CSE 487/587
Data Intensive Computing

Lecture 8: Pig-Latin

Vipin Chaudhary

vipin@buffalo.edu

716.645.4740
305 Davis Hall

Overview of Today's Lecture

- Pig Latin (lot of stuff borrowed from Yahoo Research!)
- Midterm
- Review

Problems with Mapreduce?

- Restricted programming model
  - Only two (or three) phases
  - Job chain for long data flow
- Too many lines of code even for simple logic
  - How many lines do you have for word count?
  - Programmers are responsible for this

Pig to the Rescue

- What is Pig?
  - open-source high-level dataflow system
  - Provides a simple language for queries and data manipulation,
  - Compiled into map-reduce jobs that are run on Hadoop
  - Combines the high-level data manipulation constructs of SQL with the procedural programming of map-reduce
    - Systems like Netezza, Teradata, Oracle Cluster DB are $$$$$

- Why is it important?
  - Social media and cloud companies like Yahoo, Google and Microsoft are collecting enormous data
  - Some form of ad-hoc processing and analysis of all of this information is required

Where does Pig Fit?

RDBMS

- Parallel DB expensive
- Rigid schemas
- Sql query construction

Best of SQL and Map-Reduce

COMBINE high-level declarative querying with
low-level procedural programming

Mapreduce

- Custom code for common tasks
- Complex workarounds for many tasks that do not fit map->combine->reduce

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
- Extensive UDF support
  - Available in Java, Python and Javascript currently
  - Can be written for filtering, grouping, per-tuple processing, loading and storing
  - Use Piggy Bank – repository of Java UDFs written by other users.

Data Model

- Supports four basic types
  - Atom: a simple atomic value (*int*, *long*, *double*, *string*)
    - ex: ‘John’
  - Tuple: a sequence of fields that can be any of the data types
    - ex: (‘John’, 21)
  - Bag: a collection of tuples of potentially varying structures, can contain duplicates
    - ex: {(‘John’), (‘Mike’, (21, 25))}
  - Map: an associative array, the key must be a *chararray* but the value can be any type

Pig Latin Example

Table urls: (url,category, pagerank)

Find for each sufficiently large category, the average pagerank of high-pagerank urls in that category

SQL:

```
SELECT category,  AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

Pig Latin:

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
             category, AVG(good_urls.pagerank);
```

Give me top 10 URLs

```
groups = GROUP urls BY category
Output = FOREACH groups GENERATE
             category, top10(urls)
```

Pig Latin Example

<u>Set of URLs of pages classifieds as spam but have high pagerank score.</u>

```
Spam_urls = FILTER urls BY isSpam(url)
Culprit_urls = FILTER spam_urls BY pagerank > 0.8
```

<u>Give me top 10 URLs</u>

```
groups = GROUP urls BY category
Output = FOREACH groups GENERATE
            category, top10(urls)
```

Data Model

- Atom - simple atomic value (ie: number or string)
- Tuple
- Bag
- Map

'alice'

Data Model

- Atom - simple atomic value (ie: number or string)
- Tuple – sequence of fields; each field any type
- Bag
- Map

```
('alice', 'lakers')
```

Data Model

- Atom - simple atomic value (ie: number or string)
- Tuple – sequence of fields; each field any type
- Bag – Collection of tuples
    - Duplicates possible
    - Tuples in a bag can have different field lengths and field types
- Map

$$\left\{ \begin{array}{c} (\texttt{`alice'}, \texttt{`lakers'}) \\ (\texttt{`alice'}, (\texttt{`iPod'}, \texttt{`apple'})) \end{array} \right\}$$

Data Model

- Atom - simple atomic value (ie: number or string)
- Tuple – sequence of fields; each field any type
- Bag – Collection of tuples
  - Duplicates possible
  - Tuples in a bag can have different field lengths and field types
- Map – Collection of key-value pairs
  - Key is an atom; value can be any type

$$
\left[\begin{array}{c}
\text{'fan of'} \rightarrow \left\{\begin{array}{c} (\text{'lakers'}) \\ (\text{'iPod'}) \end{array}\right\} \\
\text{'age'} \rightarrow 20
\end{array}\right]
$$

Key ‘fan of’ is mapped to a bag containing two tuples, and key ‘age’ is mapped to an atom 20

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

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

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

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

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

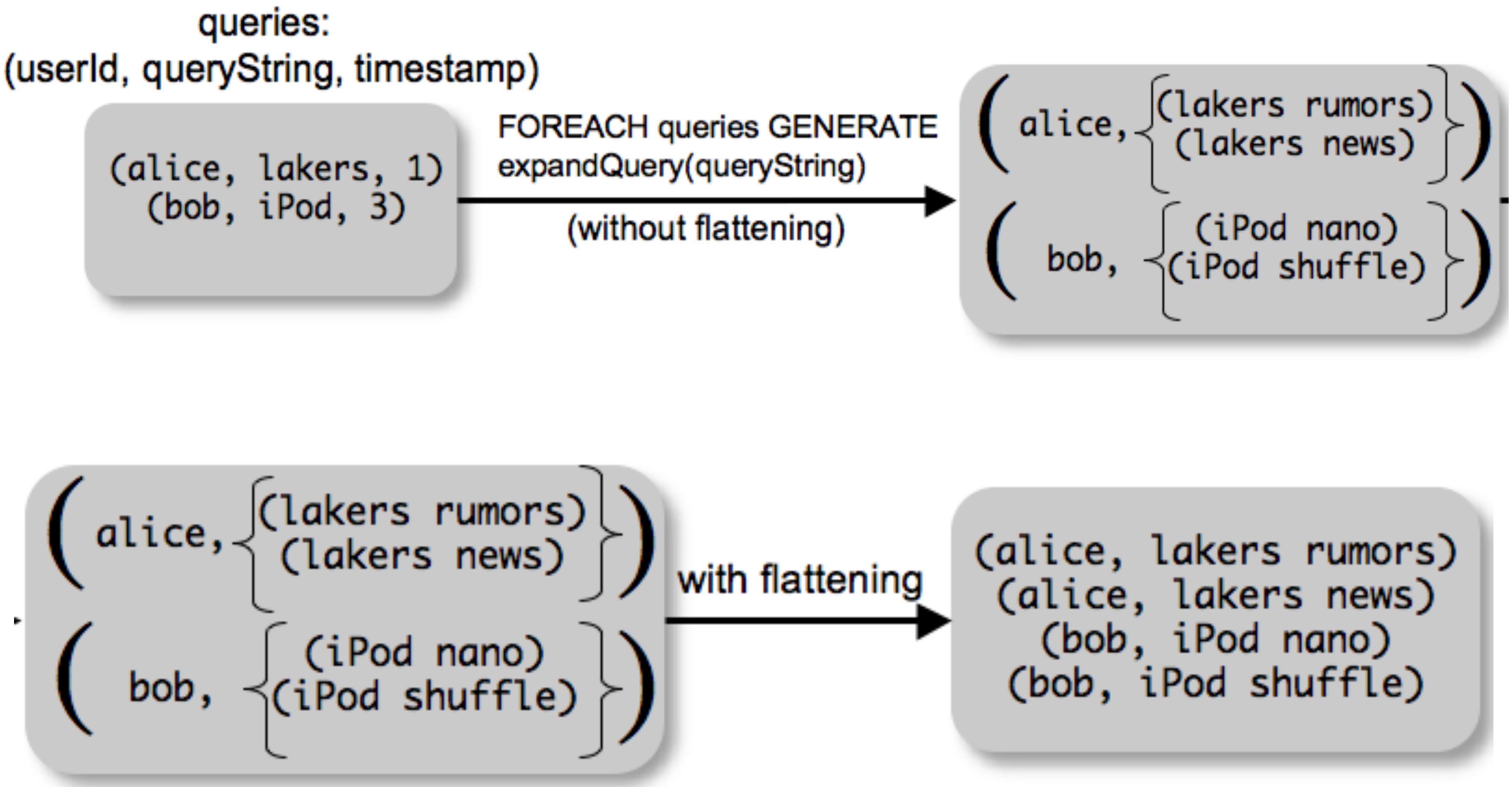
Data Model

User-Defined Functions (UDFs)

Ex: `spam_urls = FILTER urls BY isSpam(url);`

- Can be used in many Pig Latin statements
- Useful for custom processing tasks
- Can use non-atomic values for input and output
- Currently must be written in Java, Python, Javascript

Flatten


queries:
(userId, queryString, timestamp)
(alice, lakers, 1)
(bob, iPod, 3)
FOREACH queries GENERATE
expandQuery(queryString)
(without flattening)
alice,
(lakers rumors)
(lakers news)
bob,
(iPod nano)
(iPod shuffle)
alice,
(lakers rumors)
(lakers news)
bob,
(iPod nano)
(iPod shuffle)
with flattening
(alice, lakers rumors)
(alice, lakers news)
(bob, iPod nano)
(bob, iPod shuffle)

Load

- Input is assumed to be a bag (sequence of tuples)
- Can specify a deserializer with "USING"
- Can provide a schema with "AS"

```
newBag = LOAD 'filename'
   <USING functionName() >
   <AS (fieldName1, fieldName2,…)>;

Queries = LOAD 'query_log.txt'
          USING myLoad()
          AS (userID,queryString, timeStamp)
```

FOREACH

- Apply some processing to each tuple in a bag
- Each field can be:
  - A fieldname of the bag
  - A constant
  - A simple expression (like f1+f2)
  - A predefined function (like, SUM, AVG, COUNT, FLATTEN)
  - A UDF

```
newBag =
  FOREACH bagName
  GENERATE field1, field2, …;
```

FILTER

Select subset of tuples in a bag

```
newBag =  FILTER bagName
          BY   expression
          ;
```

Expression uses simple comparison operators (==, !=, <, >, …) and Logical connectors (AND, NOT, OR)

```
some_apples =
   FILTER apples BY colour != 'red';
```

Can use UDFs

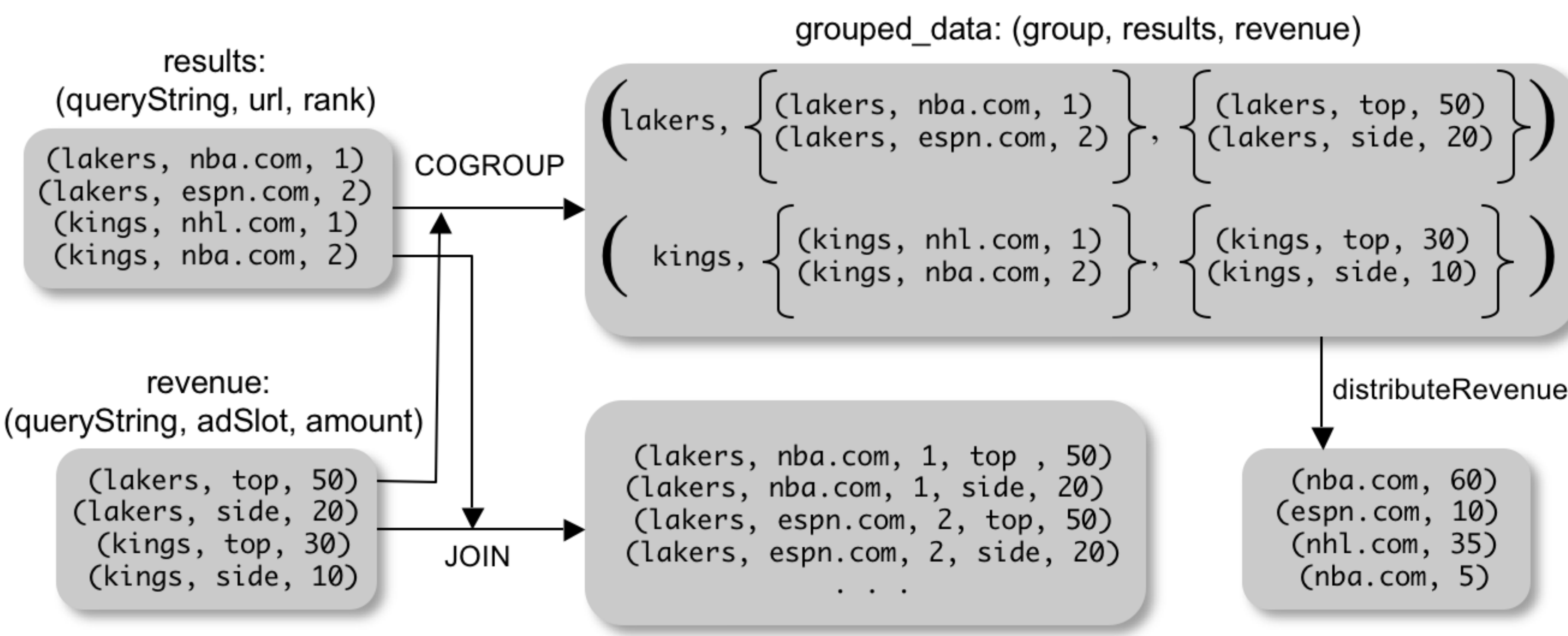
```
some_apples =
   FILTER apples BY NOT isRed(colour);
```

COGROUP

Group two datasets together by a common attribute

Groups data into nested bags

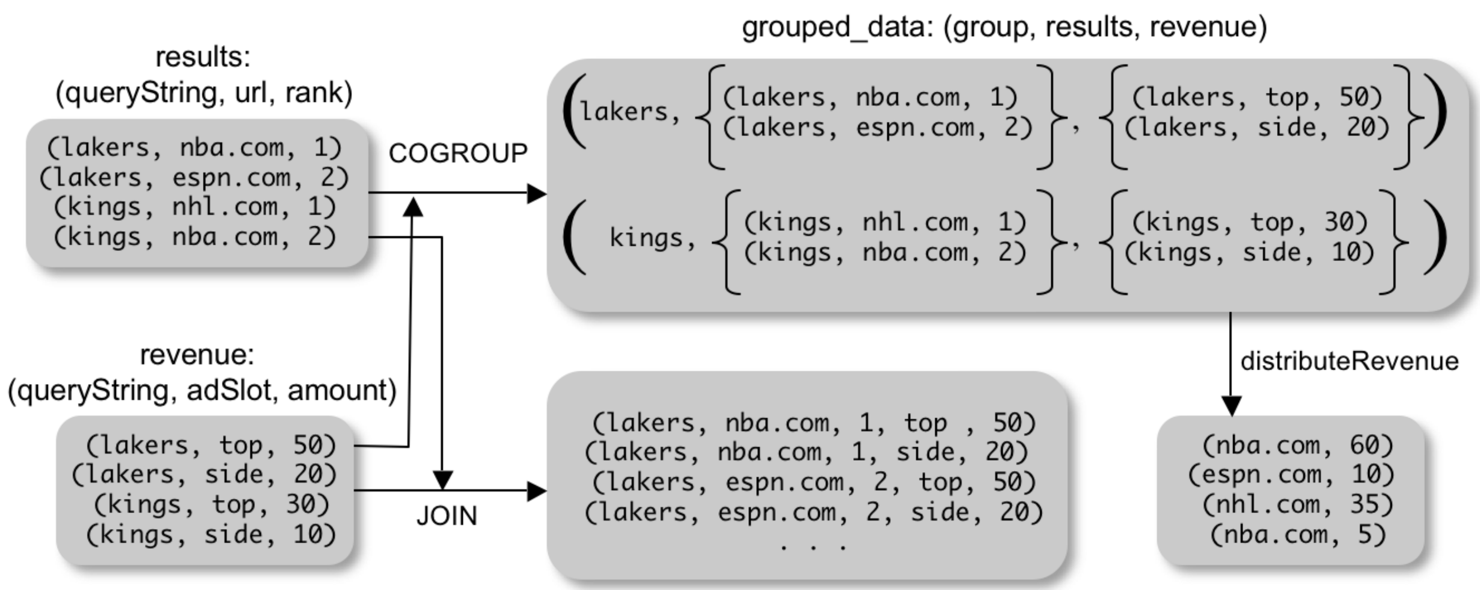
```
grouped_data = COGROUP results BY queryString,
                       revenue BY queryString;
```



COGROUP vs JOIN?

```
url_revenues =
        FOREACH grouped_data GENERATE
        FLATTEN(distributeRev(results, revenue));
```

- COGROUP takes advantage of nested data structure (combination of GROUP BY and JOIN)
- User can choose to go through with cross-product for a join or perform aggregation on the nested bags



COGROUP vs JOIN?

To process nested bags of tuples before cross product.

JOIN keyword is still available:

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

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

join_result = FOREACH temp GENERATE
              FLATTEN(results), FLATTEN(revenue);
```

STORE (DUMP)

Output data to a file (or screen)

```
STORE bagName INTO 'filename'
     <USING deserializer ()>;
```

Few Other Commands

UNION - return the union of two or more bags

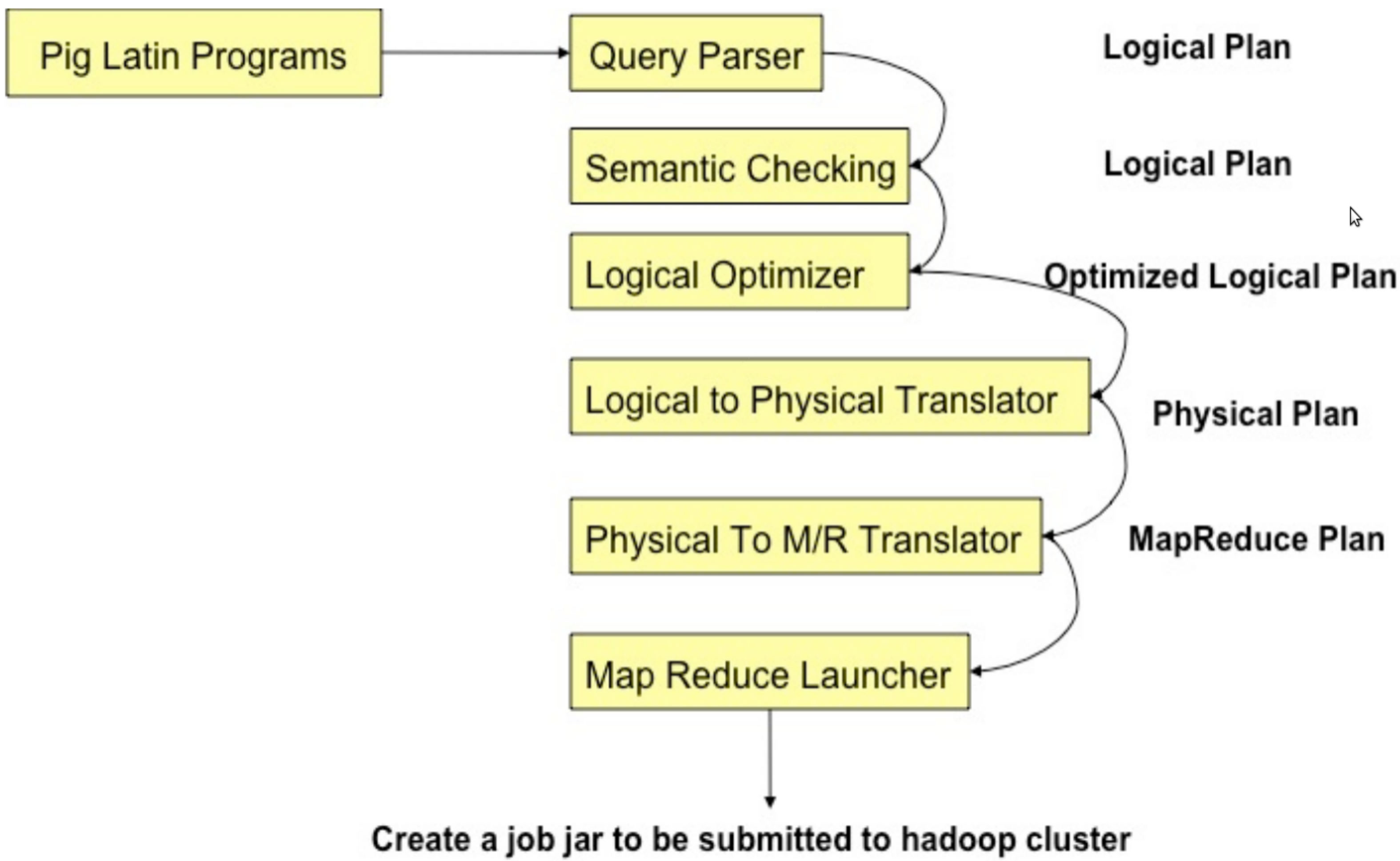
CROSS - take the cross product of two or more bags

ORDER - order tuples by a specified field(s)

DISTINCT - eliminate duplicate tuples in a bag

LIMIT - Limit results to a subset

Compilation



Compilation

Pig system does two primary tasks:

- Builds a Logical Plan from a Pig Latin script
  - Supports execution platform independence
  - No processing of data performed at this stage
- Compiles the Logical Plan to a Physical Plan and Executes
  - Convert the Logical Plan into a series of Map-Reduce statements to be executed by Hadoop Map-Reduce

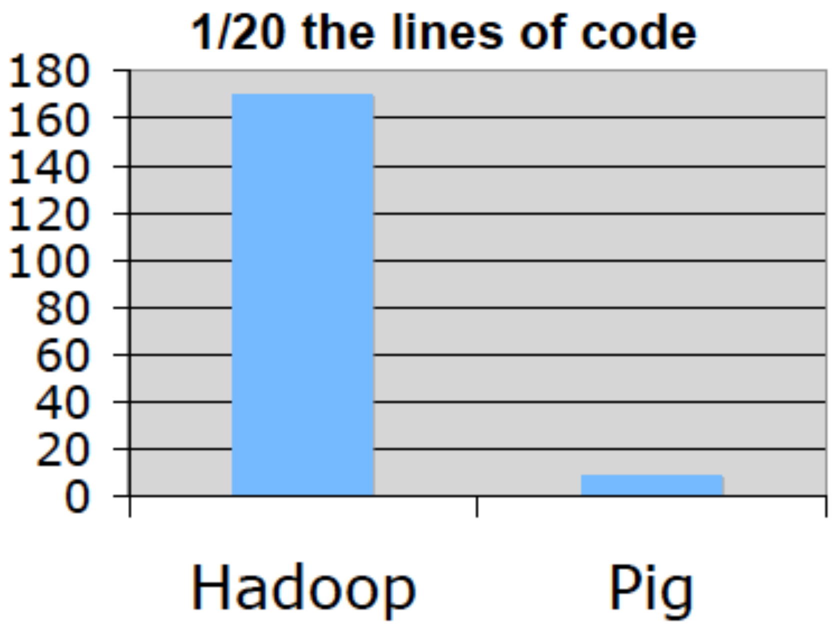
Streaming

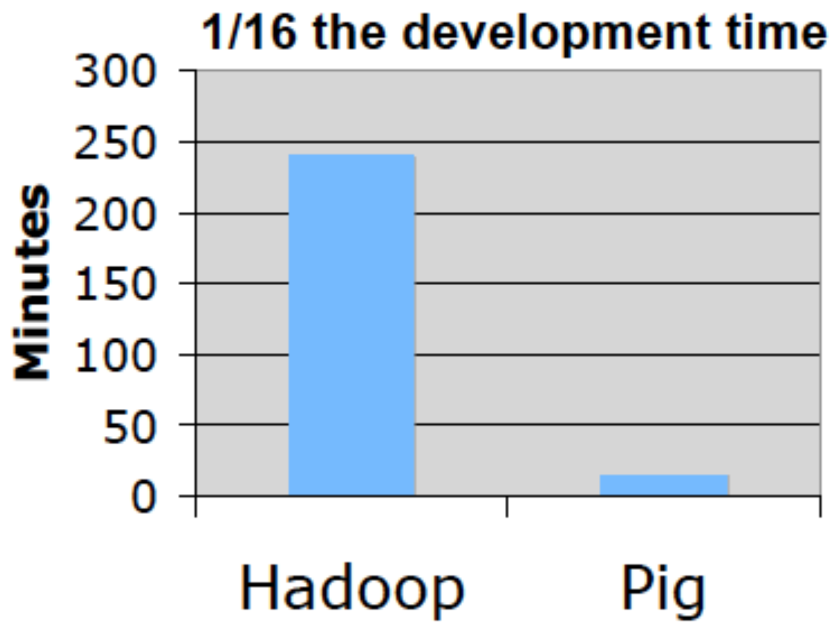
- Allows data to be pushed through external executables
- Example:

```
A = LOAD 'data';
B = STREAM A THROUGH 'stream.pl -n 5';
```

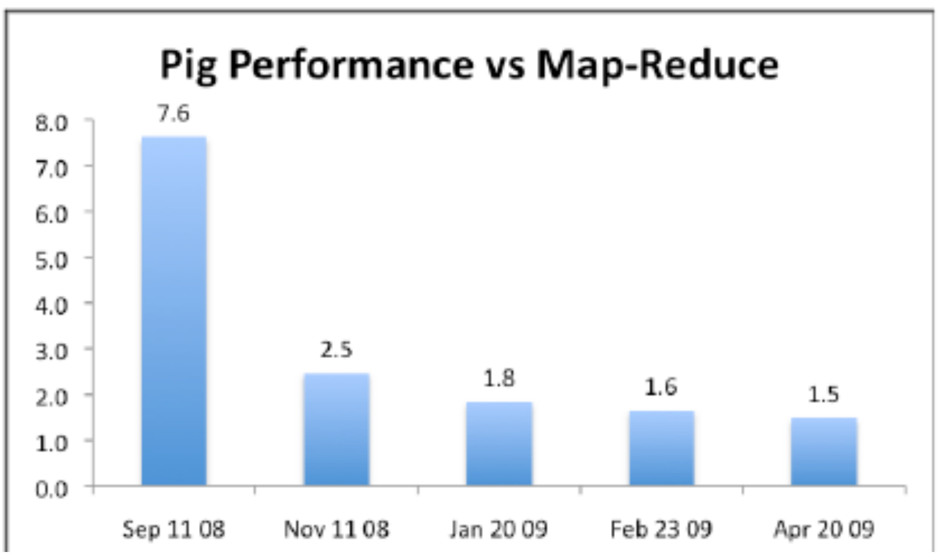
- Due to asynchronous behavior of external executables, each STREAM operator will create two threads for feeding and consuming data from external executables.

Mapreduce vs Pig





performance 1.5x Hadoop



Take Home portion of Midterm#1

- Compute the total cost of building a data center that houses 10 ExaBytes of HDD
  - HDDs, networking, racks, building (real-estate), power, cooling, …
  - You can get the cost of various parts online
  - Assume the datacenter is built in Buffalo
  - State all your assumptions with reference for pricing, etc.
  - Due March 11 at 11:59PM on UBLearns.
  - 25% of your grade for midterm

Midterm#1

- Parallel/Distributed File Systems
- Statistical Methods
- Some classification
- Mapreduce programming
- R programming
- Pig