MapReduce/PigLatin

Baraa MOHAMAD
Laboratory: LIMOS

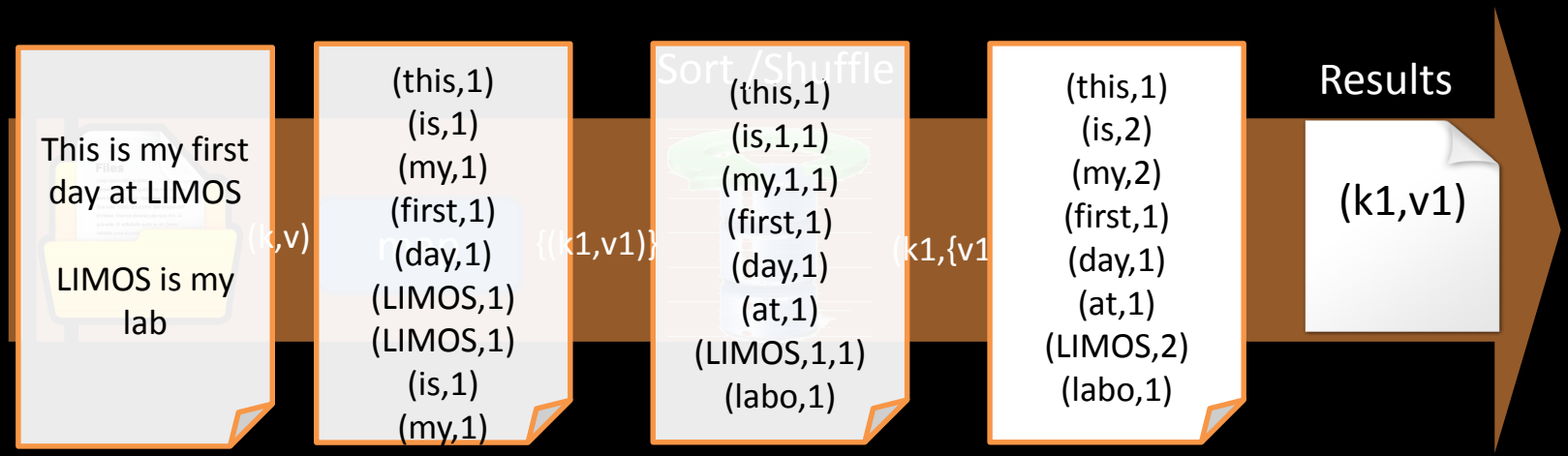
Advisors:
Dr. Farouk Toumani (University Blaise Pascal, France)
Dr. Le Gruenwald (University of Oklahoma, USA)
Dr. Laurent d'Orazio (University Blaise Pascal, France)

MapReduce

- Definition [1]
  - Programming model for processing and generating large data sets.
- Origin
  - The idea of Map and Reduce is over 40 years old presented in Functional Programming Languages e.g: Lisp.
- Goal
  - Enables automatic distribution of large-scale computations, on large clusters of commodity PCs

MapReduce

How does it work?



- **Map:** extract desired information, take a set of (key, value) pairs and generate a set of intermediate (key, value) pairs by applying some function ***f*** to all these pairs.
- **Reduce:** merge all pairs with the same key applying a reduction function ***R*** on the values
- ***f*** and ***R*** are user defined functions

MapReduce – Word Count

- Problem
  - counting the number of occurrences of each word in a large collection of documents.

```
map(String key, String value)
      // key: document name
      // value: document contents
       for each word w in value:
           EmitIntermediate(w, "1");


reduce(String key, Iterator values):
      // key: a word
      // values: a list of counts
      int result = 0;
      for each v in values
        result += ParseInt(v);
        Emit(AsString(result));
```

MapReduce- Execution Overview

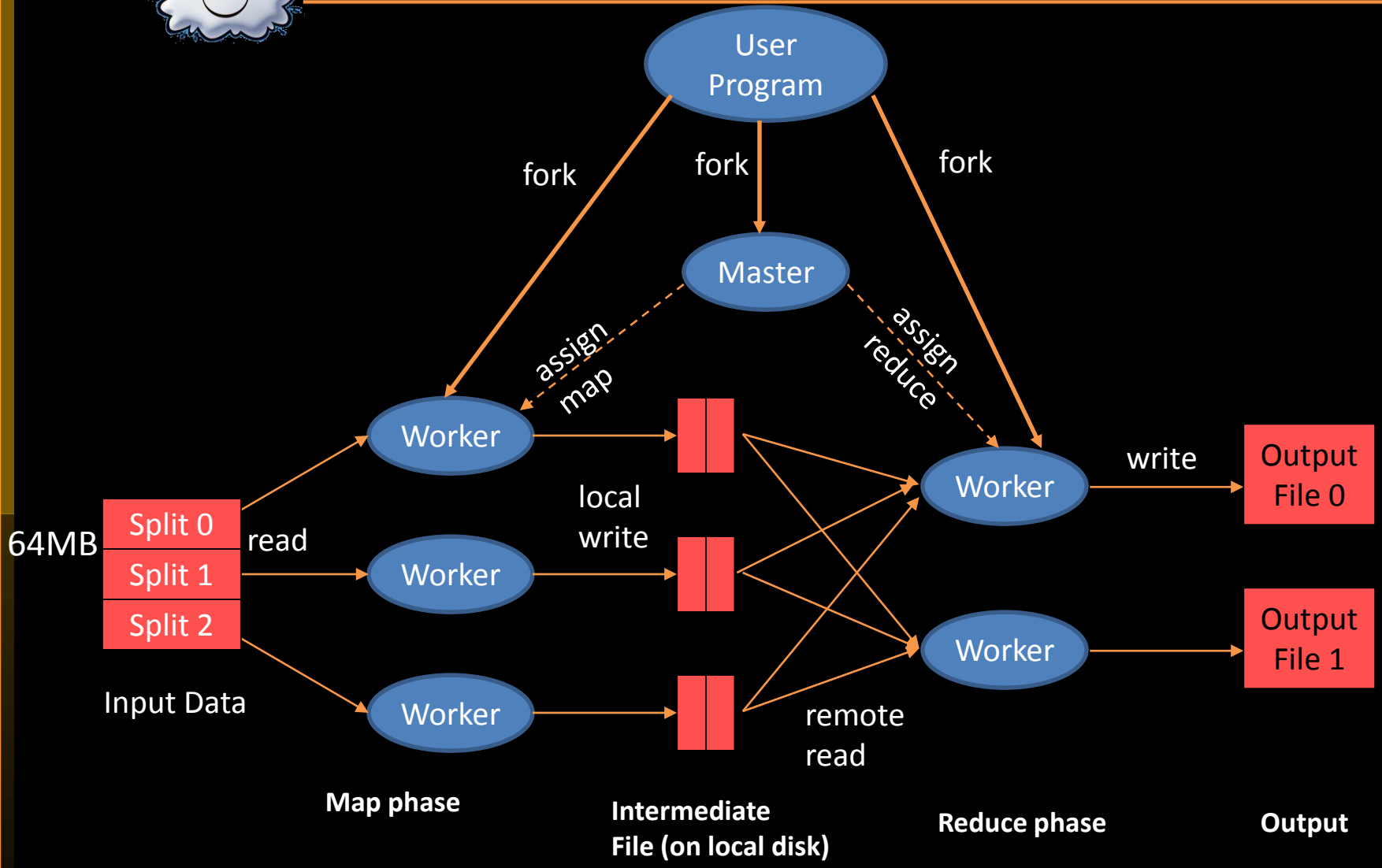


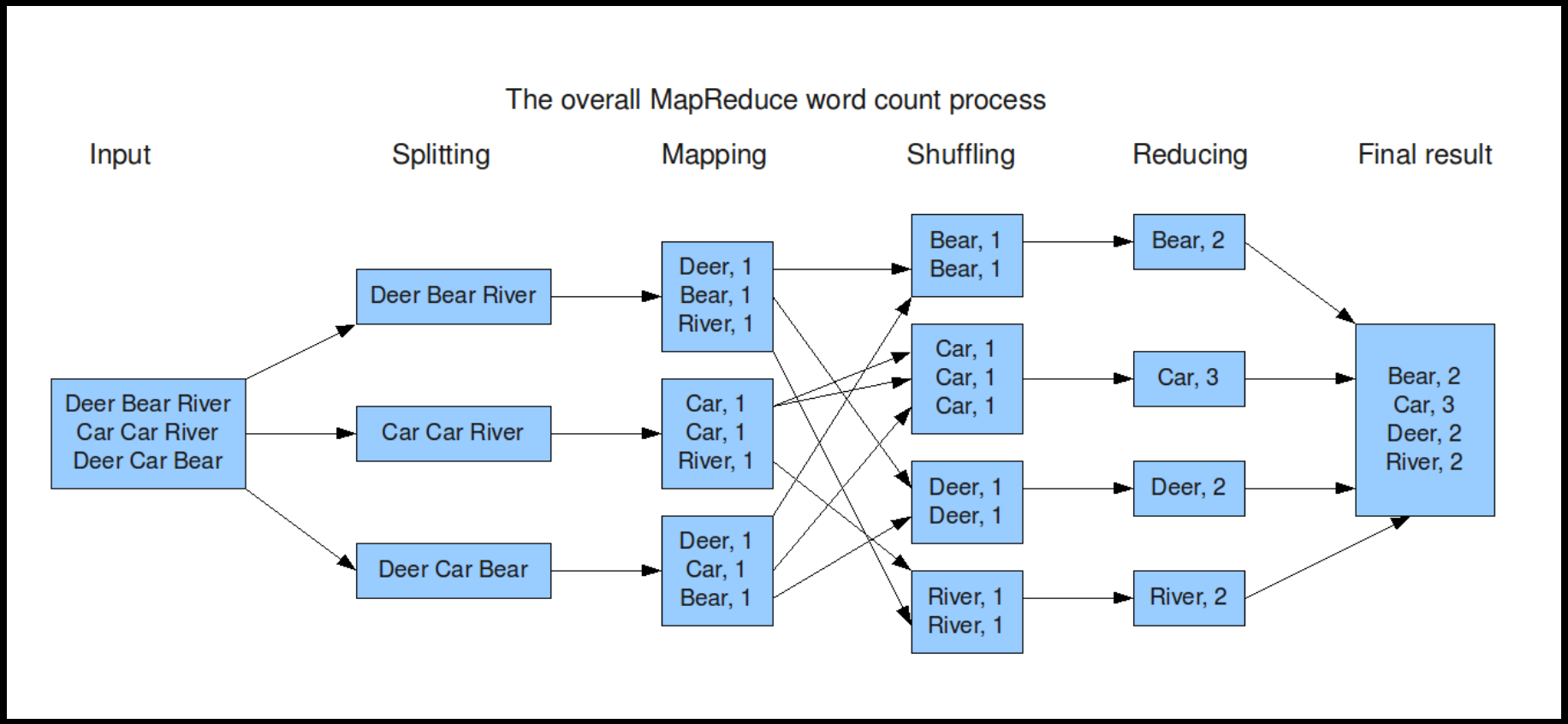
Figure 1: Execution overview [1]

MapReduce - Work Flow

- 1- Input files are split into M pieces
- 2- The master assigns map and reduce tasks to idle slave workers
- 3. Map workers
  - Read input splits,
  - Parse (key, value) pairs
  - Apply the map function
  - Write intermediate output pairs to R region on local disk
  - Pass the locations back to the master
- 4-The master notifies reduce worker about locations
- 5- Reduce workers
  - Read data from local disks of the map workers
  - Group same key records together
  - Apply the reduce function
  - Append the output to a final output file
- 6- When all tasks are completed the master wakes up the user program, which resumes the user code.

MapReduce: Execution with shuffle phase

- The shuffle phase: sorts the resulting (key,value) pairs from the map phase locally by their keys before sending them to a reducer.
- Example: [2]



MapReduce - Master data structures

- Task status: (idle, in-progress, completed)
- Master schedules tasks to Idle workers
  - Stores the identity of worker machines
- Master stores the locations and sizes of the R intermediate files produced by completed map workers
- Master pushes this info to reducers
- Master pings workers periodically to detect failures.

MapReduce - Fault Tolerance

Map worker failure

- Completed or in-progress worker is reset to idle !
  - ✓ All output is stored locally
- Reduce workers are notified when task is rescheduled on another worker

Reduce worker failure

- Only in-progress tasks are reset to idle
  - ✓ All output is stored in the global file system

Master failure

- Computation is aborted, client is notified, retry
- Master writes periodic check points

MapReduce - Locality

- **How the master schedules its tasks:**
  - Asks GFS (Google File System) for locations of replicas of input file blocks
  - The master schedules a map task on a machine that contains a replica of the corresponding input data
    - If failed → schedule a map task near a replica of that task's input data
- **Result**
  - Thousands of machines read input files at local disk speed

MapReduce - Backup Tasks

- **Problem:** in-progress worker takes a lot of time to complete one task
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Processor caches disabled
- **Effect:** Slowdown the computation

- **Solution**: the master schedules backup executions of the remaining, nearly complete, in-progress tasks
  - Task marked completed when whoever finishes it first
- **Result:** Dramatically shortens job completion time

MapReduce - Combiner

- **Problem:**
  - Map task can produce a significant number of repetitions in intermediate keys
  - All counts will be sent over network to Reduce task

- **Solution:**
  - User-defined Combiner function
    - Combines the results of a single Map worker and stores the intermediate results in a local file
    - Combiners are considered as mini-reducers, but they are executed locally on the output of each mapper
    - Partial merging before data is sent over network

- **Effect:**
  - Significant speedup in certain MapReduce operations
  - Save network bandwidth

MapReduce - Skipping Bad Records

- **Problem:**
  - Map/Reduce functions sometimes fail for particular inputs
  - Debug & fix, not always possible
- **Effect:**
  - Map/Reduce functions crash on certain records
  - Prevent a MapReduce operation from completing

- **Solution:**
  - A signal handler (in failed workers) sends a notification packet to the master
    - The packet includes sequence number of record being processed
  - If the master sees more than one failures for the same record → other workers are told to skip the record

MapReduce - Conclusion

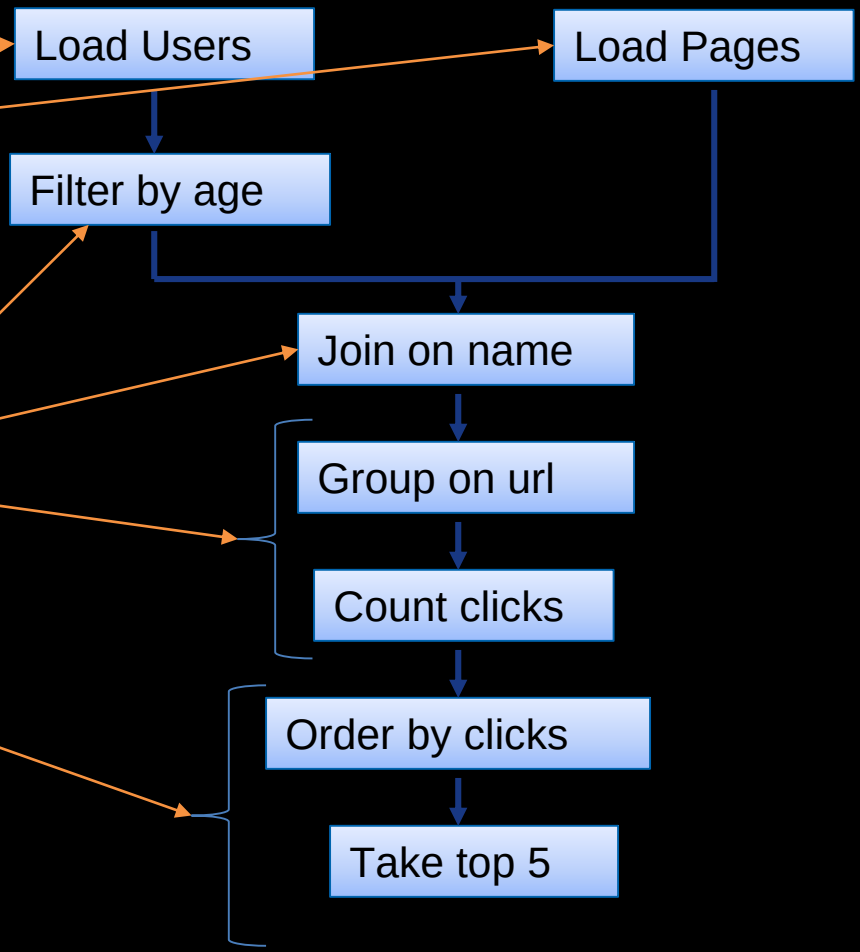
- MapReduce is a useful abstraction
  - ✓ Cluster issues (failures, network problems, slow machines) handled by library
    - ✓ Focus on problem
  - ✓ Greatly simplifies large-scale computations

- ✗ Common operations must be coded by hand
  - Join, filter, projection, aggregates, sorting, distinct
- ✗ Do low-level stuff by hand
  - ✗ Extremely rigid data flow
  - ✗ Hard to understand, maintain, extend, and optimize code

Pig* [3][4]

- **Problem :**

user data in one file,
website data in another;
We want to find the
top 5 most visited pages
by users aged 18 - 25



Images in the following slides are taken from
https://cwiki.apache.org/confluence/display/PIG/PigTalksPapers

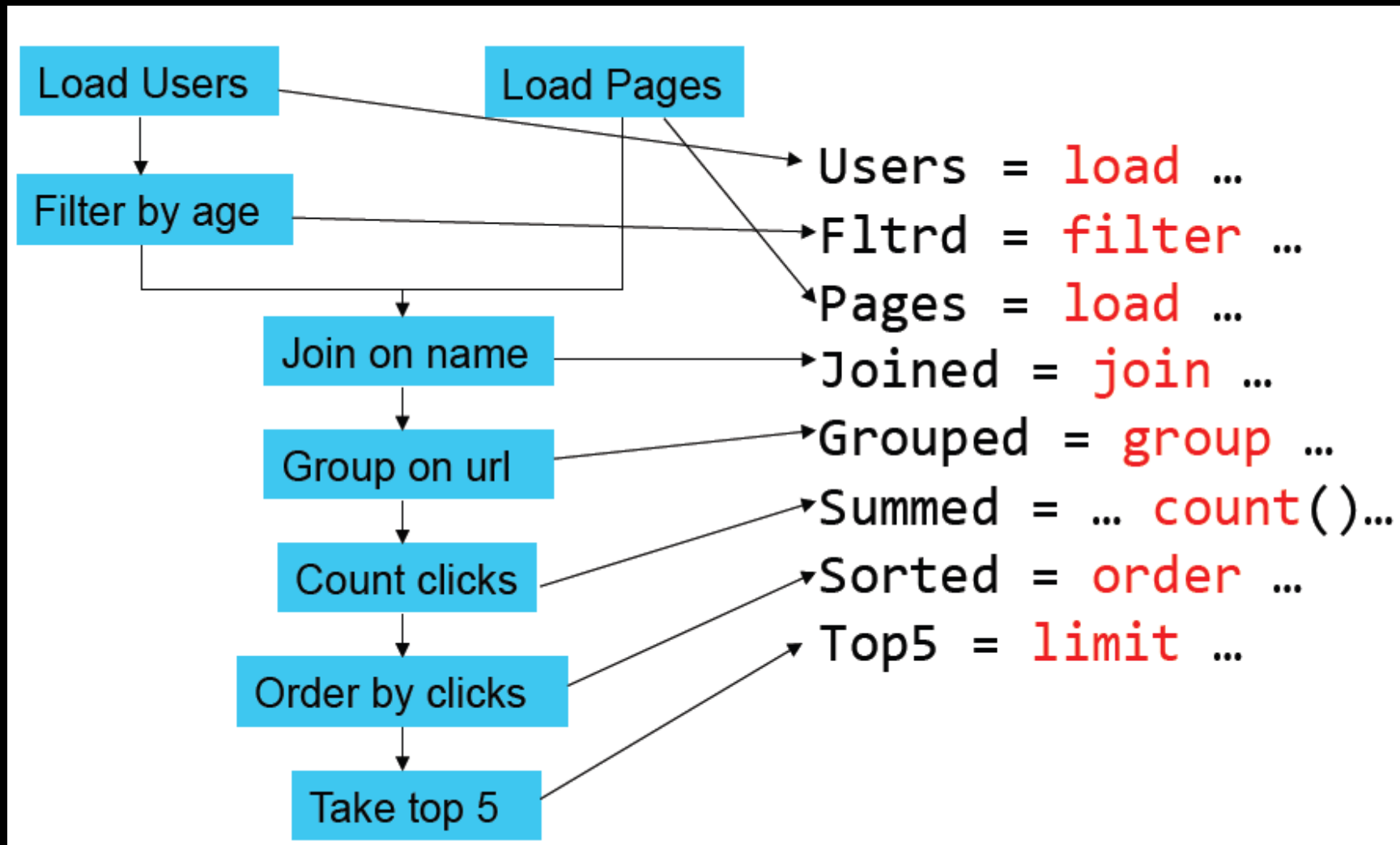
MapReduce Solution

```
public void map(LongWritable k, Text val,
        OutputCollector<Text, Text> oc,
        Reporter reporter) throws IOException {
    // Pull the key out
    String line = val.toString();
    int firstComma = line.indexOf(',');
    String value = line.substring(firstComma + 1);
    int age = Integer.parseInt(value);
    if (age < 18 || age > 25) return;
    String key = line.substring(0, firstComma);
    Text outKey = new Text(key);
    // Prepend an index to the value so we know which file
    // it came from.
    Text outVal = new Text("2" + value);
    oc.collect(outKey, outVal);
  }
}
```

Pig Latin Program

- Users = LOAD ‘users’ AS (name: chararray, age:int);
- Fltrd = FILTER Users BY
  age >= 18 AND age <= 25;
- Pages = LOAD ‘pages’ AS (user: chararray, url: chararray);
- Joined = JOIN Fltrd BY name, Pages BY user;
- Grouped = GROUP Joined BY url;
- Summed = FOREACH Grouped GENERATE group, COUNT(Joined) AS clicks;
- Sorted = ORDER Summed BY clicks DESC;
- Top5 = LIMIT Sorted 5;
- STORE Top5 INTO ‘top5sites’;

So ……..


Load Users
Load Pages
Filter by age
Join on name
Group on url
Count clicks
Order by clicks
Take top 5
Users = load …
Fltrd = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …

Pig Philosophy

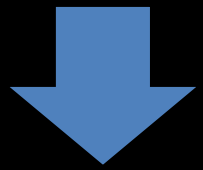
- Pigs Eat Anything
  - Can operate on all data fomats: relational, nested, or unstructured
- Pigs Live Anywhere
  - Not tied to one particular parallel framework
- Pigs Are Domestic Animals
  - Designed to be easily controlled and modified by its users.

Pig - Definition

- **Pig:** System for processing large semi structured data sets using Hadoop/ MapReduce platform
  - **Pig Latin**
    - High-level procedural language
  - **Pig Engine**
    - Parser, Optimizer and distributed query execution

Pig - Dataflow Language

- User specifies a sequence of steps
- Each step specifies only a single high-level data transformation



- Easier to keep track of variables
- Easier to understand where you are in the process of analyzing data
- High level primitives (group) → traditional database optimizations.

Pig Latin language: Simple Data Type

- *int : 42*
- *long : 42L / 42l*
- *float : 3.1415f*
- *double : 2.7182856*
- *chararray : UTF-8 String , ex:* hello world
- *bytearray : blob*

- For a complete list of data types:
  - http://pig.apache.org/docs/latest/basic.html#data-types

Pig Latin language: complex Data Type

- **Atom** : atomic data value
- **Tuple** : ordered set of fields
  - Field is a piece of data (of any data type)
- **Bag** : collection of tuples
- **Map** : set of key value pairs

Pig Latin language: Pig Latin statement

- Pig Latin statement: operator that takes a relation as input and produces another relation as output
- Relation: is a bag
  - Relations are referred to by name. Names are assigned by user as part of the Pig Latin statement
  - Exp:
    - A = filter B BY x>0;

Pig Latin Expression: LOAD

- A = **LOAD** ‘data.txt’ **USING** PigStorage() **AS** (ID:int , info:{t:(n1:int, n2:int)}, Provider: map[])

A ={ (1, {(2, 3), (4, 6)}, [‘yahoo’#’mail’]) }

Pig Latin Expressions: TOKENIZE

- Splits a string and outputs a bag of words

```
A = LOAD 'data' AS (f1:chararray);
DUMP A;
```

(Here is the first string.)
(Here is the second string.)
(Here is the third string.)

```
X = FOREACH A GENERATE TOKENIZE(f1);
DUMP X;
```

({(Here),(is),(the),(first),(string.)})
({(Here),(is),(the),(second),(string.)})
({(Here),(is),(the),(third),(string.)})

Pig Latin Expressions: Group

- groups together tuples that have the same group key

```
A = LOAD 'student' AS (name: chararray, age: int, gpa: float);
```

(John,18,4.0F)
(Mary,19,3.8F)
(Bill,20,3.9F)
(Joe,18,3.8F)

```
B = GROUP A BY age;
```

(18, {(John,18,4.0F), (Joe, 18,3.8F)})
(19,{(Mary,19,3.8F)})
(20,{(Bill,20,3.9F)})

```
C = FOREACH B GENERATE group, $1.name;
```

(18,{(John),(Joe)})
(19,{(Mary)})
(20,{(Bill)})

Pig Latin Expressions: CoGroup

- COGROUP is the same as GROUP
  - GROUP is usually used when only one relation is involved
  - COGROUP when multiple relations are involved. See GROUP for more information.

```
A = LOAD 'data1' AS (owner:chararray, pet:chararray);
B = LOAD 'data2' AS (friend1:chararray, friend2:chararray);
X = COGROUP A BY owner, B BY friend2;
```

COGROUP Vs. Join

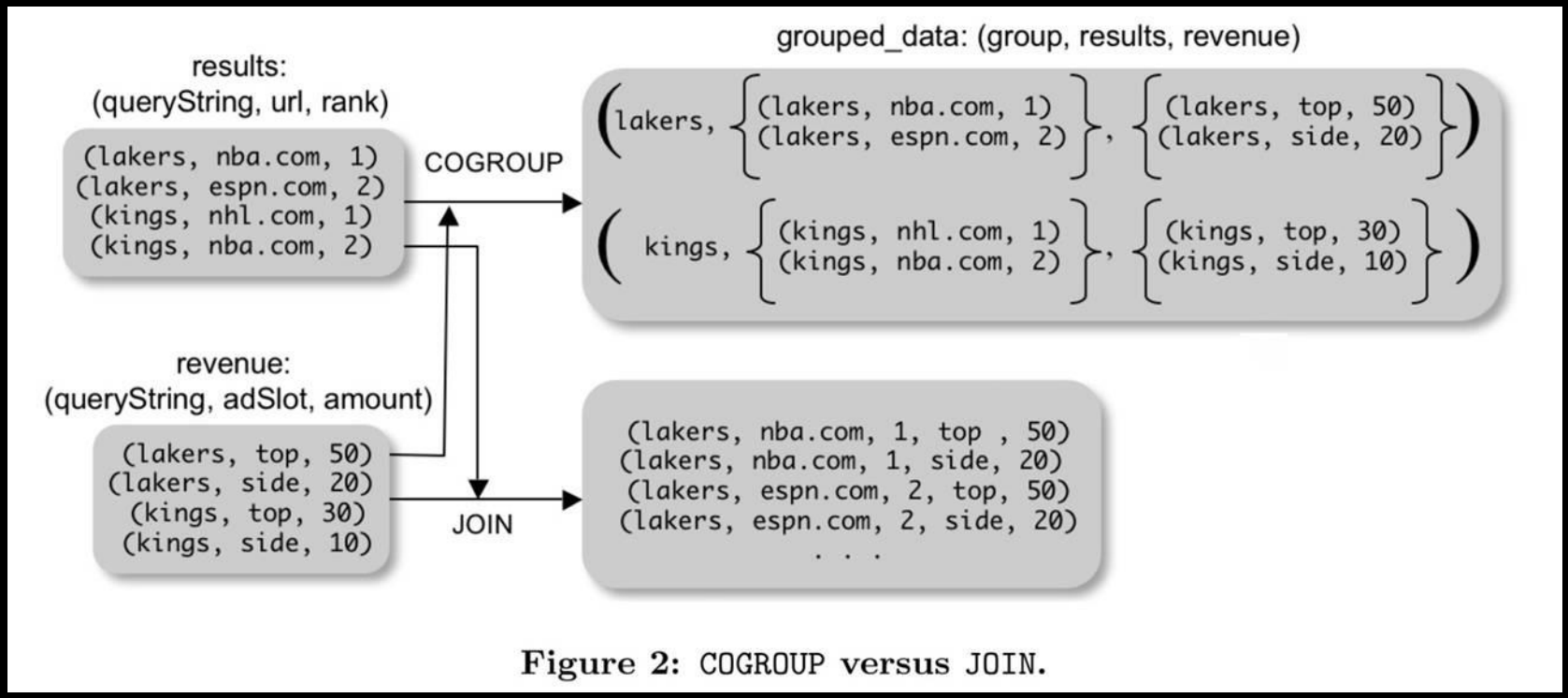


Figure 2: COGROUP versus JOIN.

Pig Latin Expressions: DISTINCT

- Removes duplicate tuples in a relation

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
```

(8,3,4)
(1,2,3)
(4,3,3)
(4,3,3)
(1,2,3)

```
X = DISTINCT A;
```

(1,2,3)
(4,3,3)
(8,3,4)

- DISTINCT does not preserve the original order of the contents

Pig Latin Expressions: Filter

- Selects tuples from a relation based on some condition

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
```

(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

```
X = FILTER A BY (a1== 8) OR (NOT (a2+a3 > a1));
```

(4,2,1)
(8,3,4)
(7,2,5)
(8,4,3)

Pig Latin Expressions: FOREACH

- Generates data transformations based on columns of data
- Projection: FOREACH A GENERATE f1, f2;

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
```

(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

2,{(1,2,3),(4,2,1),(7,2,5)}

3,{(8,3,4),(4,3,3)}

4,{(8,4,3)}

```
B = GROUP A BY $1;
C = FOREACH B {
                    D = ORDER A BY $2;
                    GENERATE D;}
```

{(4,2,1),(1,2,3),(7,2,5)}
{(4,3,3),(8,3,4)}
{(8,4,3)}

Pig Latin Expressions: Join

- Performs inner join of two or more relations based on common field values

```
A = LOAD 'data1' AS (a1:int,a2:int,a3:int);
B = LOAD 'data2' AS (b1:int,b2:int);
```

A
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

B
(2,4)
(8,9)
(1,3)
(2,7)
(2,9)
(4,6)
(4,9)

```
X = JOIN A BY a1, B BY b1;
```

(1,2,3,1,3)
(4,2,1,4,6)
(4,3,3,4,6)
(4,2,1,4,9)
(4,3,3,4,9)
(8,3,4,8,9)
(8,4,3,8,9)

Pig Latin Expressions: Split

- Partitions a relation into two or more relations

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

(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);
```

X	Y	Z
(1,2,3)	(4,5,6)	(1,2,3)
(4,5,6)		(7,8,9)

Pig Latin Expressions: Others

- **Store:** Stores data in the file system
  - STORE A INTO ‘myoutput’;
- **Union:** Computes the union of two or more relations
  - X = UNION A, B;
- **Limit:** Limits the number of output tuples
  - X = LIMIT A 3;
- **Eval Functions:** MAX , MIN, SUM, AVG, CONCAT, COUNT

- Execution of pig script starts only after output is requested

Diagnostic Operators: Describe

- Returns the schema of an alias

```
A = LOAD 'bag_data' AS
(B1:bag{T1:tuple(t1:int,t2:int)},B2:bag{T2:tuple(f1:int,f2:int)});

DESCRIBE A;
```

A: {B1: {T1: (t1: int,t2: int)},B2: {T2: (f1: int,f2: int)}}

Diagnostic Operators: Explain

- Review the logical, physical plans
- A= LOAD 'b.txt' using PigStorage(':') as (x:int,y:int);
  B= filter A by x>4;
  Explain B;

```
Logical Plan:
Store mohamad-Wed Dec 08 16:53:33 CET 2010-15 Schema: {x: int,y: int} Type: Unkn
own
|
|---Filter mohamad-Wed Dec 08 16:53:33 CET 2010-9 Schema: {x: int,y: int} Type:
bag
    |   |
    |   GreaterThan mohamad-Wed Dec 08 16:53:33 CET 2010-8 FieldSchema: boolean
Type: boolean
    |   |
    |   |---Project mohamad-Wed Dec 08 16:53:33 CET 2010-6 Projections: [0] Over
loaded: false FieldSchema: x: int Type: int
    |   |   Input: ForEach mohamad-Wed Dec 08 16:53:33 CET 2010-14
    |   |
    |   |---Const mohamad-Wed Dec 08 16:53:33 CET 2010-7 FieldSchema: int Type:
int
    |
    |---ForEach mohamad-Wed Dec 08 16:53:33 CET 2010-14 Schema: {x: int,y: int}
Type: bag
        |   |
        |   Cast mohamad-Wed Dec 08 16:53:33 CET 2010-11 FieldSchema: x: int Typ
e: int
        |   |
        |   |---Project mohamad-Wed Dec 08 16:53:33 CET 2010-10 Projections: [0]
 Overloaded: false FieldSchema: x: bytearray Type: bytearray
        |       Input: Load mohamad-Wed Dec 08 16:53:33 CET 2010-5
        |   |
        |   Cast mohamad-Wed Dec 08 16:53:33 CET 2010-13 FieldSchema: y: int Typ
e: int
        |   |
        |   |---Project mohamad-Wed Dec 08 16:53:33 CET 2010-12 Projections: [1]
 Overloaded: false FieldSchema: y: bytearray Type: bytearray
        |       Input: Load mohamad-Wed Dec 08 16:53:33 CET 2010-5
        |
        |---Load mohamad-Wed Dec 08 16:53:33 CET 2010-5 Schema: {x: bytearray,y:
 bytearray} Type: bag

-----------------------------------------------
```

Diagnostic Operators: Explain

/cygdrive/c/pig/pigtmp

```
-----------------------------------------------
Physical Plan:
-----------------------------------------------
Store(fakefile:org.apache.pig.builtin.PigStorage) - mohamad-Wed Dec 08 16:53:33
CET 2010-26
|
|---Filter[bag] - mohamad-Wed Dec 08 16:53:33 CET 2010-22
    |   |
    |   Greater Than[boolean] - mohamad-Wed Dec 08 16:53:33 CET 2010-25
    |   |
    |   |---Project[int][0] - mohamad-Wed Dec 08 16:53:33 CET 2010-23
    |   |
    |   |---Constant(4) - mohamad-Wed Dec 08 16:53:33 CET 2010-24
    |
    |---New For Each(false,false)[bag] - mohamad-Wed Dec 08 16:53:33 CET 2010-21
        |   |
        |   Cast[int] - mohamad-Wed Dec 08 16:53:33 CET 2010-18
        |   |
        |   |---Project[bytearray][0] - mohamad-Wed Dec 08 16:53:33 CET 2010-17
        |   |
        |   Cast[int] - mohamad-Wed Dec 08 16:53:33 CET 2010-20
        |   |
        |   |---Project[bytearray][1] - mohamad-Wed Dec 08 16:53:33 CET 2010-19
        |
        |---Load(b.txt:PigStorage(':')) - mohamad-Wed Dec 08 16:53:33 CET 2010-1
$
```

Diagnostic Operators: Illustrate

- Displays a step-by-step execution of a sequence of statements
  - The data load statement must include a schema
  - The relation used with the ILLUSTRATE command cannot include the map data type, the LIMIT and SPLIT operators, or nested FOREACH statements
- Selects an appropriate and concise set of example data automatically
  - test your programs on small datasets and get faster turnaround times

Diagnostic Operators: Illustrate

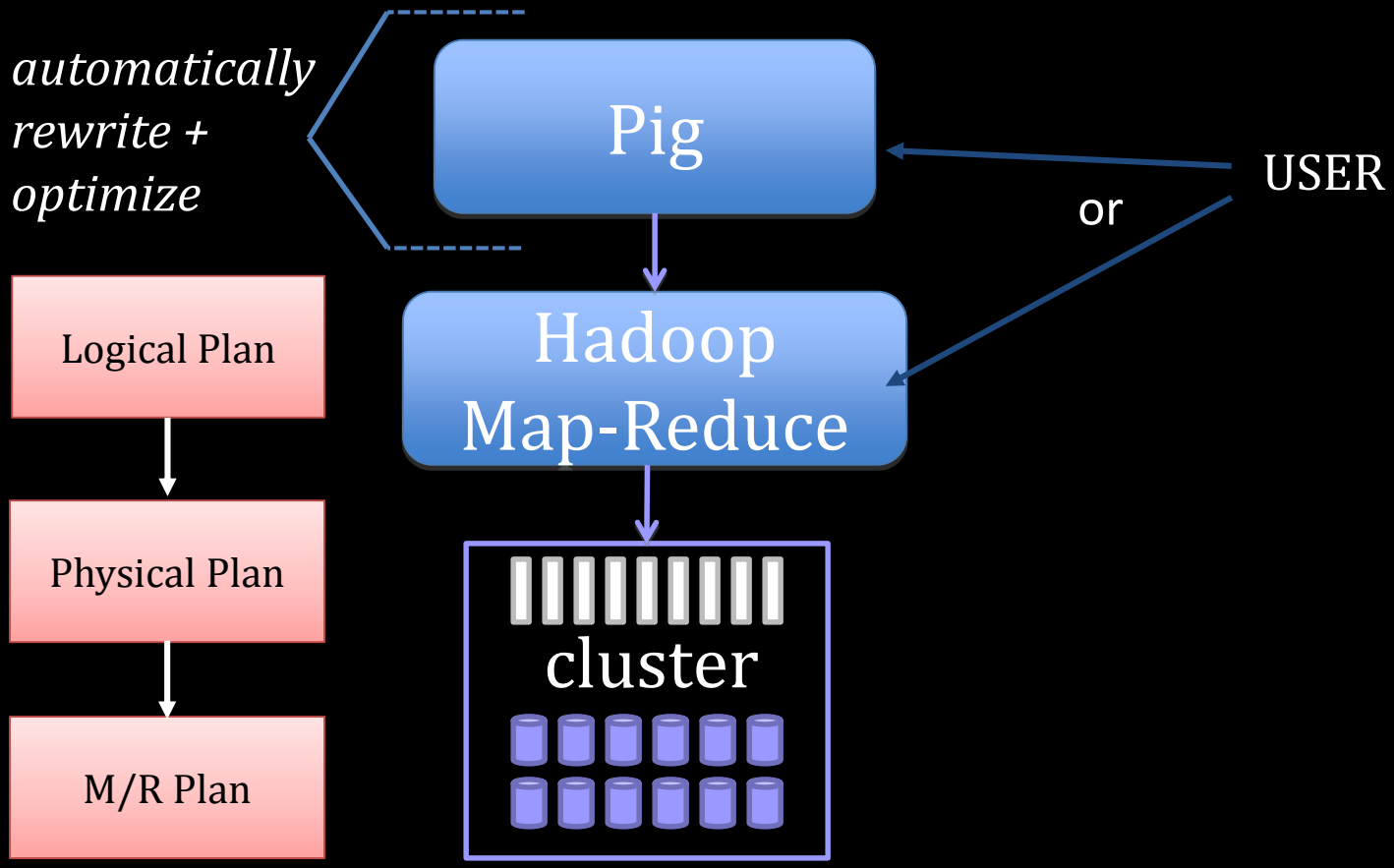
```
A= LOAD 'b.txt' using PigStorage(':') as (x:int,y:int);
B= filter A by x>4;
ILLUSTRATE B;
Store B into ‘bresult.txt’;
```

/cygdrive/c/pig/pigtmp

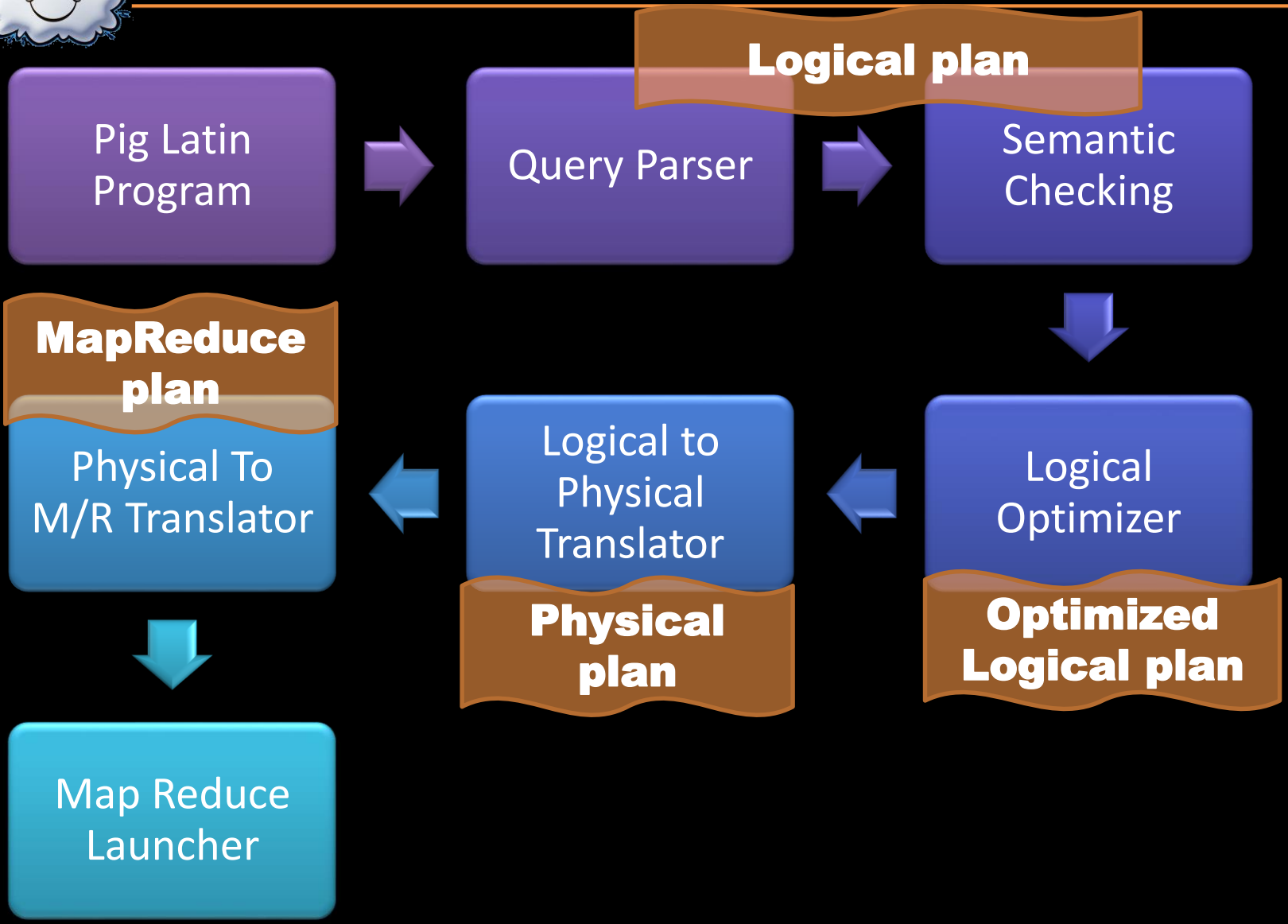
```
mohamad@pclimos17 /cygdrive/c/pig/pigtmp
$ java -cp ./pig.jar org.apache.pig.Main -x local testExplain.pig
-------------------------------------
| A     | x: bytearray | y: bytearray |
-------------------------------------
|       | 1            | 5            |
|       | 8            | 6            |
-------------------------------------
-------------------------
| A     | x: int | y: int |
-------------------------
|       | 1      | 5      |
|       | 8      | 6      |
-------------------------
-------------------------
| B     | x: int | y: int |
-------------------------
|       | 8      | 6      |
-------------------------

2010-12-08 17:15:24,860 [main] INFO  org.apache.pig.backend.local.executionengin
e.LocalPigLauncher - 100% complete!
2010-12-08 17:15:24,860 [main] INFO  org.apache.pig.backend.local.executionengin
e.LocalPigLauncher - Success!!
```

Implementation



Implementation



MapReduce Plan

- Embeds each physical operator inside a Map-Reduce stage to arrive at a Map-Reduce plan

- Boundaries for M/R include group/cogroup, distinct, cross, order by

Pig/Pig Latin - Conclusion

- Pig Latin:
  - ✓ It is a sweet spot between Map-Reduce and SQL
  - ✓ Provides common data processing operations
  - ✓ Easy to plug-in user code
  - ✗ No metadata
  - ✗ No Indexing
  - ✗ Philosophy: Optimize it yourself

References

- [1] *MapReduce: Simplified Data Processing on Large Clusters*; Jeffrey Dean ,Sanjay Ghemawat, 2004
- [2] http://www.searchworkings.org/blog/-/blogs/introduction-to-hadoop/
- [3] *Pig Latin: A Not-So-Foreign Language for Data Processing*; Olston C. , Reed B., Srivastava U. ,Kumar R. , Tomkins A. ; SIGMOD, 2008
- [4] http://pig.apache.org/

Questions