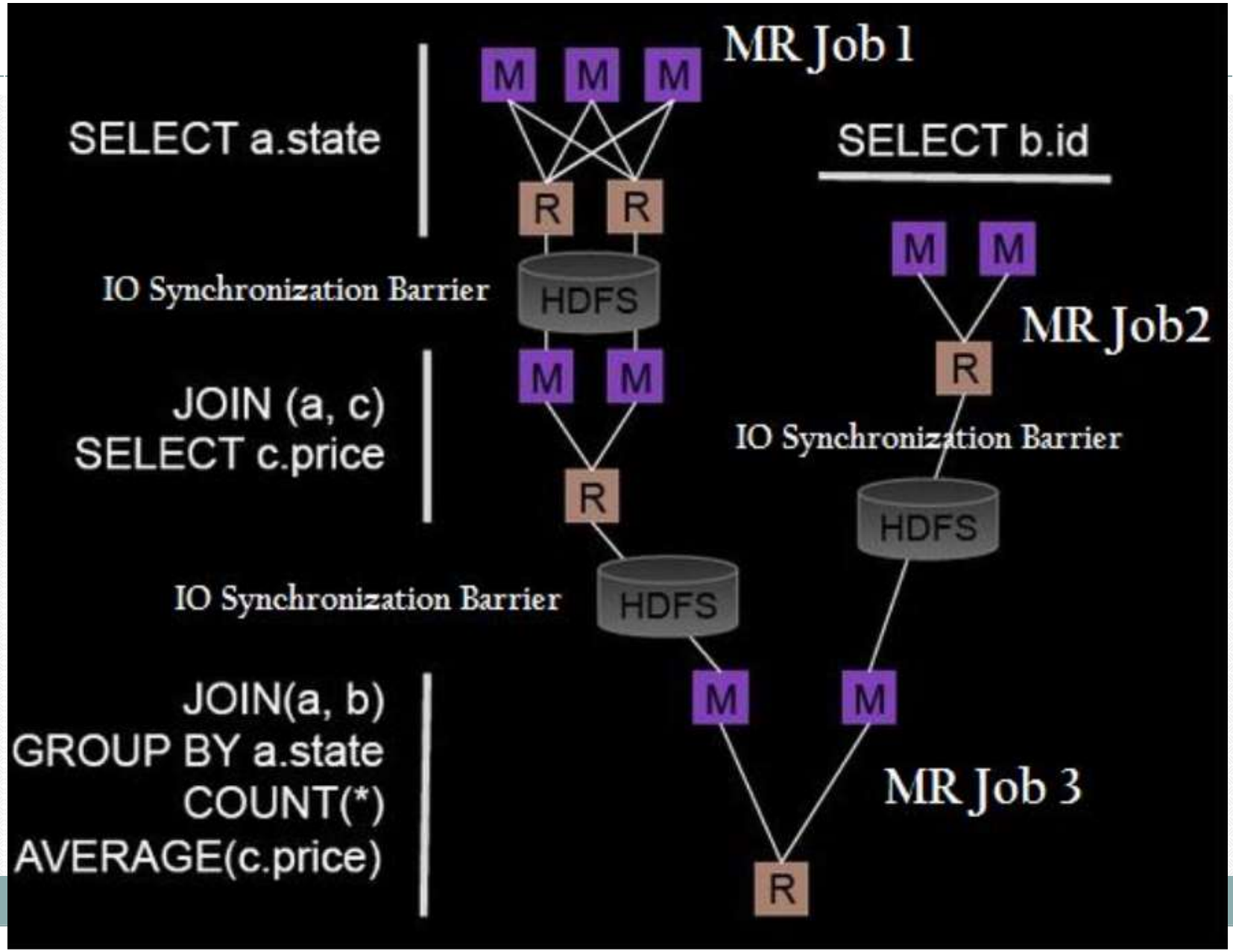
- Allows an application to be modeled as a data flow
- Edges (nodes) reflect the data movement
- Vertices represent the data processing tasks
- After processing the query, Pig or Hive generate a directed acyclic graph (DAG)
- DAG is ready for execution
- Tez is well suited as the executor for Pig or Hive jobs
- Tez is not only an engine though
- Tez also provides common modules to develop applications
- Tez focuses on flexibility and customizability.
- Developers may implement actual MapReduce jobs via the Tez library.

Apache Tez Example

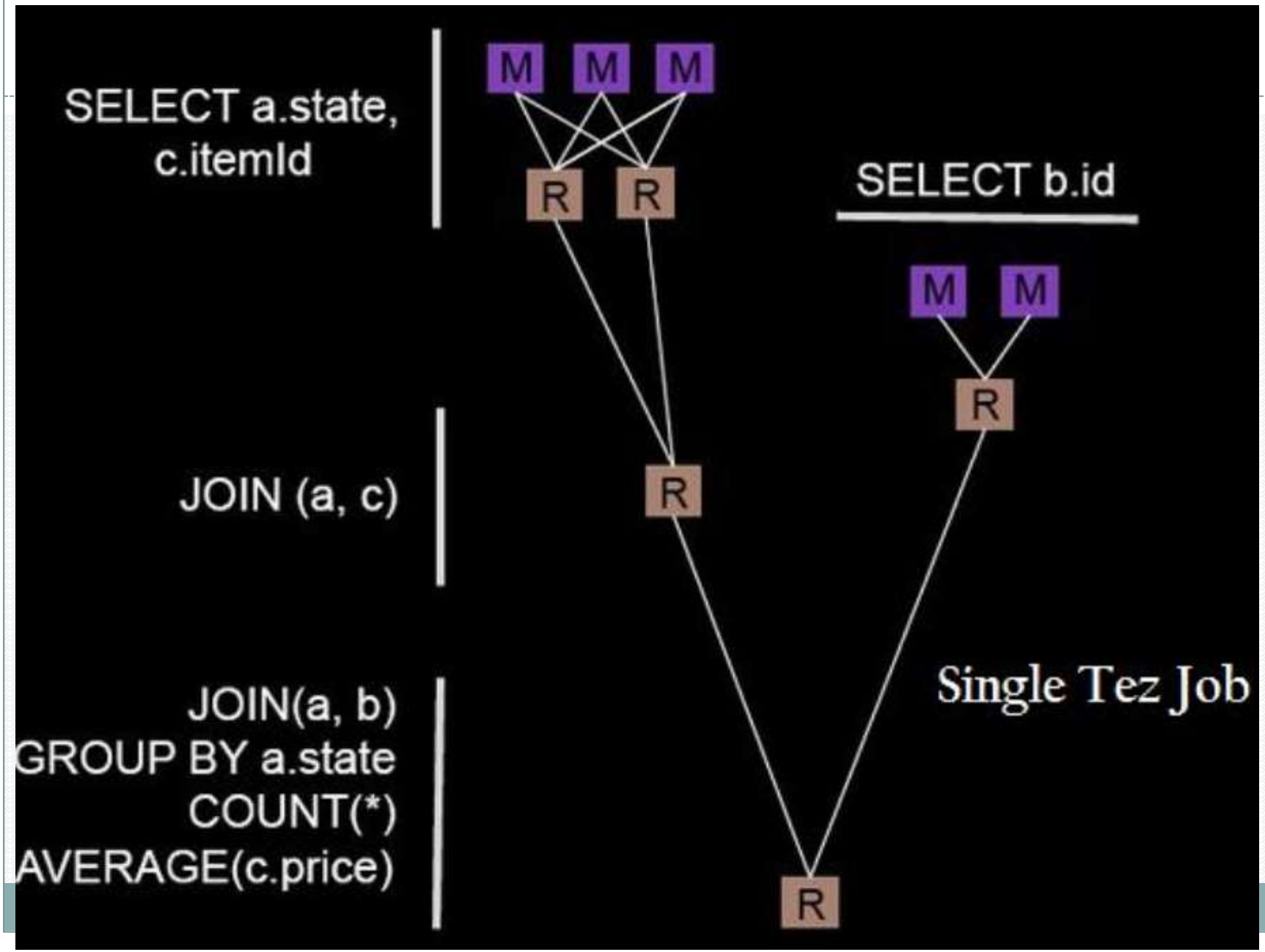
- Example Hive query

```
SELECT a.state, COUNT(*), AVERAGE(c.price)
FROM a
JOIN b ON(a.id = b.id)
JOIN c ON(a.itemId = c.itemId)
GROUP BY a.state
```

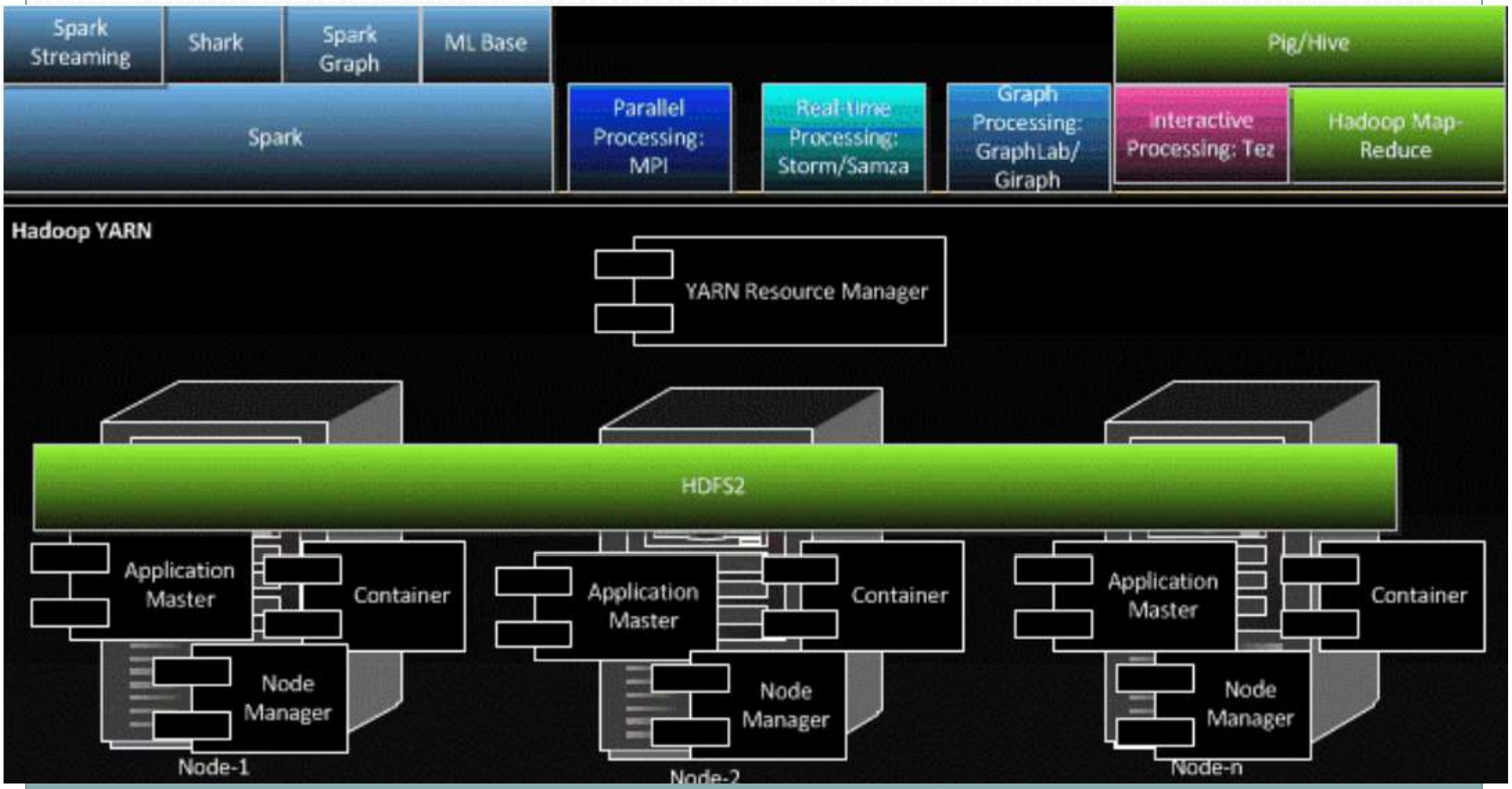
Query in MapReduce



Query in Apache Tez



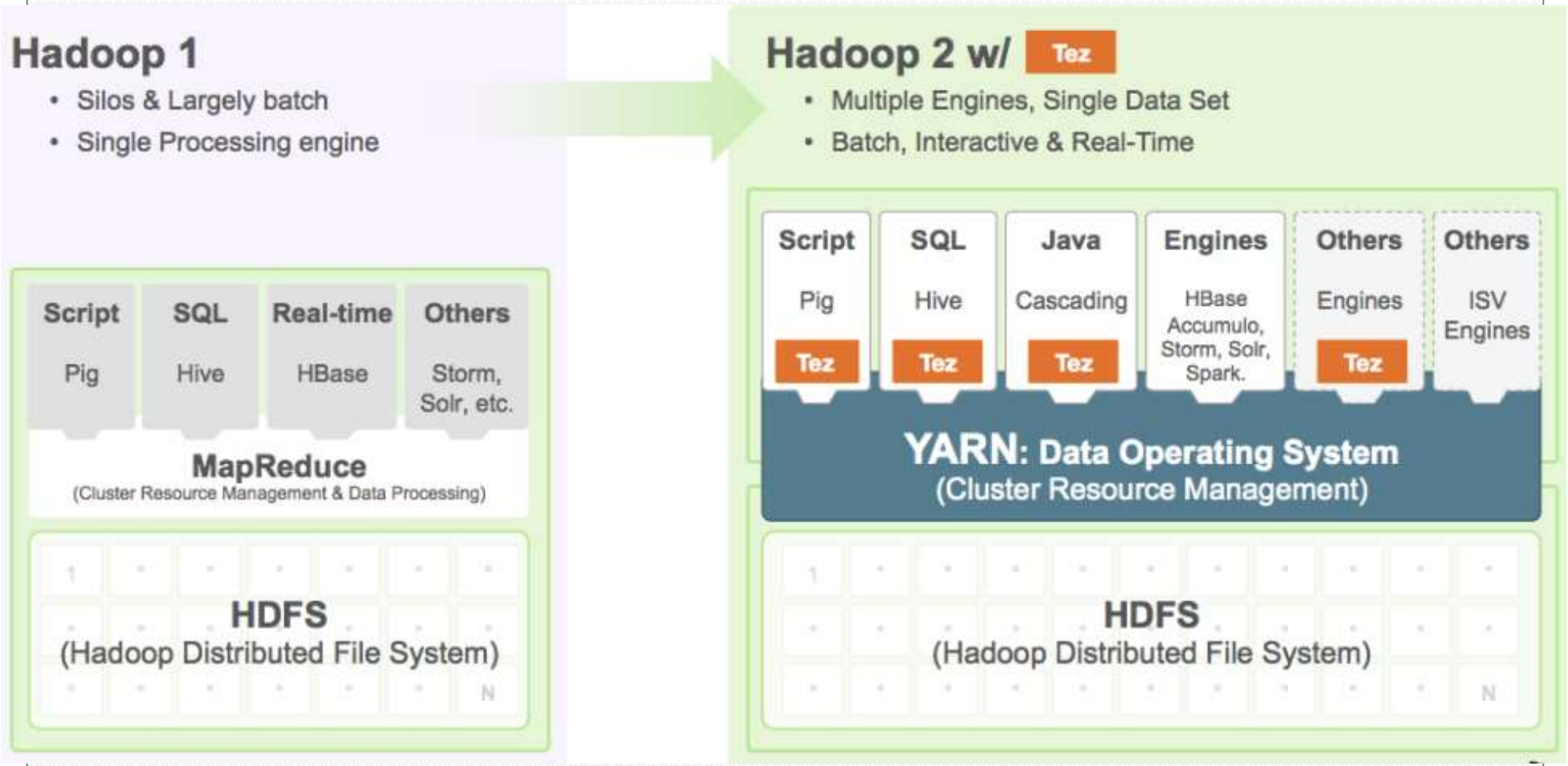
Expanded YARN Framework



Tez Summary

- Tez development framework includes Java APIs
  - Provide developers with an intuitive interface to create the data-processing graphs for the applications' data-processing flows.
- Tez provides additional APIs to inject custom logic
- Modular input, output, and processor elements are provided.
- Applications developed via these APIs execute efficiently in a Hadoop environment
- Tez framework provides the interaction with other stack components
- Tez applications represent customized, optimized, and YARN integrated solutions that take advantage of Hadoop's scalable and fault-tolerant runtime environment.

Tez Summary


Hadoop 1
• Silos & Largely batch
• Single Processing engine
Script
Pig
SQL
Hive
Real-time
HBase
Others
Storm, Solr, etc.
MapReduce
(Cluster Resource Management & Data Processing)
HDFS
(Hadoop Distributed File System)
Hadoop 2 w/ Tez
• Multiple Engines, Single Data Set
• Batch, Interactive & Real-Time
Script
Pig
Tez
SQL
Hive
Tez
Java
Cascading
Tez
Engines
HBase Accumulo, Storm, Solr, Spark.
Others
Engines
Tez
Others
ISV Engines
YARN: Data Operating System
(Cluster Resource Management)
HDFS
(Hadoop Distributed File System)

Class 4
Apache Pig

PLATFORM FOR ANALYZING LARGE DATA SETS

Summary

- Big demand for parallel data processing
  - Emerging tools that do not look like SQL DBMS
  - Programmers like dataflow pipes over static files
- Hence the excitement about Map-Reduce
- But, Map-Reduce is too low-level and rigid

Pig Latin

Sweet spot between map-reduce and SQL

Source:isg.ics.uci.edu/slides/YahooPig.ppt

What is Pig

- A platform for analyzing large data sets
  - Contains a high-level language for **expressing** data analysis programs.
- Compiles down to MapReduce jobs
- Developed by Yahoo!
- Open-source language

Pig Components

Two Main Components

- High-level language (Pig Latin)
  - Set of commands
- Two execution modes
  - Local: reads/write to local file system
  - Mapreduce: connects to Hadoop cluster and reads/writes to HDFS

Two modes

- Interactive mode
  - Console
- Batch mode
  - Submit a script

Pig's Team

In 2009, 8 out of 10 Pig contributors were from Yahoo.

Source:isg.ics.uci.edu/slides/YahooPig.ppt

Apache Pig Background

- Yahoo! was the first big adopter of Hadoop.
- Hadoop gained popularity in the company quickly.
- Yahoo! Research developed Pig to address the need for a higher level language.
- Roughly 30% of Hadoop jobs run at Yahoo! are Pig jobs.

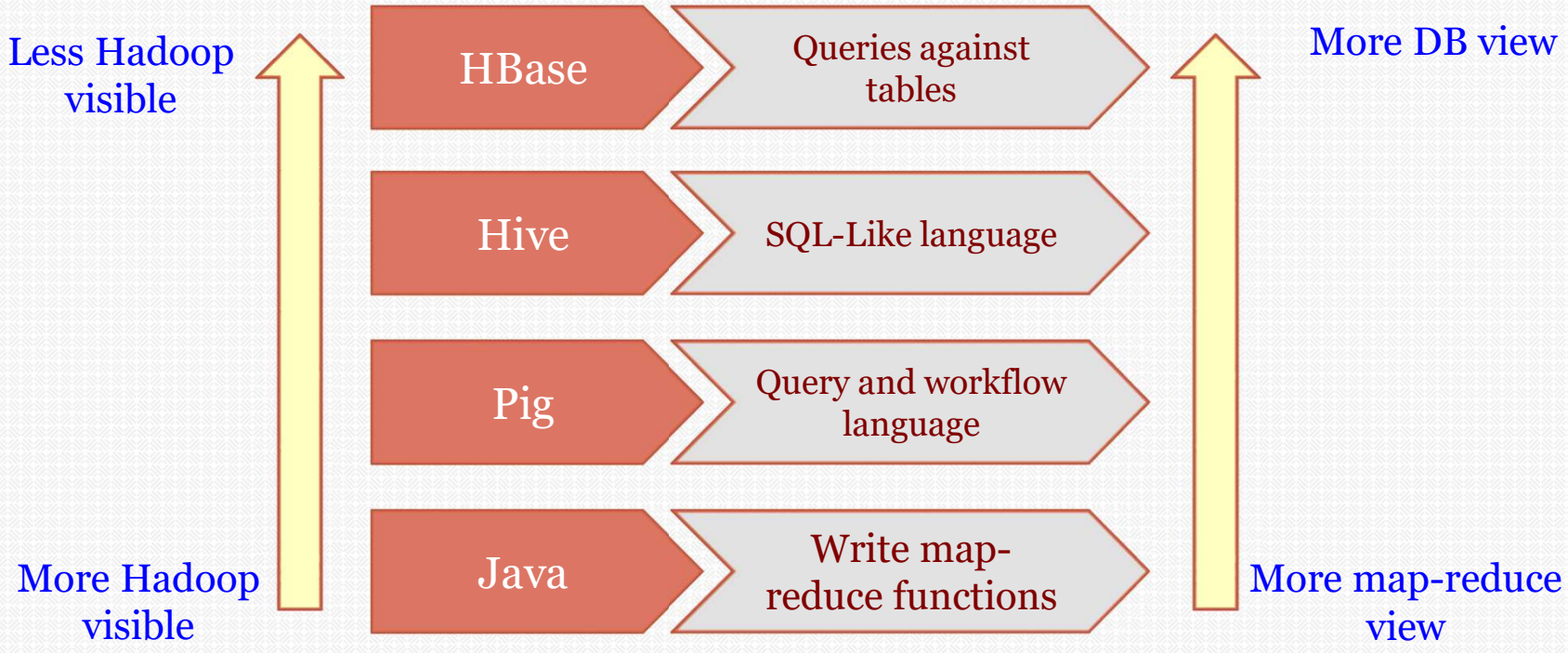
How Apache Pig is used?

- Web log processing
- Data processing for web search platforms
- Ad hoc queries for large datasets
- Rapid prototyping of algorithms for large datasets

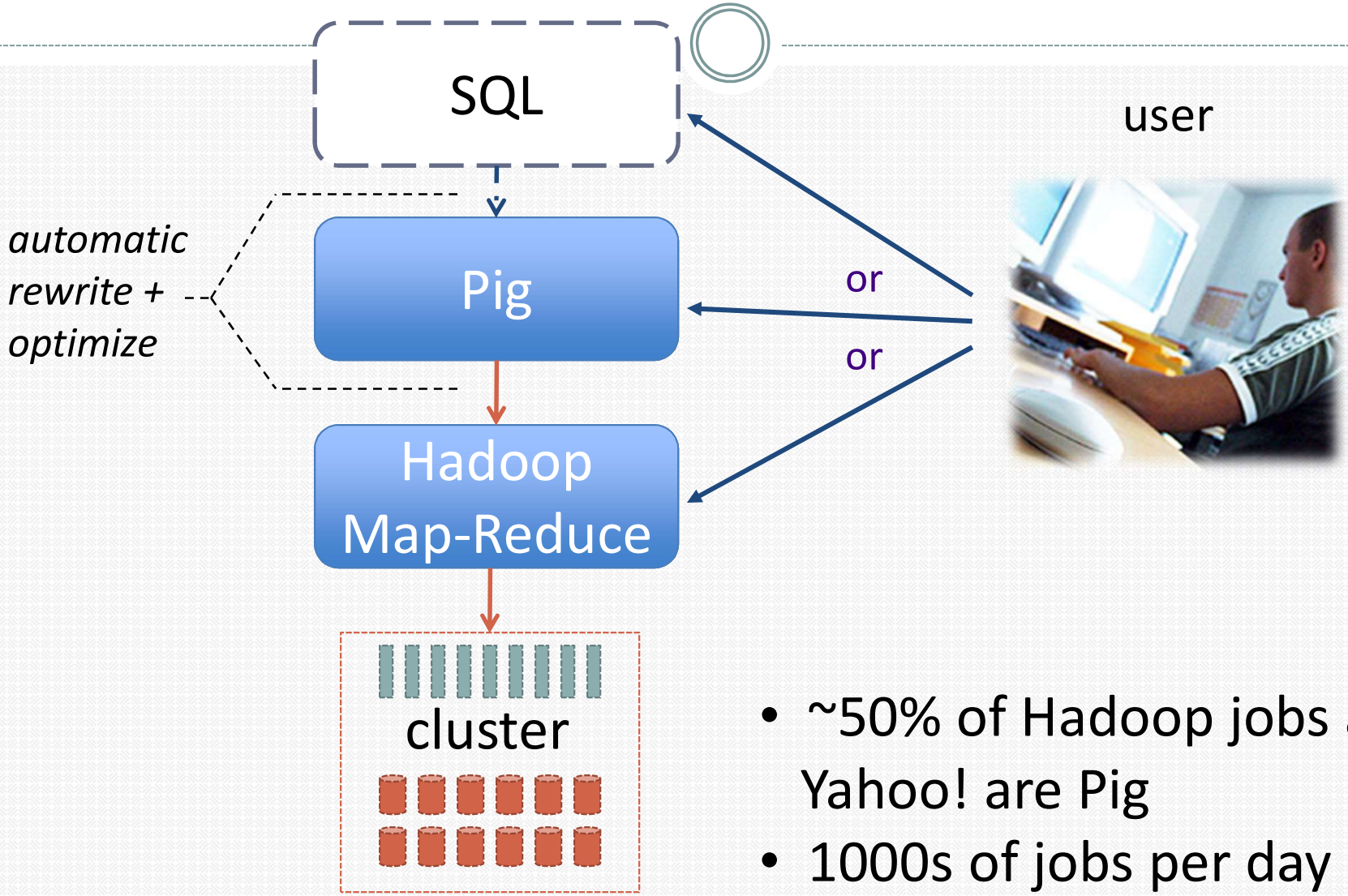
Apache Pig

- High level scripting language in Apache Hadoop stack
- Good for data analysis as data flows.
- Has many data manipulation commands
- User Defined Functions(UDF)
  - invoke code in many languages
  - JRuby, Jython and Java.
- Can run Pig scripts from other languages.
- Pig may be useod as a component for large and complex applications

Levels of Abstraction



Implementation



- ~50% of Hadoop jobs at Yahoo! are Pig
- 1000s of jobs per day

Pig Latin vs. SQL

- Pig Latin is procedural (dataflow programming model)
  - Step-by-step query style is much cleaner and easier to write
- SQL is declarative but not step-by-step style

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;
```

Pig Latin

```
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';
```

Pig Latin vs. SQL

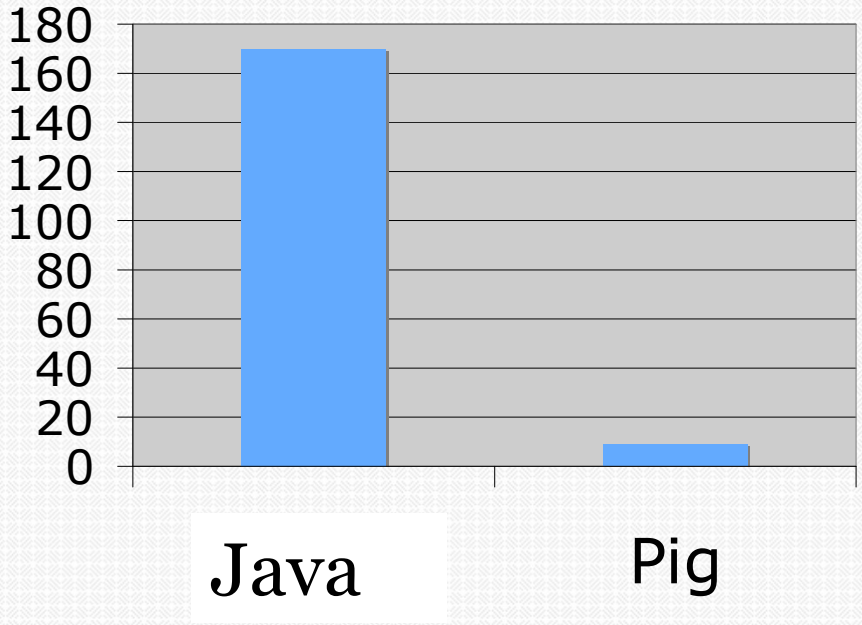
- **In Pig Latin**
  - Lazy evaluation (data not processed prior to STORE command)
  - Data can be stored at any point during the pipeline
  - Schema and data types are lazily defined at run-time
  - An execution plan can be explicitly defined
    - Use optimizer hints
    - Due to the lack of complex optimizers
- **In SQL:**
  - Query plans are solely decided by the system
  - Data cannot be stored in the middle
  - Schema and data types are defined at the creation time

Apache Pig

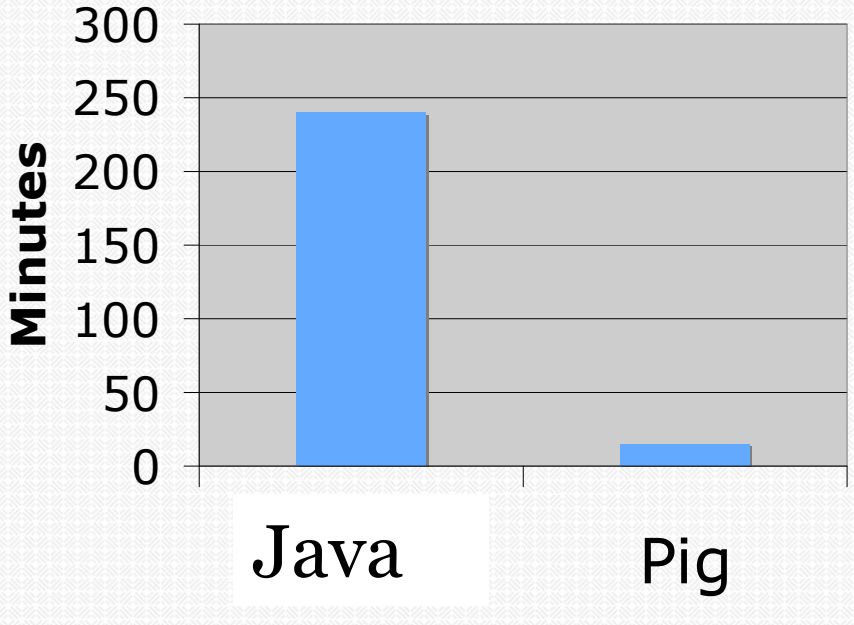
- Map Reduce is very powerful, but:
  - Requires a Java programmer.
  - User has to write common SQL functionality from scratch
    - joins, filtering, etc.
- Pig provides a higher level language, Pig Latin:
  - Increases productivity.
  - 10 lines of Pig Latin ≈ 200 lines of Java.
  - 4 hours in Java ≈ 15 minutes in Pig Latin.
  - Good for non-Java programmers.
  - Provides common operations:
    - Join, Group, Filter, Sort.

Java vs. Pig

1/20 the lines of code



1/16 the development time

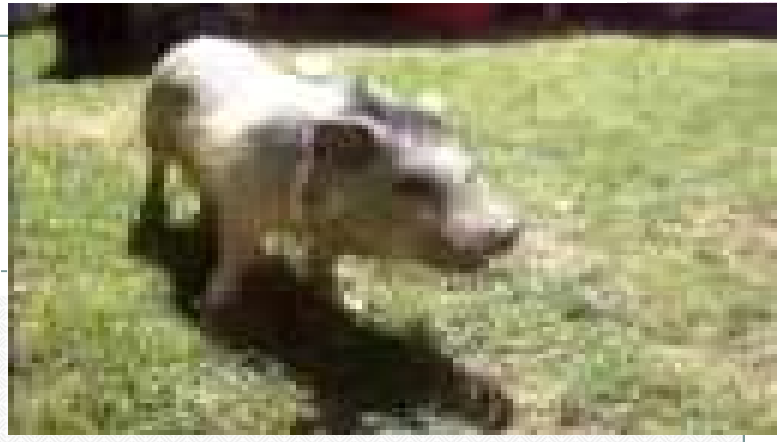


Performance is comparable (Java is slightly better)

Pig Lating

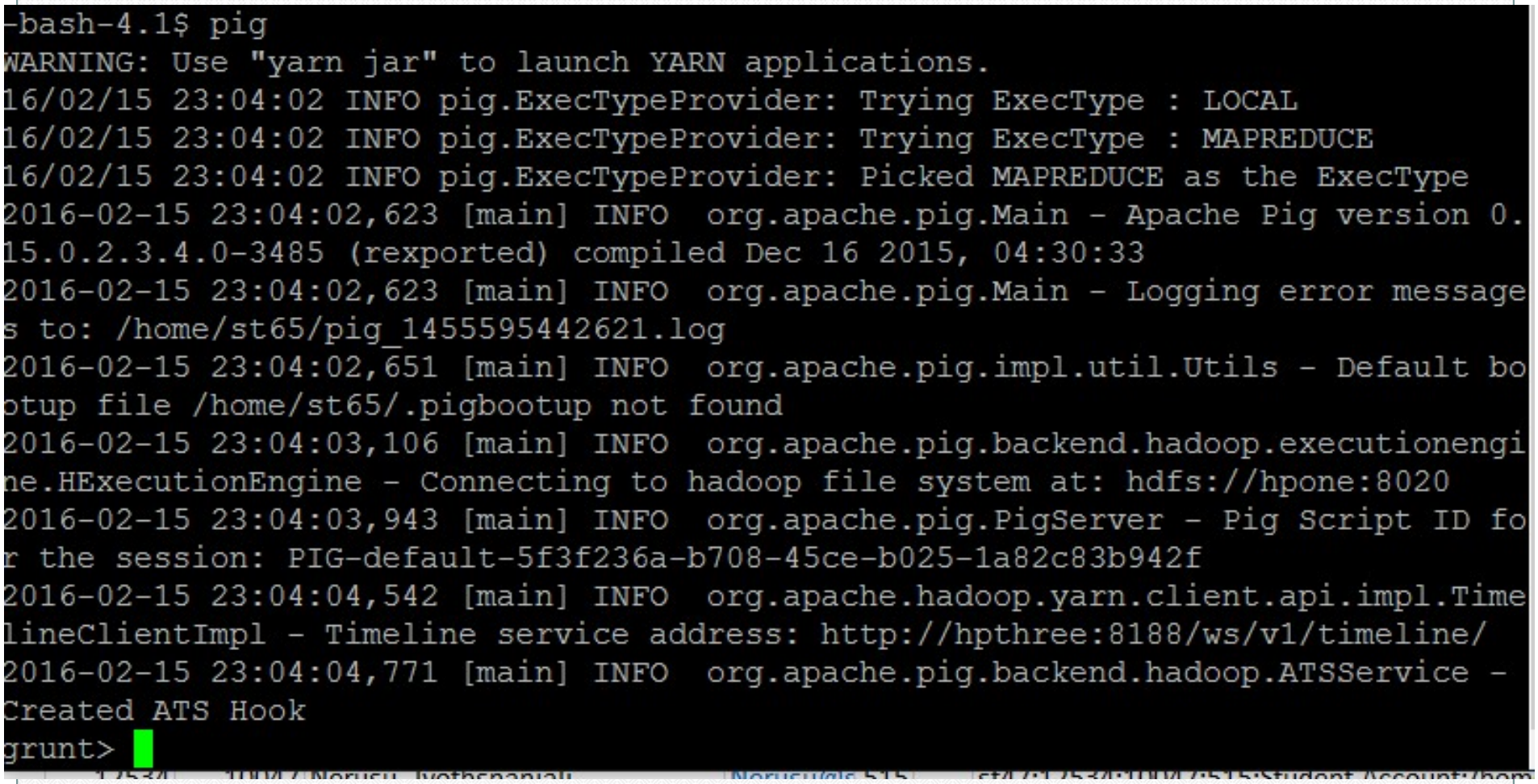
- Pig Latin is a data flow language
  - Neither procedural nor declarative.
- User code and existing binaries can be included almost anywhere.
- Metadata not required, but used when available.
- Supports nested types.
- Operates on files in HDFS
  - No need for table mappings
  - No need for ETL

Several ways to use Pig



- Type commands in Pig's Grunt (Pig CLI)
- Submit a script to Pig
- Use it from Java with PigServer Java class and JDBC-like interface
- Use it from Eclipse with PigPen
  - PigPen is an Eclipse plugin
  - Allows textual and graphical scripting
  - Samples data and shows example data flow

Grunt is a Pig's CLI



How Apache Pig works

Pig Latin

```
A = LOAD 'myfile'
    AS (x, y, z);
B = FILTER A by x > 0;
C = GROUP B BY x;
D = FOREACH A GENERATE
    x, COUNT(B);
STORE D INTO 'output';
```

pig.jar:

- parses
- checks
- optimizes
- plans execution
- submits jar to Hadoop
- monitors job progress

Execution Plan

Map:
Filter

Reduce:
Count

Example. Pig reads and displays a file

```
grunt> log = load 'excite-small.log' as (user, timestamp, query);
grunt> dump log
```

- Pig reads the **excite-small.log** file from HDFS
  - It takes it from your HDFS user directory
- Pig translates the command into MapReduce jobs
  - The jobs are executed in parallel in the cluster
- Output looks like this:

(818D7157855D5D48,970916080224,halloween)

(99BA461C4F96233F,970916023950,"tibetan terrier")

(887E337AB02C2BF7,970916170230,edgar allen poe)

Pig Data Types

- Scalar types:
  - Int
  - Long
  - Double
  - Chararray
  - Bytearray
- Complex types:
  - Map: associative array
  - Tuple: ordered list of data, elements may be of any scalar or complex type
  - Bag: unordered collection of tuples

Pig Example (continued)

```
log = load 'excite-small.log' as (user, timestamp, query);
 grpd = group log by user;
dump grpd;
--- OUTPUT: ----------------------------------------
(74BB49A63C0C4D79,
{(74BB49A63C0C4D79,970916104542,mcgraw hill),
(74BB49A63C0C4D79,970916073222,green tree),
(74BB49A63C0C4D79,970916073158,green tree),
(74BB49A63C0C4D79,970916071727,be-cubed),
(74BB49A63C0C4D79,970916071706,be-cubed),
(74BB49A63C0C4D79,970916104549,mcgraw hill),
(74BB49A63C0C4D79,970916071651,b-cubed)}
)
```

Apache Pig Example

- Aggregation

Let's count the number of times each user appears in the excite data set.

```
log  = LOAD 'excite-small.log'
       AS (user, timestamp, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE group, COUNT(log);
STORE cntd INTO 'output';
```

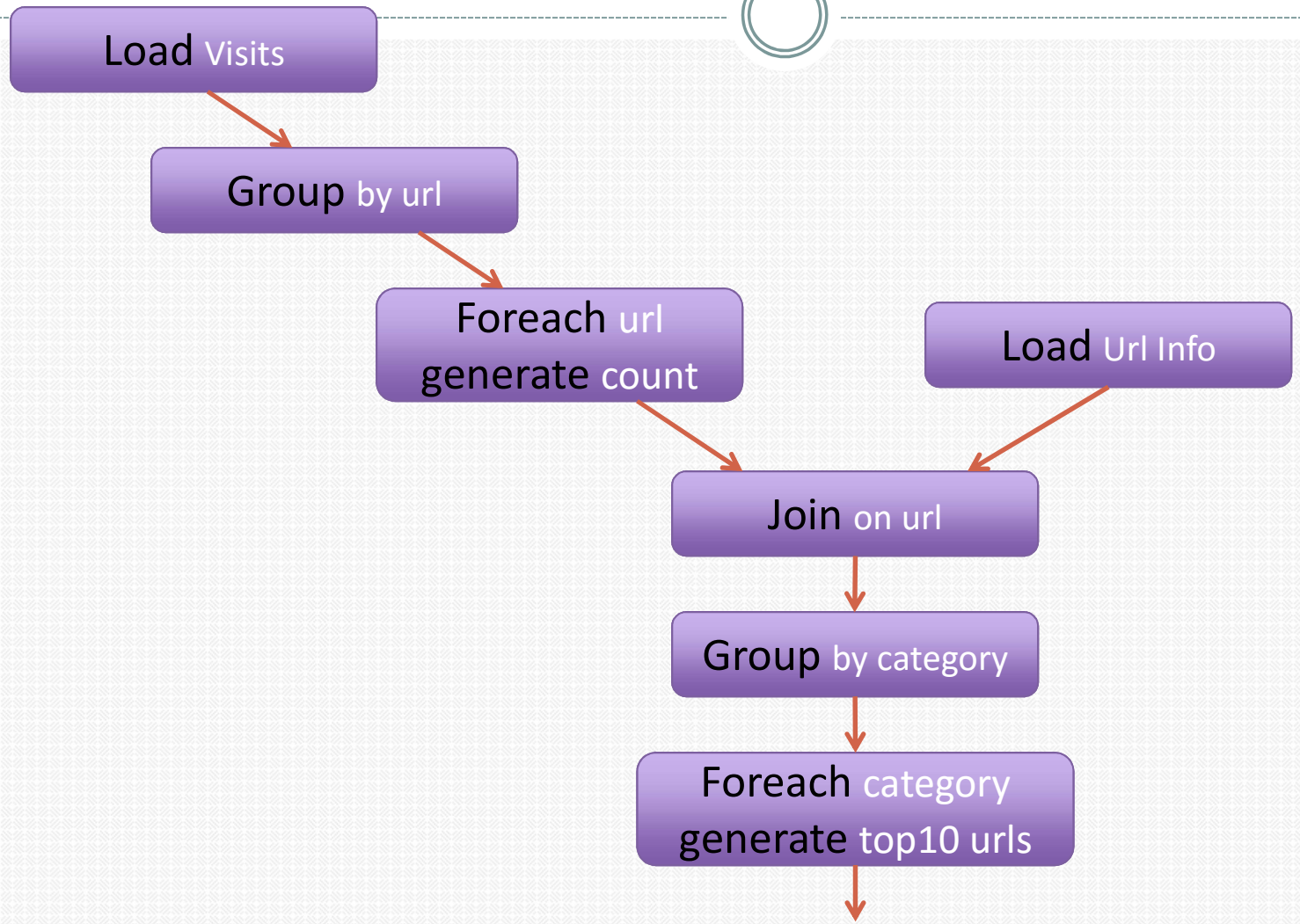
Results:

```
002BB5A52580A8ED        18
005BD9CD3AC6BB38        18
```

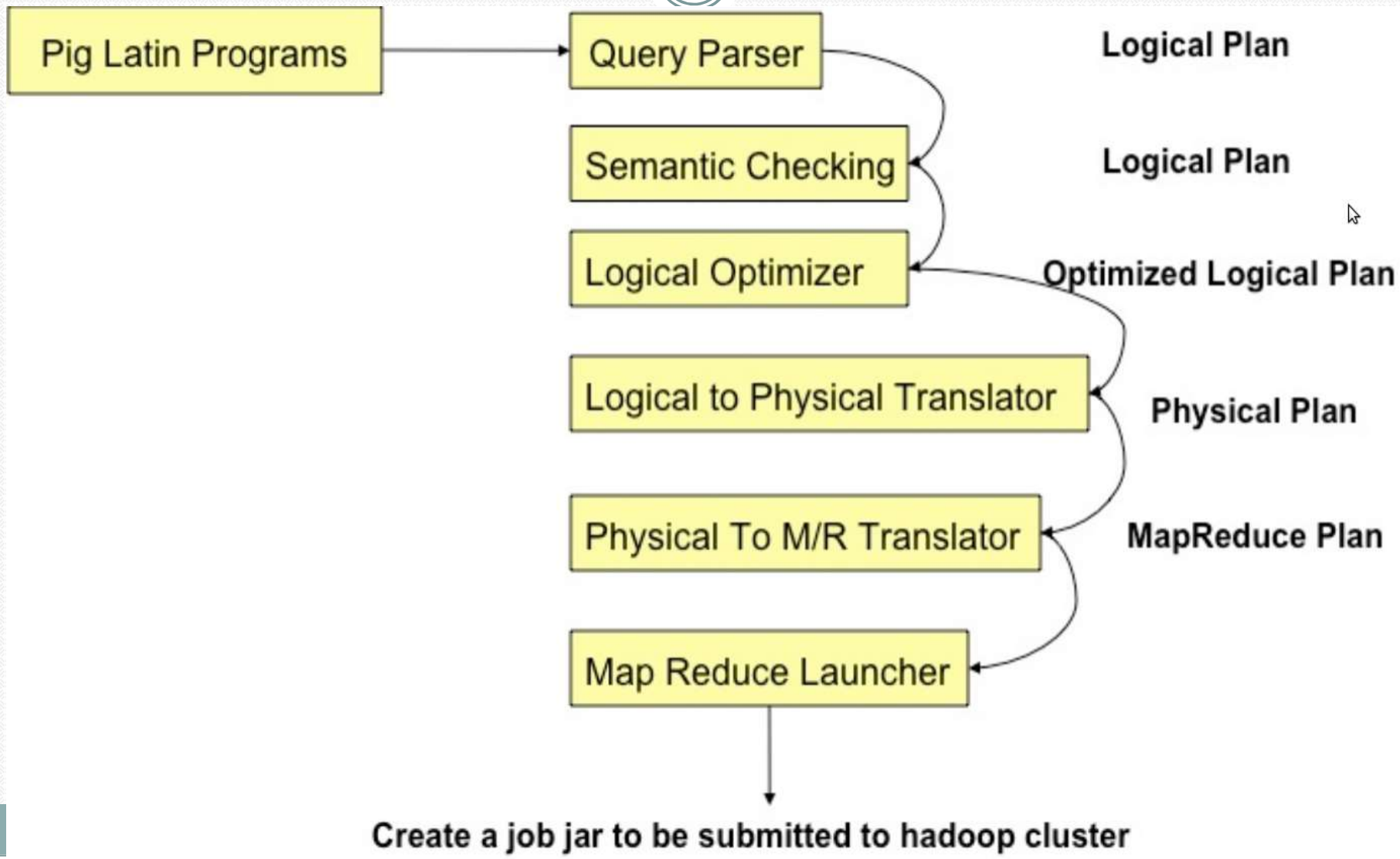
Another Example in Pig Latin

```
visits       = load '/data/visits' as (user, url, time);
gVisits      = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo      = load '/data/urlInfo' as (url, category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

This is how it is excecuted

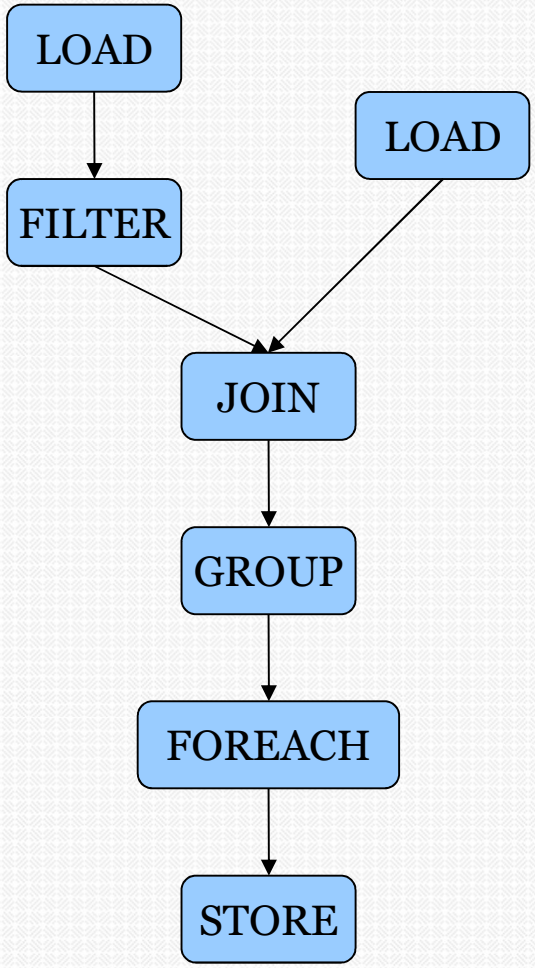


Pig Compilation



Logic Plan

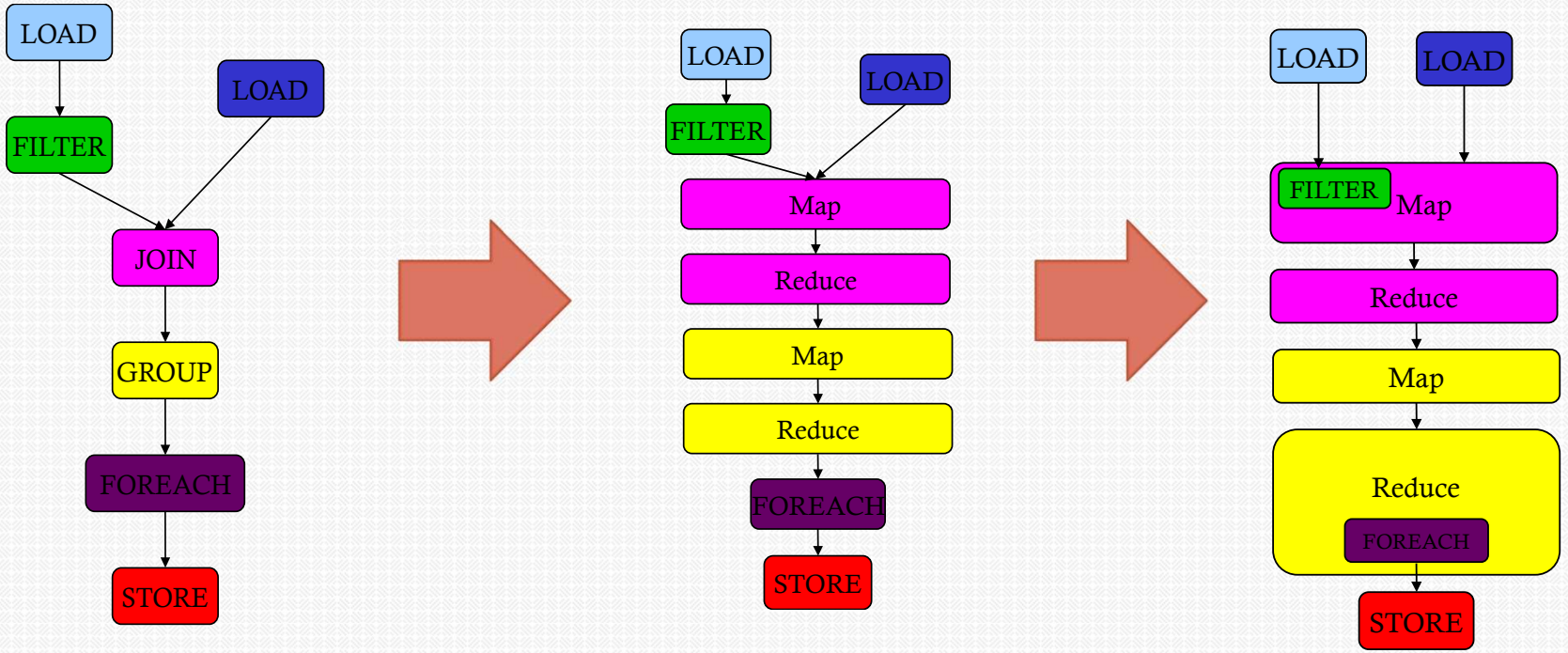
```
A=LOAD 'file1' AS (x, y, z);

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 the logical plan
- **Except for:**
  - Join, Distinct, (Co)Group, Order
- Several optimizations are done automatically

Generation of Physical Plans



If the Join and Group By are on the same key ➔ The two map-reduce jobs would be merged into one.

Grouping

- In Pig grouping is separate operation from applying aggregate functions.
- In previous slide, output of group statement is (key, bag), where key is the group key and bag contains a tuple for every record with that key.
- For example:

```
alan 1
bob  9    =>    alan, {(alan, 1),(alan, 3)}
alan 3          bob, {(bob, 9)}
```

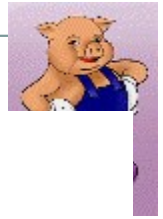
Filtering

Now let's apply a filter to the groups so that we only get the high frequency users.

```
log   = LOAD 'excite-small.log'
         AS (user, time, query);
grpd  = GROUP log BY user;
cntd  = FOREACH grpd GENERATE
         group, COUNT(log) AS cnt;
fltrd = FILTER cntd BY cnt > 50;
STORE fltrd INTO 'output';
```

Results:

```
0B294E3062F036C3        61
128315306CE647F6        78
7D286B5592D83BBE        59
```



Ordering

Sort the high frequency users by frequency.

```
log   = LOAD 'excite-small.log'
          AS (user, time, query);
grpd  = GROUP log BY user;
cntd  = FOREACH grpd GENERATE
          group, COUNT(log) AS cnt;
fltrd = FILTER cntd BY cnt > 50;
srtd  = ORDER fltrd BY cnt;
STORE srtd INTO 'output';
```

Results:

```
7D286B5592D83BBE        59
0B294E3062F036C3        61
128315306CE647F6        78
```

Prepare Data For Joins

Run a pig job to prepare the Shakespare and King James Bible data for the following examples:

```
A = load 'bible';
B = foreach A generate
    flatten(TOKENIZE((chararray)$0)) as word;
C = filter B by word matches '\\w+';
D = group C by word;
E = foreach D generate COUNT(C), group;
store E into 'bible_freq';
```

Repeat for Shakespeare, changing ‘bible’ to ‘input’ in line 1 and ‘bible_freq’ to ‘shakespeare_freq’ in line 7.

Join

We can use join to find words that are in both the King James version of the Bible and Shakespeare's collected works.

```
bard   = LOAD 'shakespeare_freq' AS (freq, word);
kjv    = LOAD 'bible_freq' AS (freq, word);
inboth = JOIN bard BY word, kjv BY word;
STORE inboth INTO 'output';
```

Results:

```
2       Abide   1       Abide
2       Abraham 111     Abraham
3   …
```

Anti-Join

Find words that are in the Bible that are not in Shakespeare.

```
bard   = LOAD 'shakespeare_freq' AS (freq, word);
kjv    = LOAD 'bible_freq' AS (freq, word);
grpd   = COGROUP bard BY word, kjv BY word;
nobard = FILTER grpd BY COUNT(bard) == 0;
out    = FOREACH nobard GENERATE FLATTEN(kjv);
STORE out INTO 'output';
```

Results:

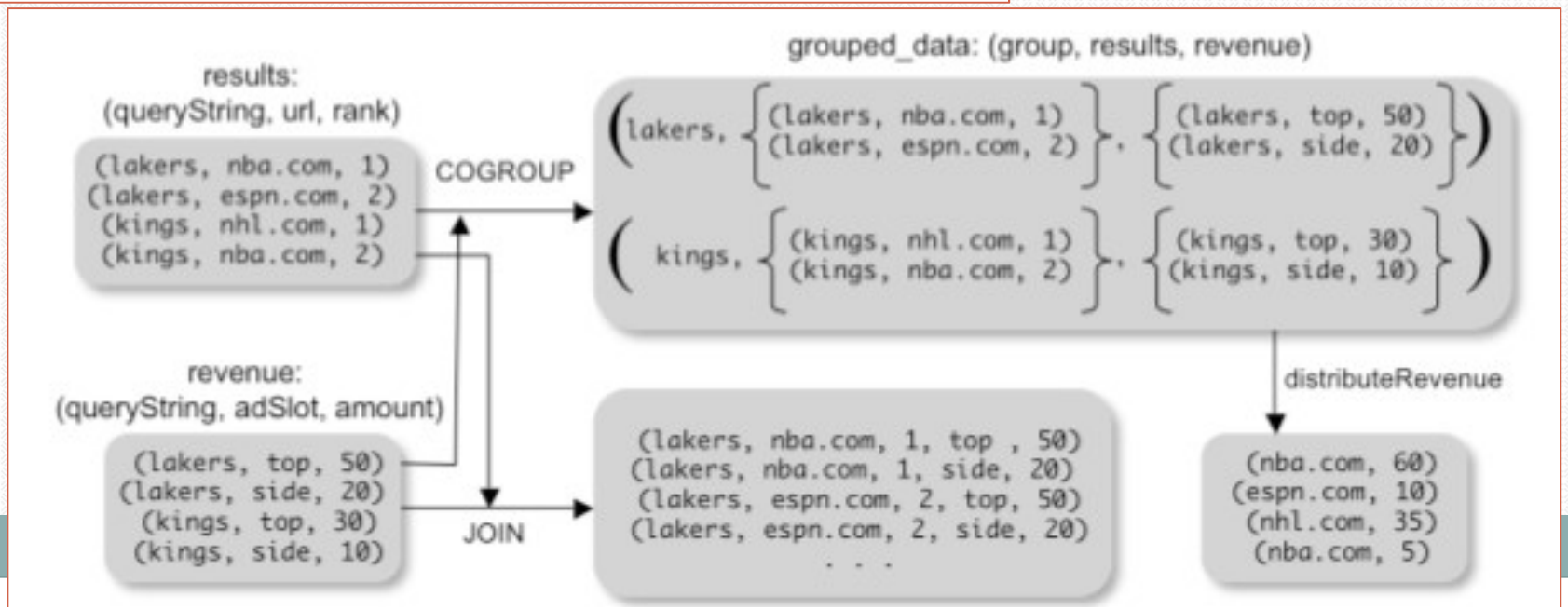
```
1       Abimael
22      Abimelech
```

CoGroup: combination of **join** and **group by**

```
grouped_data  =  COGROUP results BY queryString,
                        revenue BY queryString;
```

```
join_result   =  JOIN results BY queryString,
                 revenue BY queryString;
```

```
temp_var    =  COGROUP results BY queryString,
               revenue BY queryString;
join_result =  FOREACH temp_var GENERATE
               FLATTEN(results), FLATTEN(revenue);
```



Cogrouping

- Cogrouping is a generalization of grouping.
- Keys for two (or more) inputs are collected.
- In previous slide, output of group statement is (key, bag1, bag2), where key is the group key, bag1 contains a tuple for every record in the first input with that key, and bag2 contains a tuple for every record in the second input with that key.

Cogrouping Continued

- For example:

```
Input 1:                    Input 2:
alan 1                      alan 5
bob  9                      bob  9
alan 3

Output:
alan, {(alan, 1),(alan, 3)}, {(alan, 5)}
bob, {(bob, 9),}, {}
john, {}, {(john, 3)}
```

Using Types

- By default Pig treats data as un-typed.
- User can declare types of data at load time.

```
log = LOAD 'shakespeare_freq'
        AS (freq:int, word: chararray);
```

- If data type is not declared but script treats value as a certain type, Pig will assume it is of that type and cast it.

```
log      = LOAD 'shakespeare_freq' AS (freq, word);
weighted = FOREACH log
           GENERATE freq * 100; --freq cast to int
```

Custom Load & Store

- By default Pig assumes data is tab separated UTF-8.
- User can set a different delimiter.
- If you have a different format, you use another load/store function to handle de/serialization of the data

```
A = LOAD 'data.json'
    USING PigJsonLoader();
…
STORE INTO 'output.json'
    USING PigJsonLoader();
```

User Defined Functions

- For logic that cannot be done in Pig.
- Can be used to do column transformation, filtering, ordering, custom aggregation.
- For example, you want to write custom logic to do user session analysis:

```
log  = LOAD 'excite-small.log'
        AS (user, time, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE
        group, SessionAnalysis(log);
STORE cntd INTO 'output';
```

Nested Operations

- A FOREACH can apply a set of operators to every tuple in turn.
- Using the previous session analysis example, assume the UDF requires input to be sorted by time:

```
log  = LOAD 'excite-small.log'
        AS (user, time, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd {
       srtd = ORDER log BY time;
       GENERATE group, SessionAnalysis(srtd);
}
STORE cntd INTO 'output';
```

Split

- Data flow need not be linear:

```
A = LOAD 'data';
B = GROUP A BY $0;
…
C = GROUP A by $1;
…
```

- Split can be used explicitly:

```
A = LOAD 'data';
SPLIT A INTO NEG IF $0 < 0, POS IF $0 > 0;
```

- `B = FOREACH NEG GENERATE …`

```
…
```

Pig Commands

Pig Command	What it does
load	Read data from file system.
store	Write data to file system.
foreach	Apply expression to each record and output one or more 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	Join two or more inputs based on a key.
order	Sort records based on a key.
distinct	Remove duplicate records.
union	Merge two data sets.
split	Split data into 2 or more sets, based on filter conditions.
stream	Send all records through a user provided binary.
dump	Write output to stdout.
limit	Limit the number of records.

Example I: More Details

Read file from HDFS

The input format (text, tab delimited)

Define run-time schema

```
raw = LOAD 'excite.log' USING PigStorage('\t') AS (user, id, time, query);

clean1 = FILTER raw BY id > 20 AND id < 100;

clean2 = FOREACH clean1 GENERATE
            user, time,
            org.apache.pig.tutorial.sanitze(query) as query;

user_groups = GROUP clean2 BY (user, query);

user_query_counts = FOREACH user_groups
      GENERATE group, COUNT(clean2), MIN(clean2.time), MAX(clean2.time);

STORE user_query_counts INTO 'uq_counts.csv' USING PigStorage(',');
```

Filter the rows on predicates

For each row, do some transformation

Grouping of records

Compute aggregation for each group

Store the output in a file

Text, Comma delimited

Pig: Language Features

- **Keywords**
  - Load, Filter, Foreach Generate, Group By, Store, Join, Distinct, Order By, …
- **Aggregations**
  - Count, Avg, Sum, Max, Min
- **Schema**
  - Defines at query-time not when files are loaded
- UDFs
- Packages for common input/output formats

Example 2

Script can take arguments

Data are “ctrl-A” delimited

Define types of the columns

```
A = load '$widerow' using PigStorage('\u0001') as (name: chararray, c0: int, c1: int, c2: int);
B = group A by name parallel 10;
C = foreach B generate group, SUM(A.c0) as c0, SUM(A.c1) as c1, SUM(A.c2) as c2;
D = filter C by c0 > 100 and c1 > 100 and c2 > 100;
store D into '$out';
```

Specify the need of 10 reduce tasks

Example 3: Re-partition Join

Register UDFs & custom inputformats

Function the jar file to read the input file

```
register pigperf.jar;
A = load 'page_views' using
org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
        as (user, action, timespent, query_term, timestamp,
estimated_revenue);
B = foreach A generate user, (double) estimated_revenue;
alpha = load 'users' using PigStorage('\u0001') as (name,
phone, address, city, state, zip);
beta = foreach alpha generate name, city;
C = join beta by name, B by user parallel 40;
D = group C by $0;
E = foreach D generate group, SUM(C.estimated_revenue);
store E into 'L3out';
```

Load the second file

Join the two datasets (40 reducers)

Group after the join (can reference columns by position)

This grouping can be done in the same map-reduce job because it is on the same key (Pig can do this optimization)

Example 4: Replicated Join


```
register pigperf.jar;
A = load ‘page_views' using
org.apache.pig.test.udf.storefunc.PigPerformanceLoader()
        as (user, action, timespent, query_term, timestamp,
estimated_revenue);
Big = foreach A generate user, (double) estimated_revenue;
alpha = load ’users' using PigStorage('\u0001') as (name,
phone, address, city, state, zip);
small = foreach alpha generate name, city;
C = join Big by user, small by name using ‘replicated’;
store C into ‘out';
```

Map-only join (the small dataset is the second)

Optimization in joining a big dataset with a small one

Example 5: Multiple Outputs


```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);

DUMP A;
(1,2,3)
(4,5,6)
(7,8,9)

SPLIT A INTO X IF f1<7, Y IF f2==5, Z IF (f3<6 OR f3>6);

DUMP X;
(1,2,3)
(4,5,6)

DUMP Y;
(4,5,6)

STORE x INTO 'x_out';
STORE y INTO 'y_out';
STORE z INTO 'z_out';
```

Split the records into sets

Dump command to display the data

Store multiple outputs

Run independent jobs in parallel


```
D1 = load 'data1' ...

D2 = load 'data2' ...

D3 = load 'data3' ...

C1 = join D1 by a, D2 by b

C2 = join D1 by c, D3 by d
```

C1 and C2 are two independent jobs that can run in parallel

More Links

- Getting Started:
  - http://pig.apache.org/docs/r0.15.0/start.html
- Basics
  - http://pig.apache.org/docs/r0.15.0/basic.html
- Built In Functions
  - http://pig.apache.org/docs/r0.15.0/func.html
- User-Defined Functions
  - http://pig.apache.org/docs/r0.15.0/udf.html
- Control Structures
  - http://pig.apache.org/docs/r0.15.0/cont.html
- Shell and Utility Commands
  - http://pig.apache.org/docs/r0.15.0/cmds.html