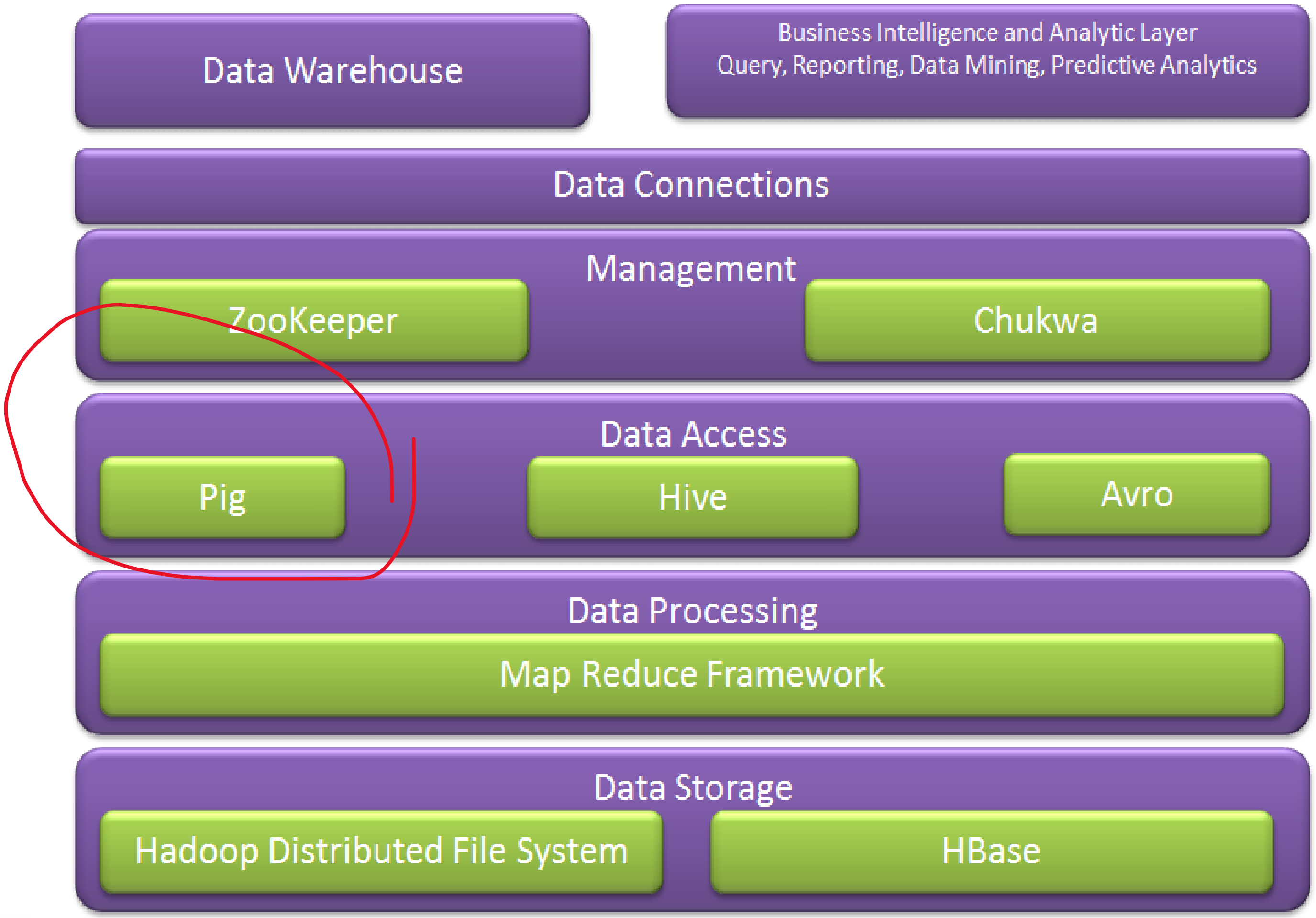
MapReduce High-Level Languages:

PIG

Hadoop Ecosystem



Data Warehouse

Business Intelligence and Analytic Layer
Query, Reporting, Data Mining, Predictive Analytics

Data Connections

Management

ZooKeeper

Chukwa

Data Access

Pig

Hive

Avro

Data Processing

Map Reduce Framework

Data Storage

Hadoop Distributed File System

HBase

Query Languages for Hadoop

- **Java:** Hadoop’s Native Language
- **Pig:** Query and Workflow Language (Yahoo)
- **Hive:** SQL-Based Language (Facebook)
- **HBase:** Column-oriented DB for MapReduce

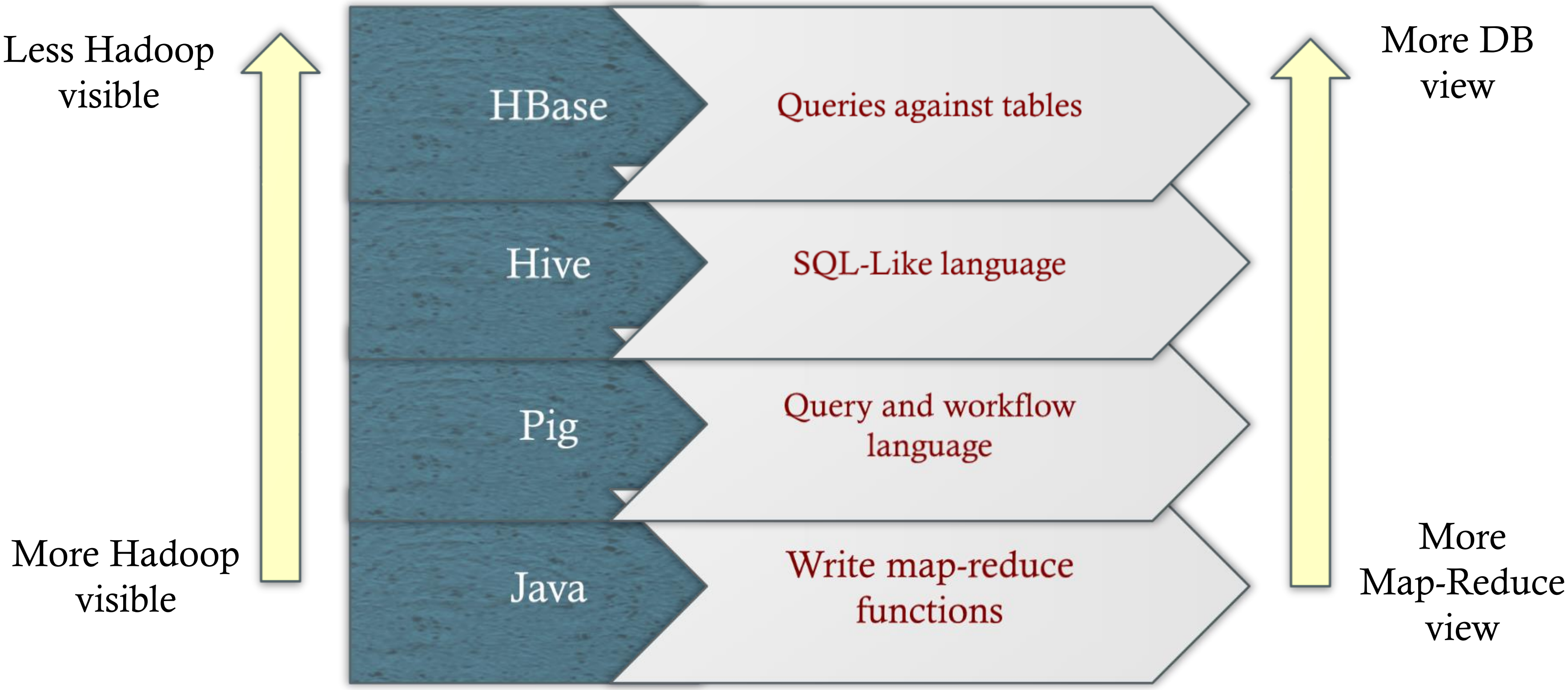
Java is Hadoop's Native Language

- Hadoop itself is written in Java

- **Provided Java APIs**
  - For mappers, reducers, combiners, partitioners
  - Input and output formats

- **Other languages, e.g., Pig or Hive, convert their queries to Java MapReduce code**

Levels of Abstraction



Less Hadoop visible

More Hadoop visible

HBase

Queries against tables

Hive

SQL-Like language

Pig

Query and workflow language

Java

Write map-reduce functions

More DB view

More Map-Reduce view

Apache Pig

(Chapter 16, Hadoop: The Definitive Guide)

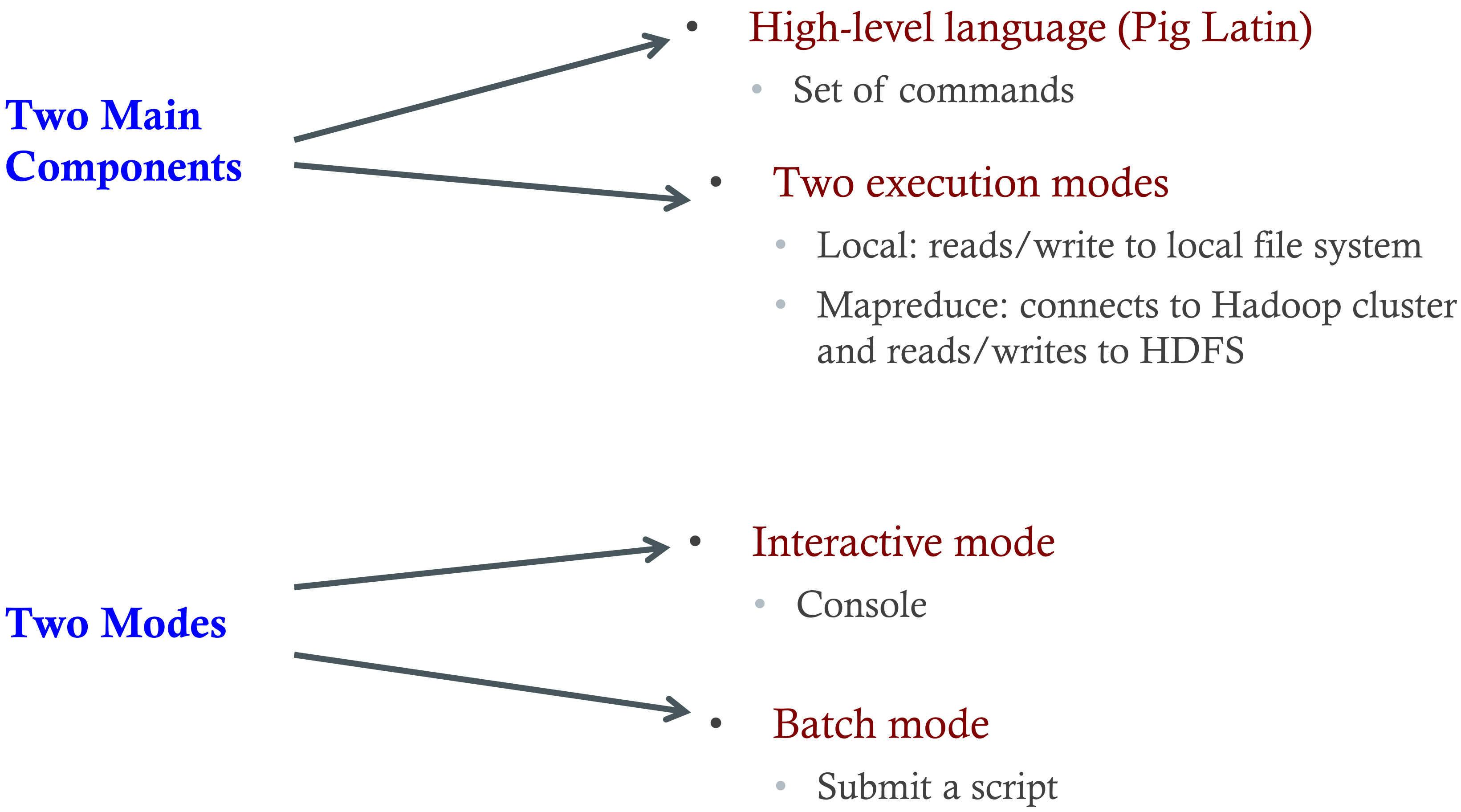
What is Apache Pig ?

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

High-Level Language

```
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(',');
```

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

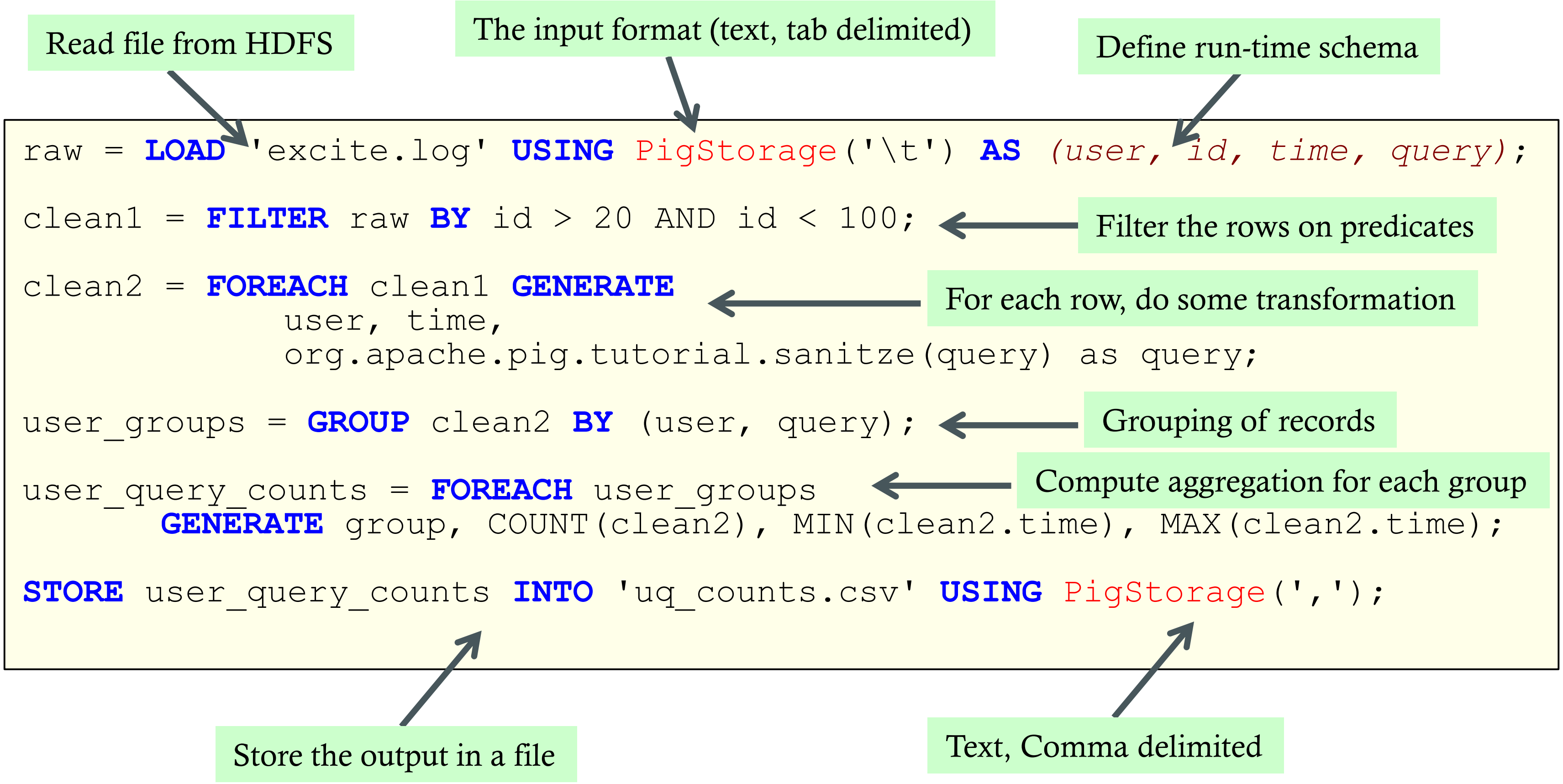
Two Modes

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

Why Language like Pig?

- Common design patterns as KEY WORDS
  - joins, distinct, counts
- Data flow analysis
  - A script can map to multiple map-reduce jobs
- Avoids Java-level errors
  - not everyone can write java code
- Can be interactive mode
  - Issue commands and get results

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

http://pig.apache.org/docs/tutorial.html

Pig: Language Features

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

Example2: Parameterized Template

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,
AVG(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: 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 gene

store E into 'L3out'
```

Load the second file

Join the two datasets (40 reducers)

Group after the join (can reference columns by position)

This join and grouping,
how many map-reduce jobs ?

Example 3: Partition 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);

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 = fore

store E i
```

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
(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

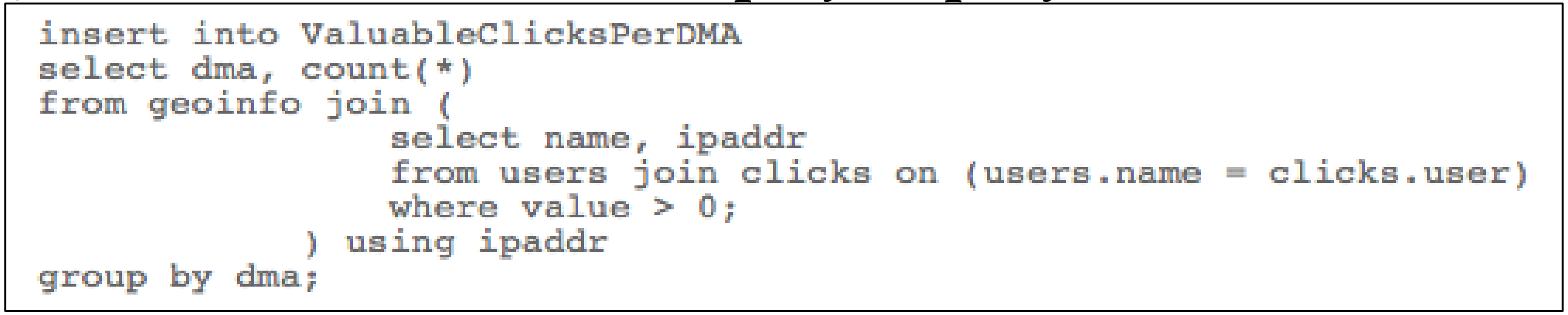
```
D1 = load 'data1' …

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

Pig Latin vs. SQL

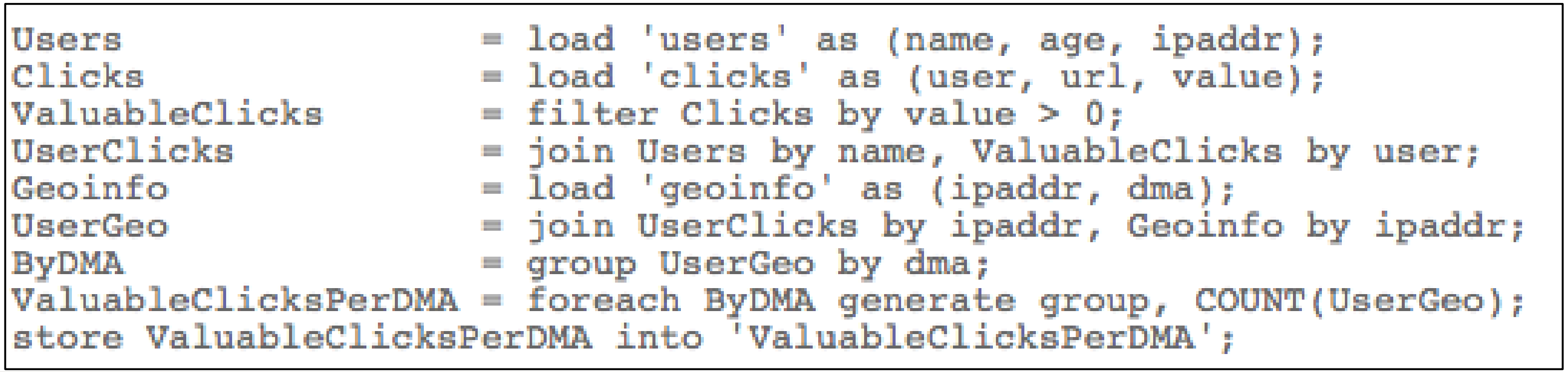
- Pig Latin is procedural (dataflow programming model)
  - Step-by-step query style is easier to write for some
- SQL is declarative but not step-by-step style

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

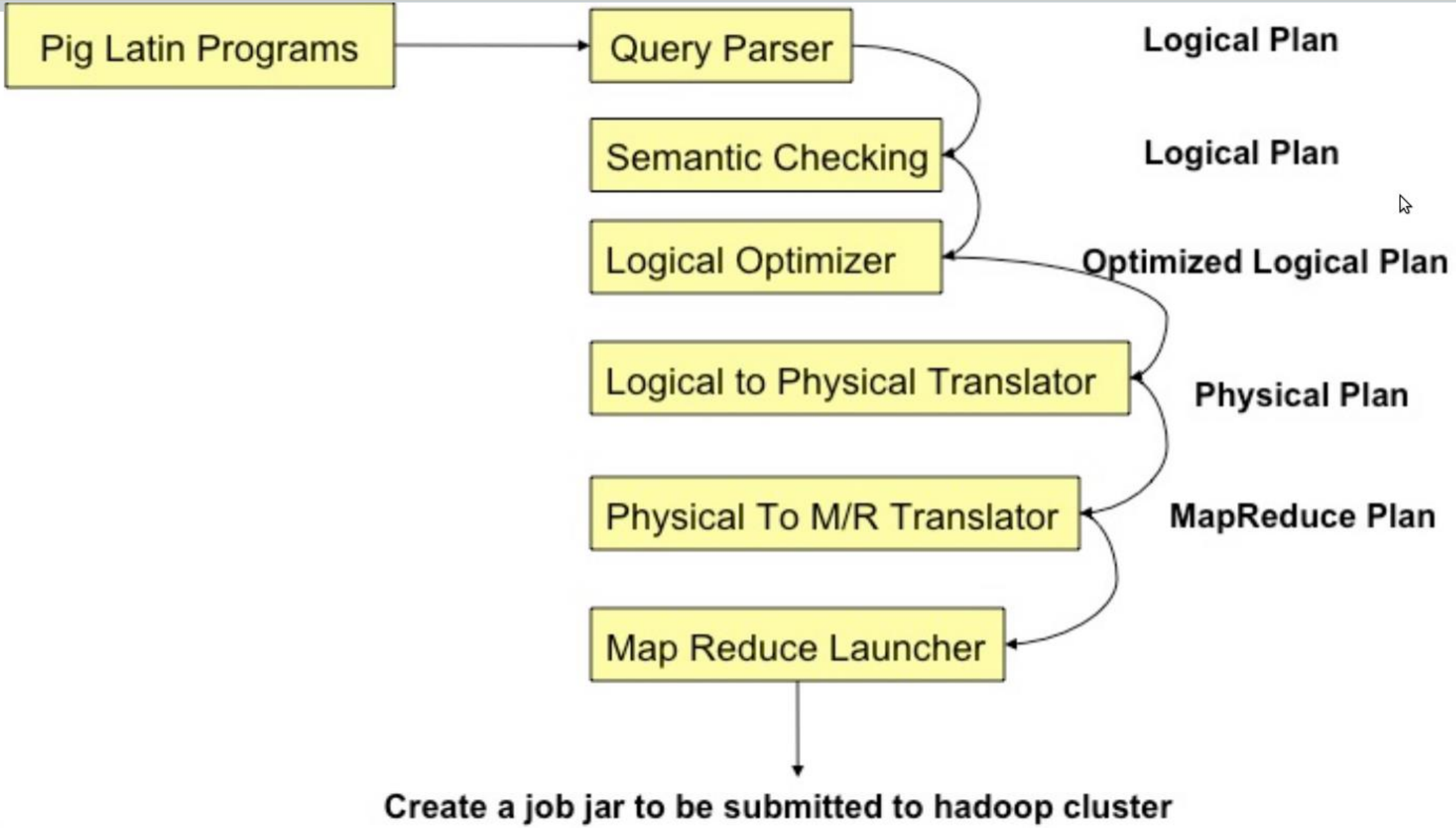


```
Users                = load 'users' as (name, age, ipaddr);
Clicks               = load 'clicks' as (user, url, value);
ValuableClicks       = filter Clicks by value > 0;
UserClicks           = join Users by name, ValuableClicks by user;
Geoinfo              = load 'geoinfo' as (ipaddr, dma);
UserGeo              = 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 by users (via hints)
    - Use optimizer hints (due to the lack of complex optimizers)
- **In SQL:**
  - Query plans are solely decided by the system (powerful opt)
  - Data cannot be stored in the middle (or, at least not user-accessible)
  - Schema and data types are defined at the creation time

Pig Compilation



Pig Latin Programs

Query Parser

Logical Plan

Semantic Checking

Logical Plan

Logical Optimizer

Optimized Logical Plan

Logical to Physical Translator

Physical Plan

Physical To M/R Translator

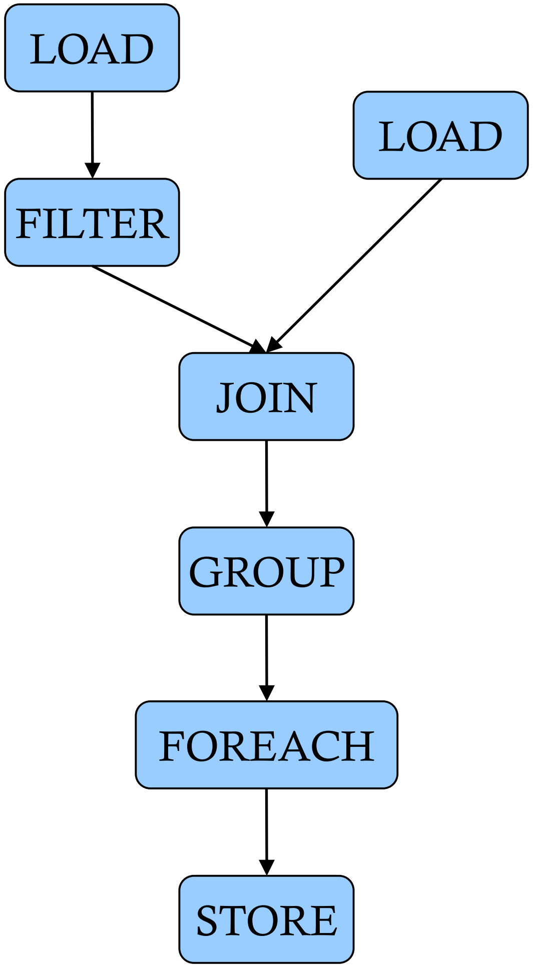
MapReduce Plan

Map Reduce Launcher

Create a job jar to be submitted to hadoop cluster

Logic Plan

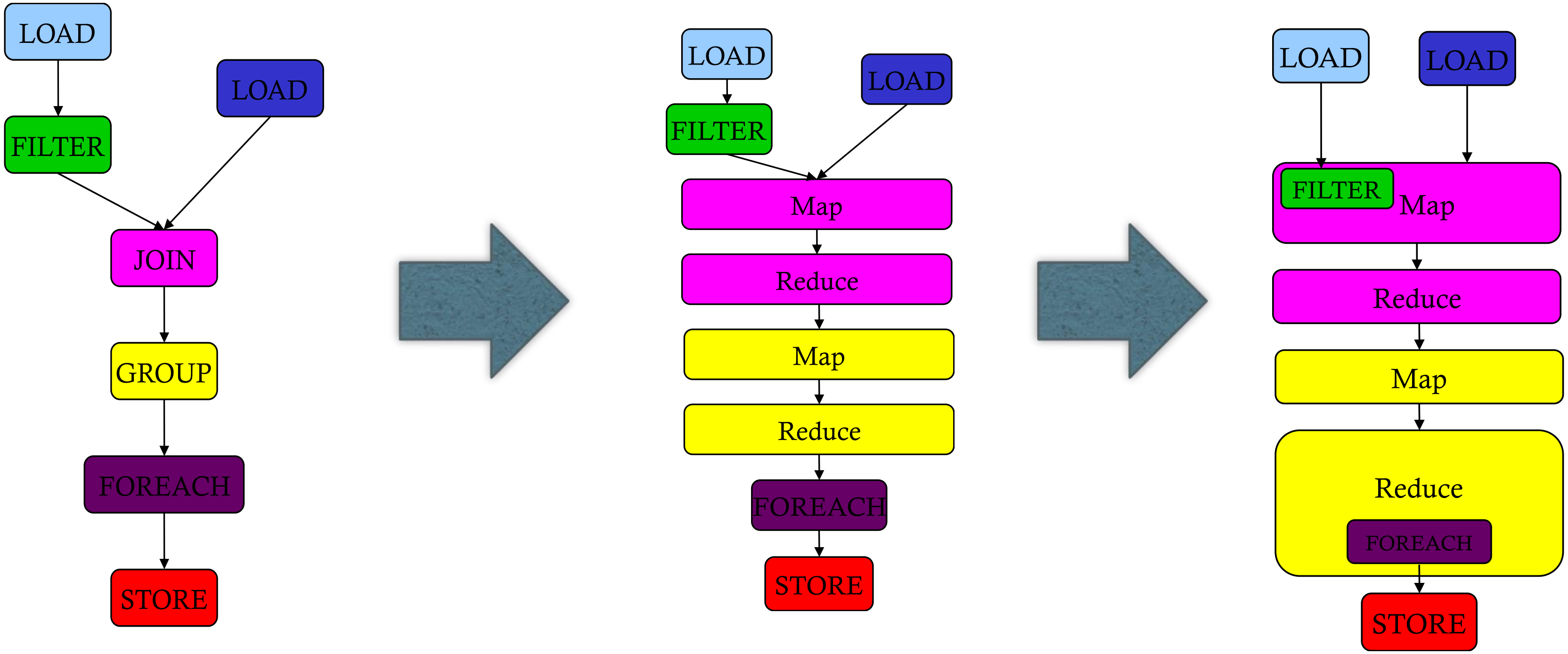
```
A=LOAD 'file1' AS (x, y, z);

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

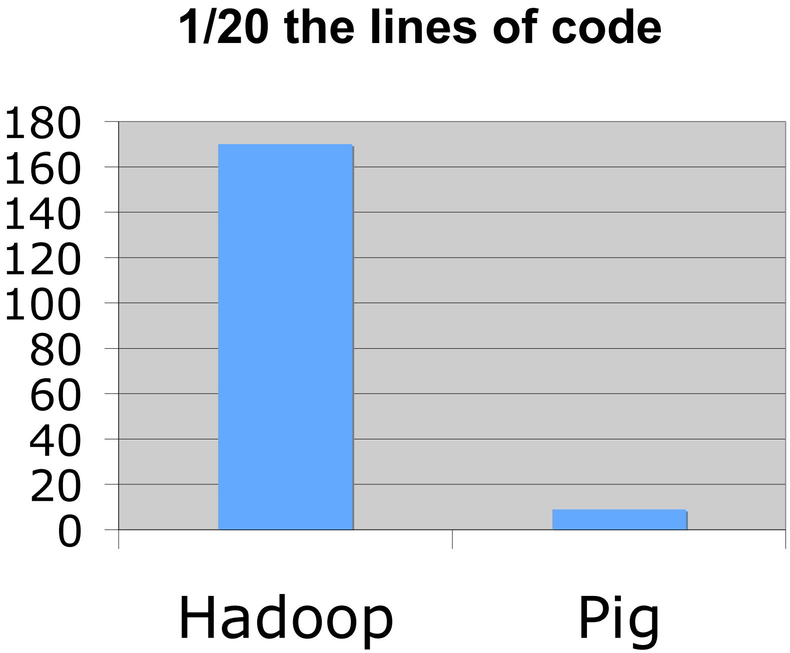
- Mostly 1:1 correspondence with logical plan
- **Except for:**
  - Join, Distinct, (Co)Group, Order
- Some optimizations are done automatically

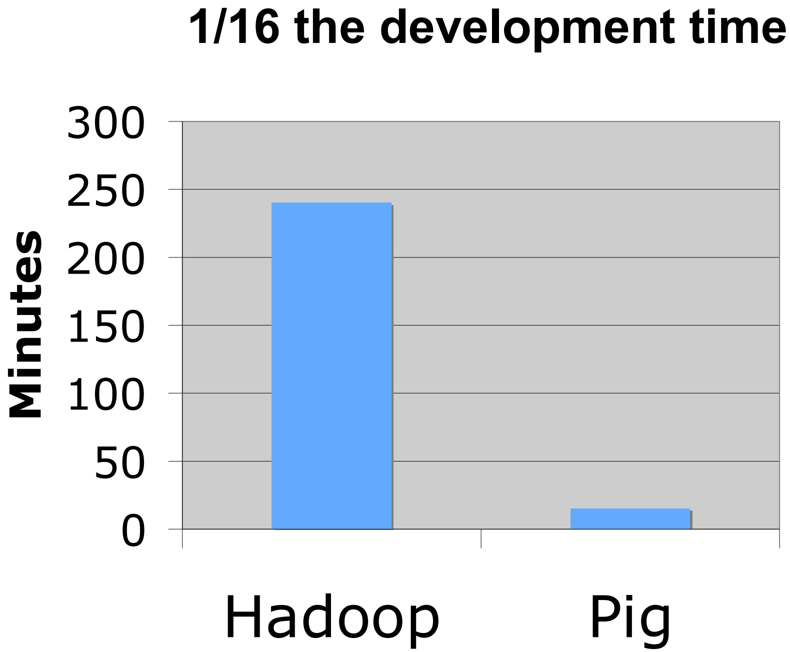
Generation of Physical Plans



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

Java vs. Pig





Performance is comparable
(Java is slightly better)

Pig References

- **Pig Tutorial**
  - http://pig.apache.org/docs/r0.7.0/tutorial.html
- **Pig Latin Reference Manual 2**
  - http://pig.apache.org/docs/r0.7.0/piglatin_ref1.html
- **Pig Latin Reference Manual 2**
  - http://pig.apache.org/docs/r0.7.0/piglatin_ref2.html
- **PigMix Queries**
  - https://hpccsystems.com/why-hpcc-systems/benchmarks/pigmix-hpcc

Apache Pig