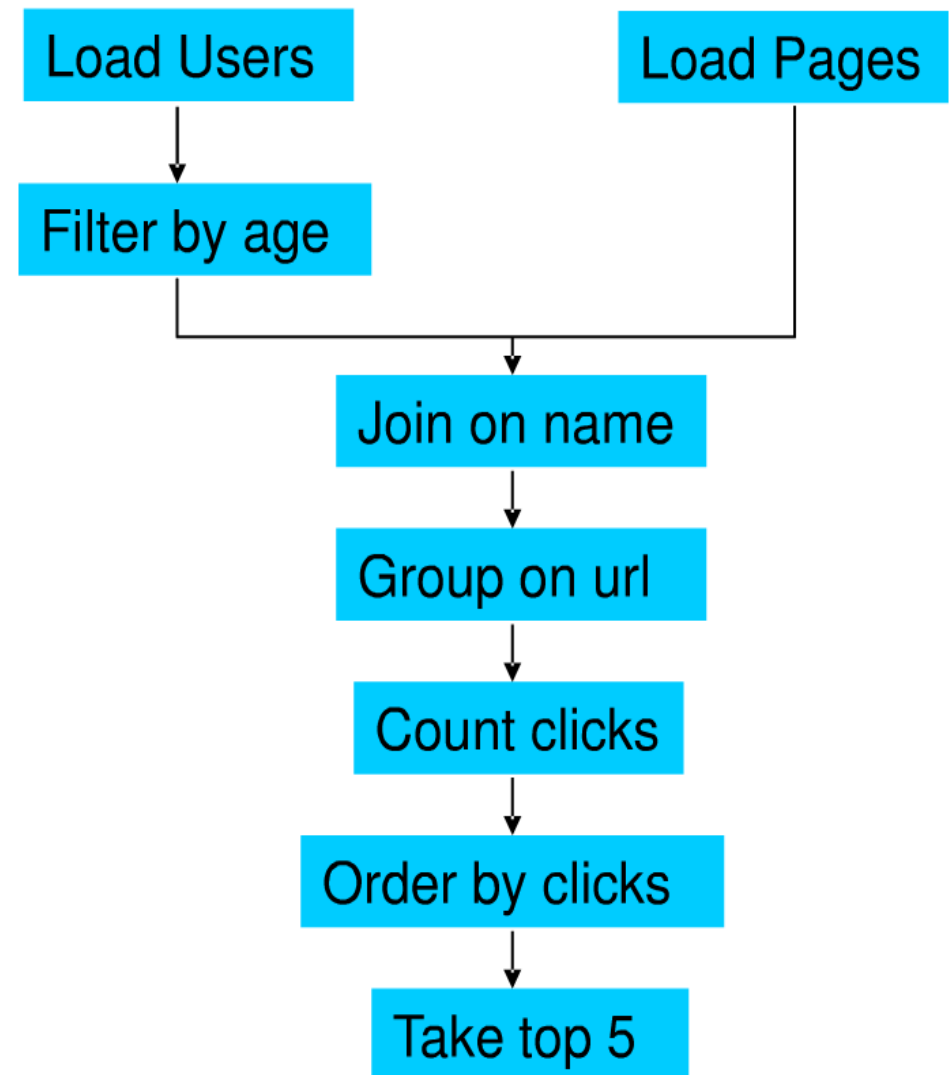

Pig, a high level data
processing system on
Hadoop

Pig to the Rescue

- High level dataflow language (Pig Latin)
 - Much simpler than Java
 - Simplifies the data processing
- Puts the operations at the appropriate phases
- Chains multiple MR jobs

Motivation by Example

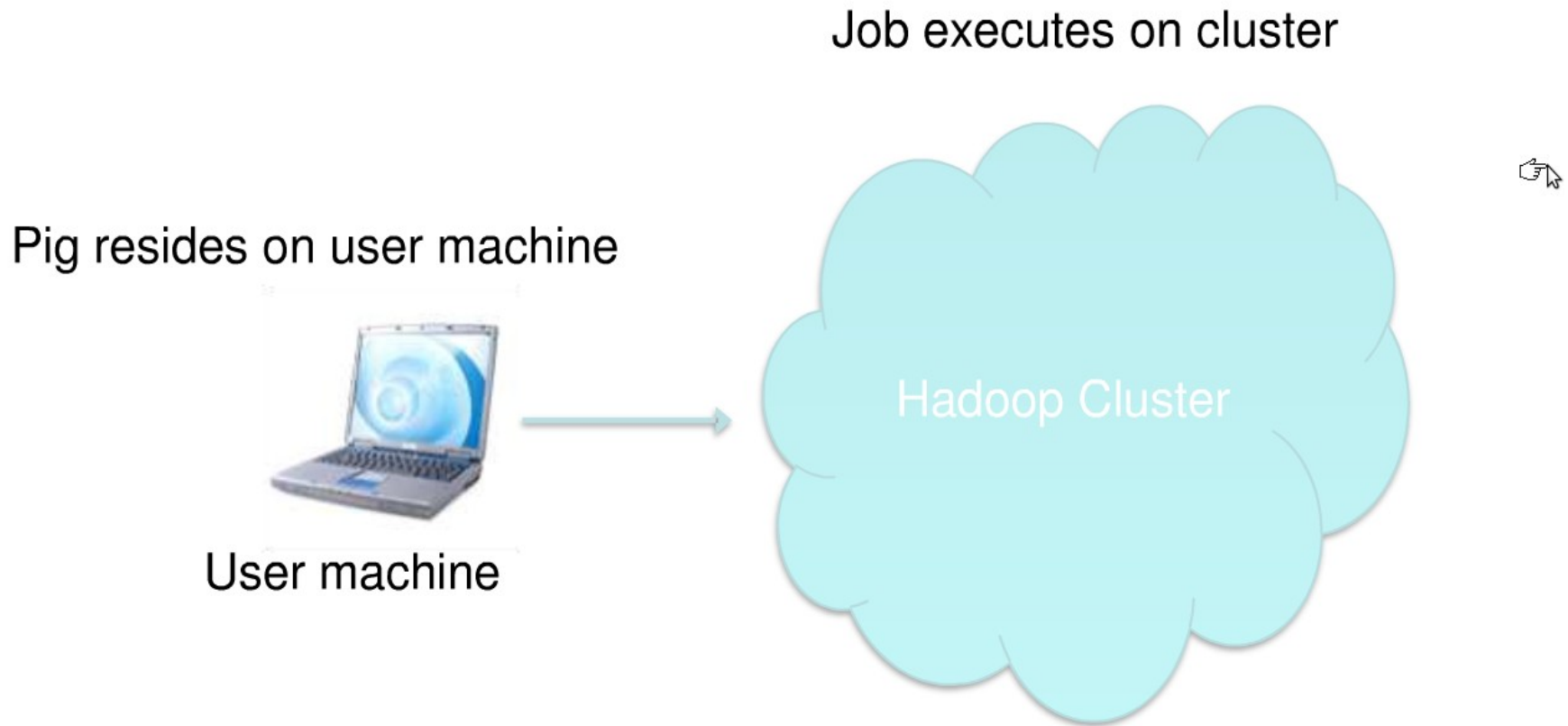
- Suppose we have user data in one file, website data in another file.
- We need to find the top 5 most visited pages by users aged 18-25



In Pig Latin

```
Users = load 'users' as (name, age);
Fltrd = filter Users by
    age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = joinFltrdby name, Pages by user;
Grpd = groupJndbyurl;
Smmd = foreachGrpdgenerate group,
COUNT(Jnd) as clicks;
Srted = orderSmmdby clicks desc;
Top5 = limitSrted 5;
store Top5 into 'top5sites';
```

Pig runs over Hadoop



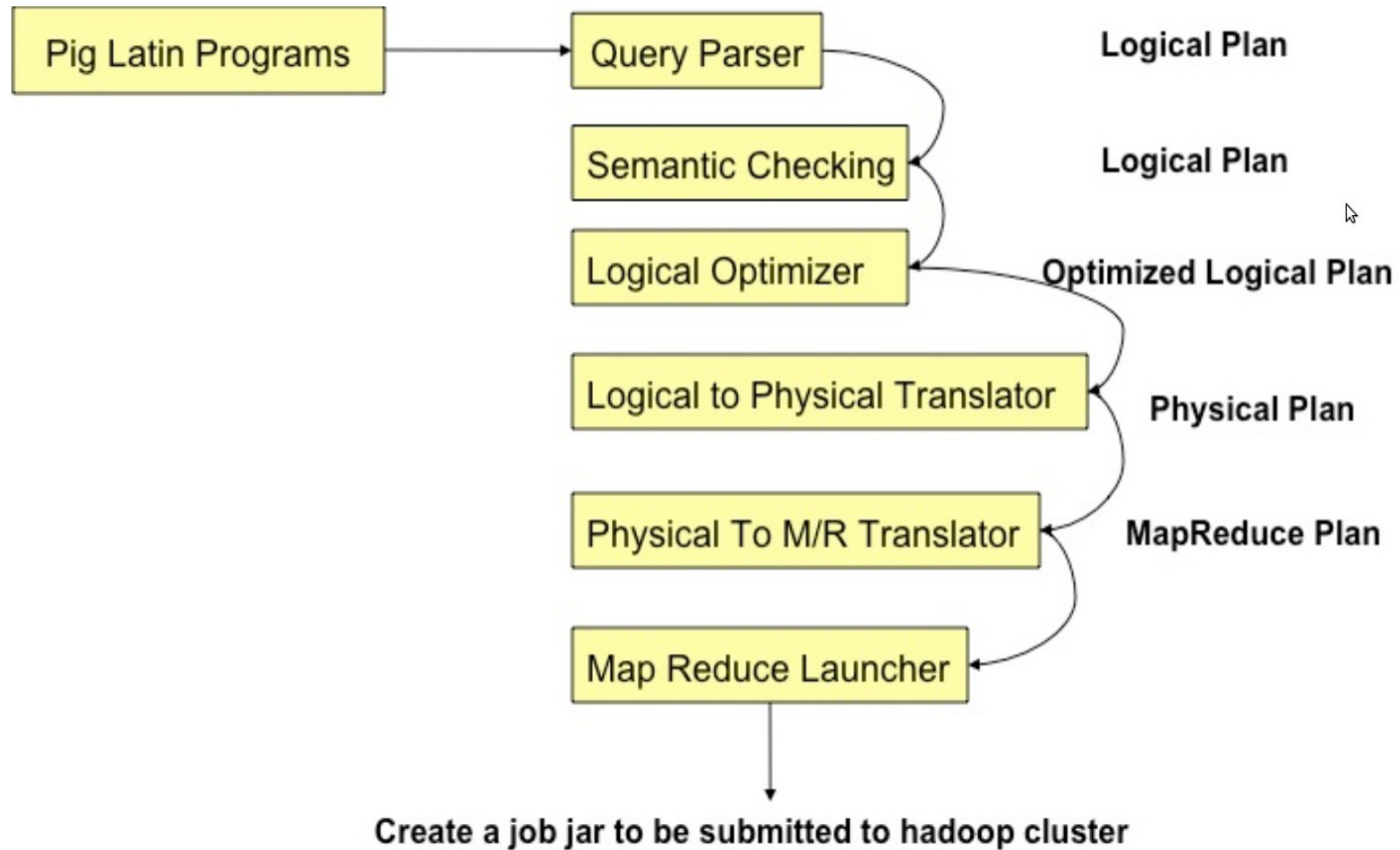
No need to install anything extra on your Hadoop cluster.

View of Pig from inside

Pig Execution Modes

- Local mode
 - ❑ Launch single JVM
 - ❑ Access local file system
 - ❑ No MR job running
- Hadoop mode
 - ❑ Execute a sequence of MR jobs
 - ❑ Pig interacts with Hadoop master node

Compilation

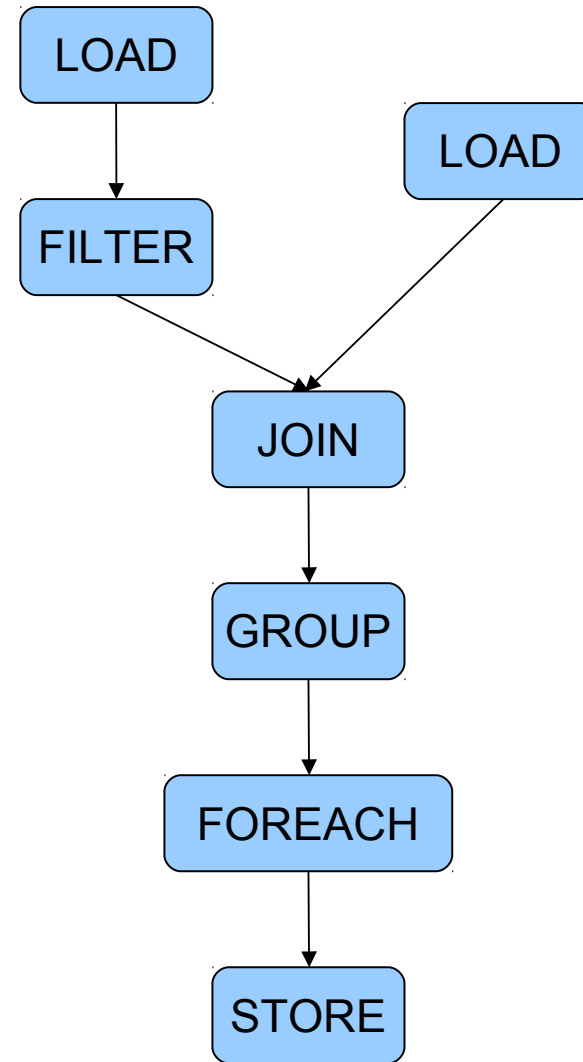


Parsing

- Type checking with schema
- Reference verification
- Logical plan generation
 - One-to-one fashion
 - Independent of execution platform
 - Limited optimization
 - No execution until DUMP or STORE

Logical 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';
```



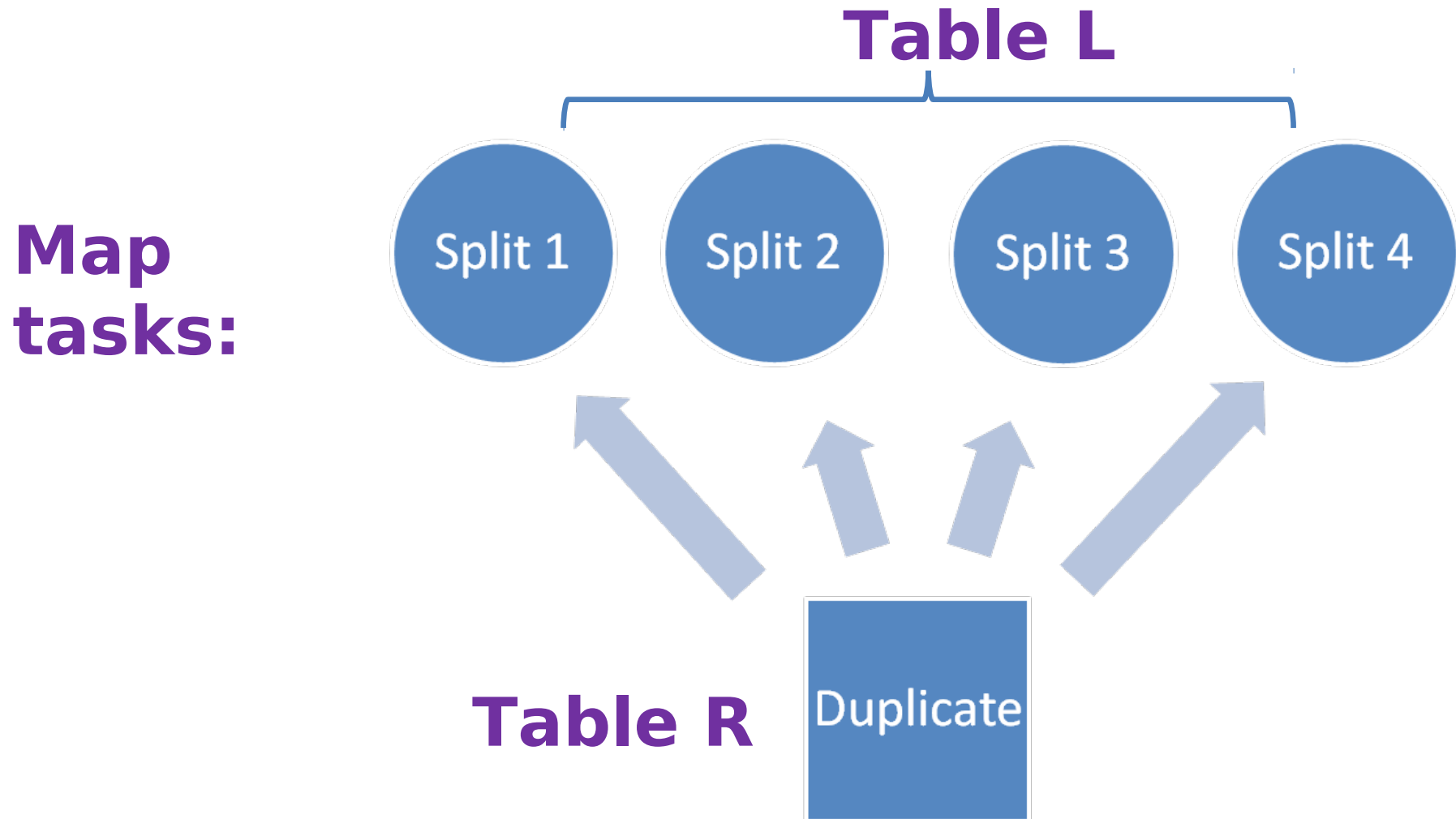
Physical Plan

- 1:1 correspondence with most logical operators
- Except for:
 - DISTINCT
 - (CO)GROUP
 - JOIN
 - ORDER

Joins in MapReduce

- Two typical types of join
 - Map-side with Index (akin to hash-join)
 - Map-side join parallel (akin to sort-join)
 - Reduce-side join (akin to sort-merge-join)

Map-side with Index



Two Map Reduce with Index

Table L

Map tasks:

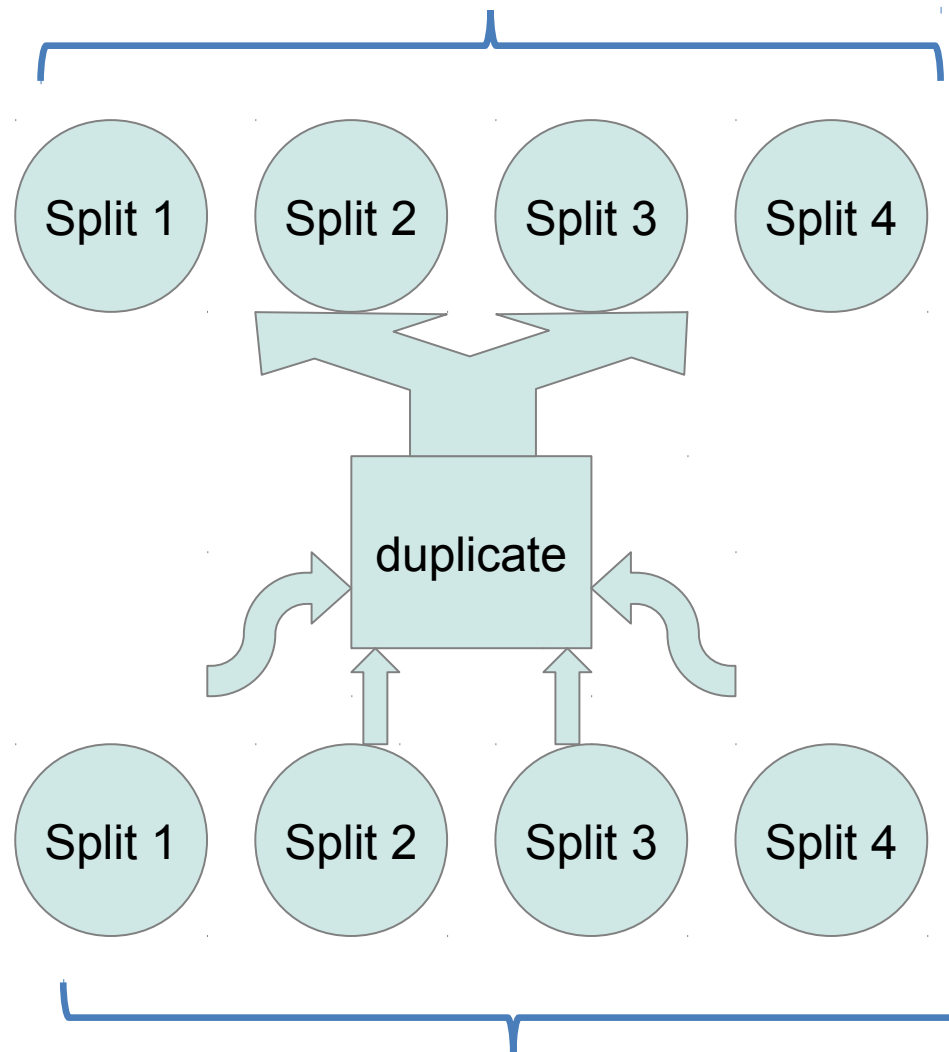


Table R

Map-side Join

Table L (sorted)

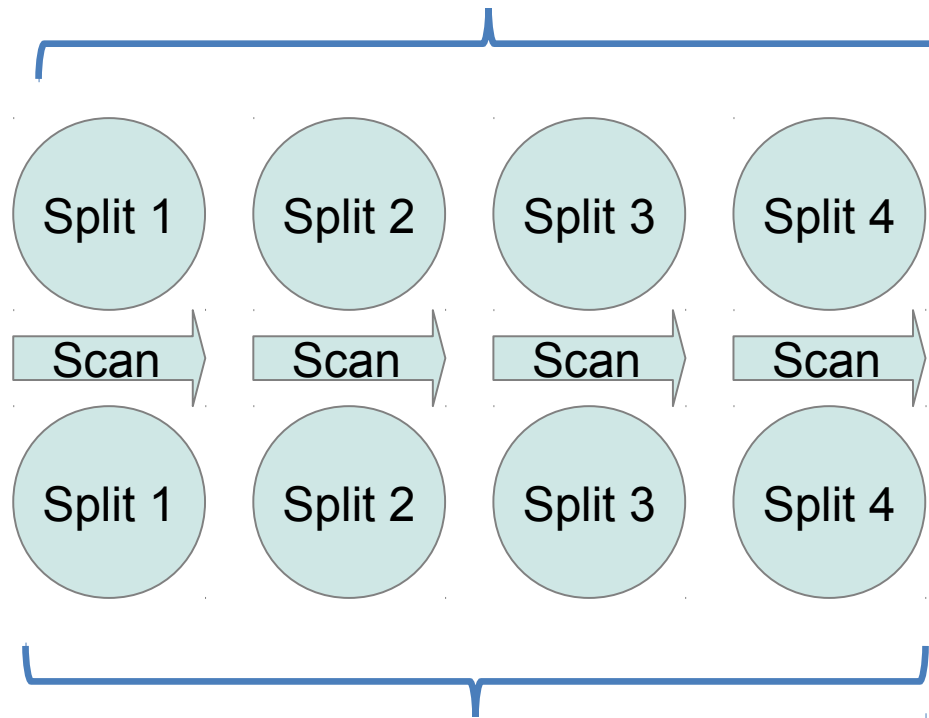
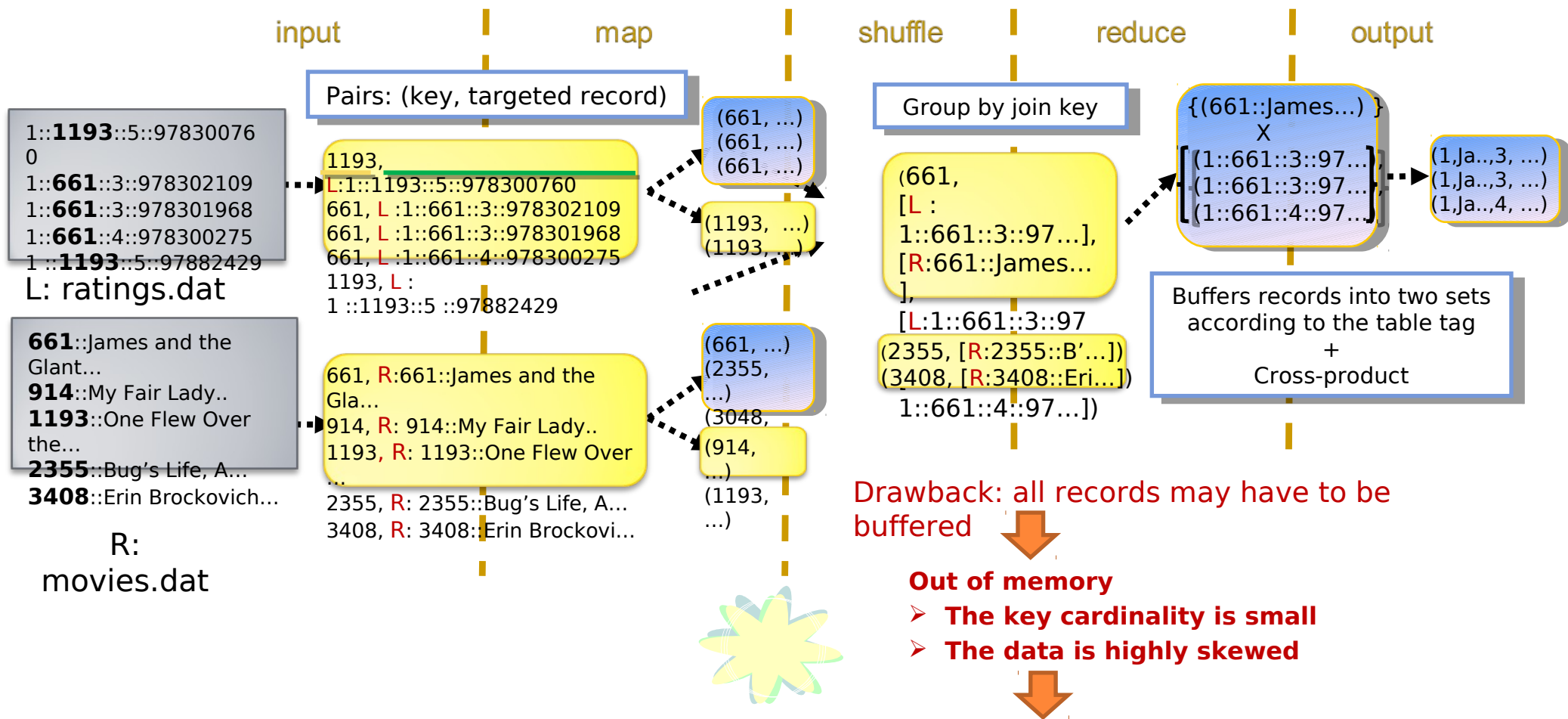


Table R (sorted)

**Map
tasks:**

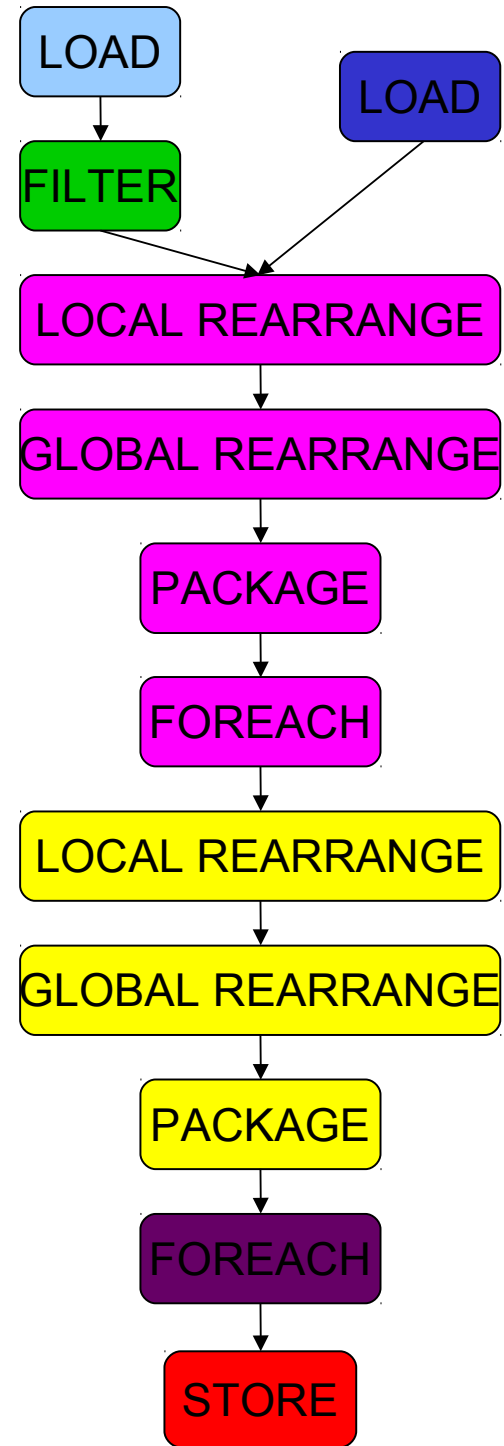
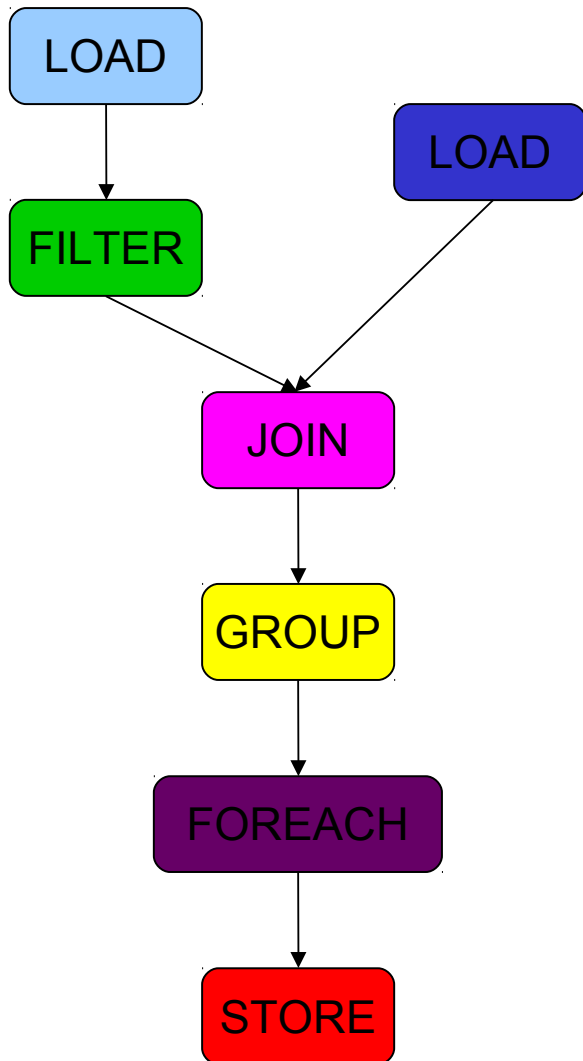
REDUCE-SIDE JOIN



Phase /Function	Improvement
Map Function	Output key is changed to a composite of the join key and the table tag.
Partitioning function	Hashcode is computed from just the join key part of the composite key
Grouping function	Records are grouped on just the join key

Physical Plan

- 1:1 correspondence with most logical operators
- Except for:
 - DISTINCT
 - (CO)GROUP
 - JOIN
 - ORDER

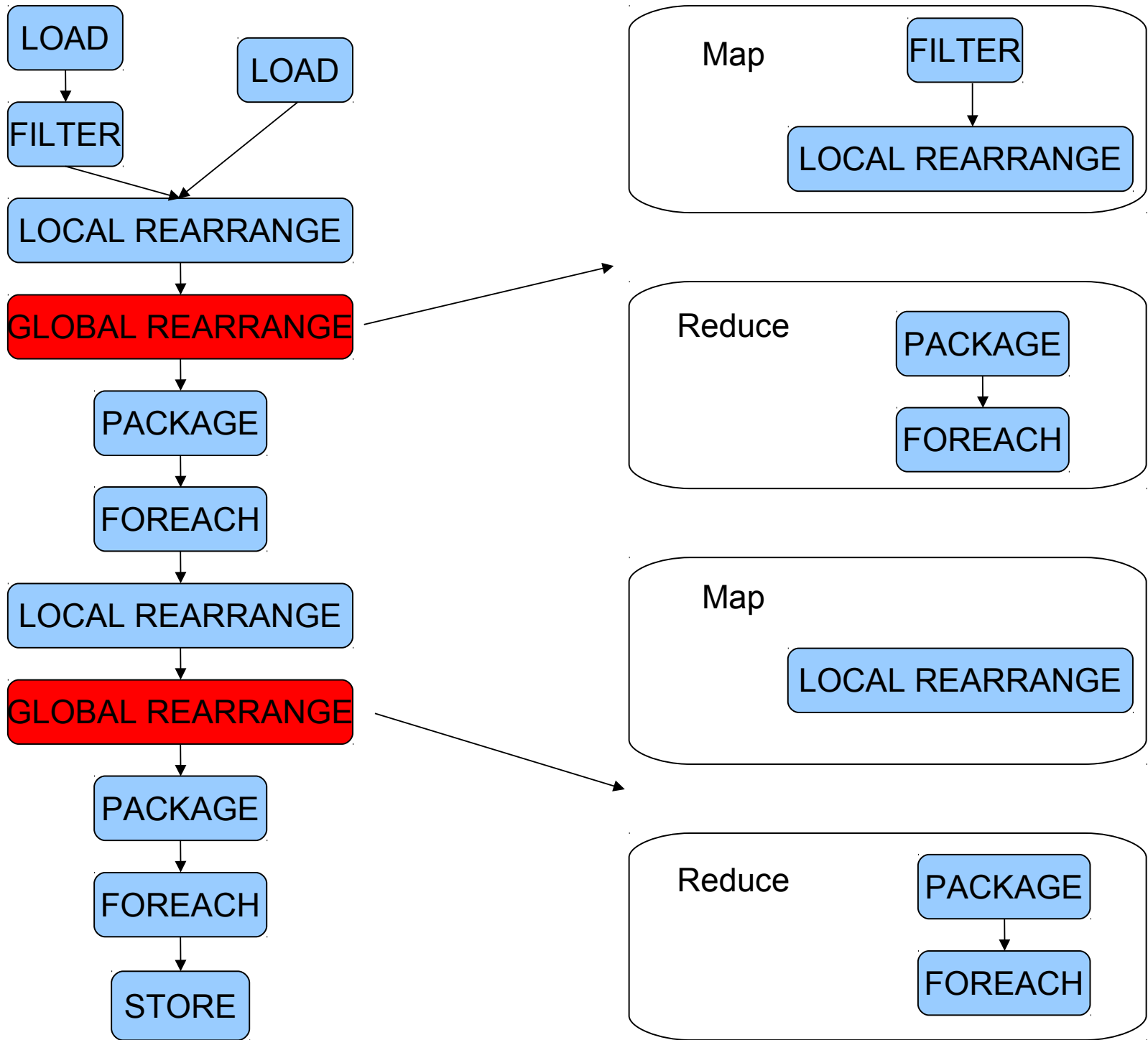


Physical Optimizations

- Always use combiner for pre-aggregation
- Insert SPLIT to re-use intermediate result
- Early projection (logical or physical?)

MapReduce Plan

- Determine MapReduce boundaries
 - GLOBAL REARRANGE
 - STORE/LOAD
- Some operations are done by MapReduce framework
- Coalesce other operators into Map & Reduce stages
- Generate job jar file



Execution in Hadoop Mode

- The MR jobs not dependent on anything in the MR plan will be submitted for execution
- MR jobs will be removed from MR plan after completion
 - Jobs whose dependencies are satisfied are now ready for execution
- Currently, no support for inter-job fault-tolerance

Discussion of the Two Readings on Pig (SIGMOD 2008 and VLDB 2009)

Discussion Points for Reading 1

- Examples of the nested data model, CoGroup, and Join (Figure 2)
- Nested query in Section 3.7

What are the Logical, Physical, and MapReduce

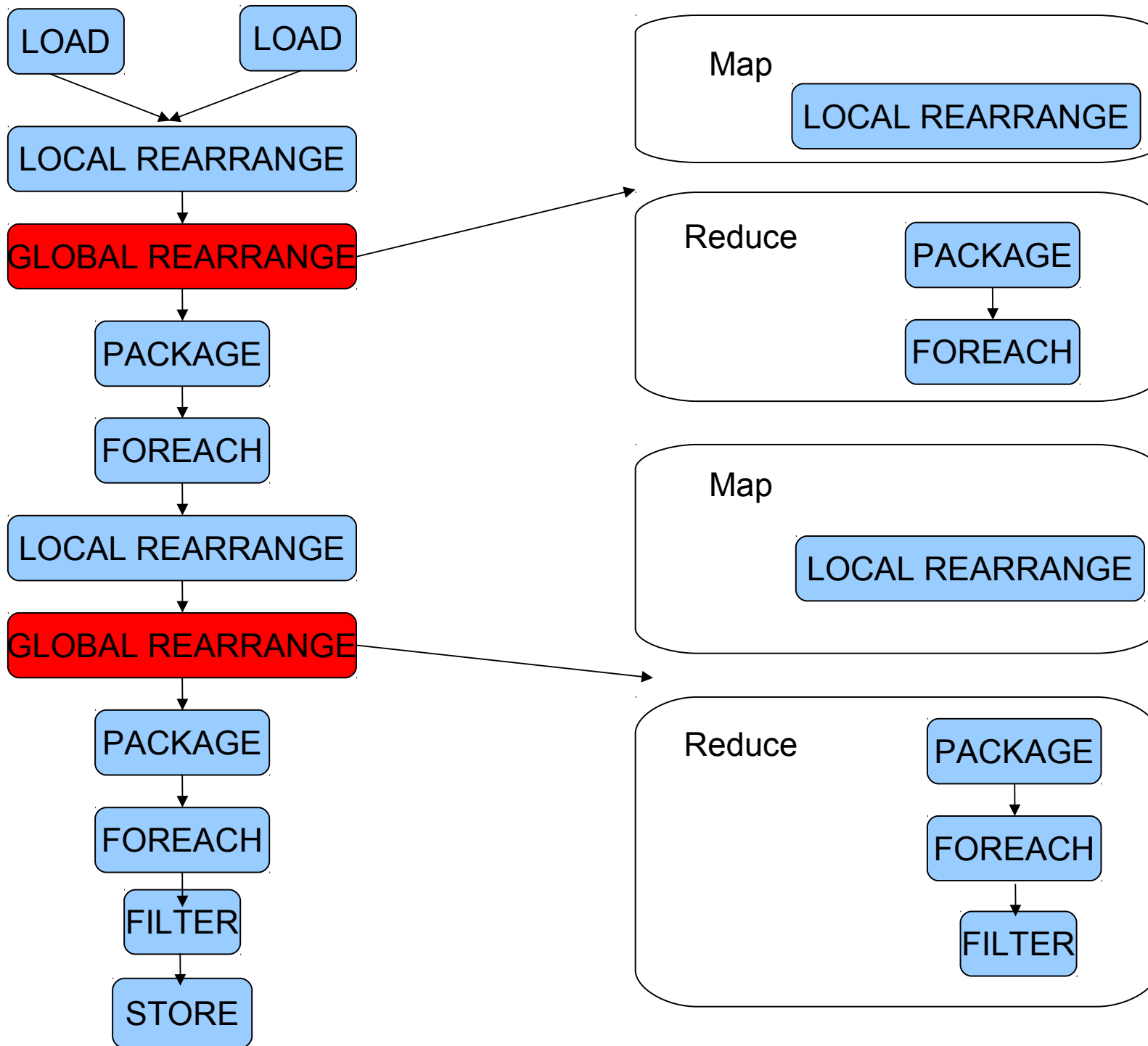
Operators

LOAD GROUP COGROUP FILTER FOREACH ORDER

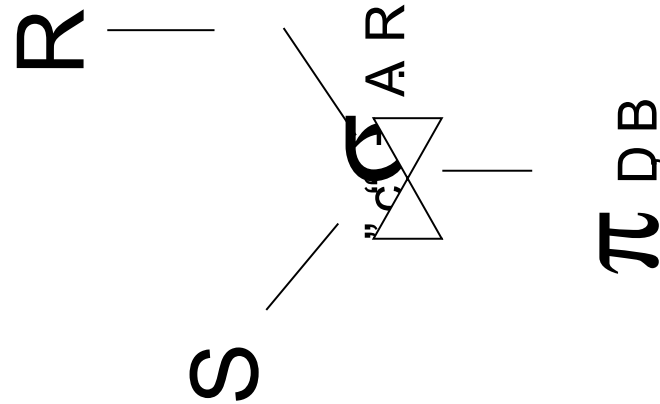
= LOAD USING AS ()

[Generate Query](#)

<pre>visits = LOAD 'visits.txt' AS (user, url, time); pages = LOAD 'pages.txt' AS (url, pagerank); v_p = JOIN visits BY url, pages BY url; users = GROUP v_p BY user; useravg = FOREACH users GENERATE group, AVG(v_p.pagerank) AS avgpr; answer = FILTER useravg BY avgpr > '0.5'; STORE answer INTO '/user/alan/answer';</pre>	<pre>visits: (Amy, cnn.com, 8am) (Amy, frogs.com, 9am) (Fred, snails.com, 11am) pages: (cnn.com, 0.8) (frogs.com, 0.8) (snails.com, 0.3) v_p: (Amy, cnn.com, 8am, cnn.com, 0.8) (Amy, frogs.com, 9am, frogs.com, 0.8) (Fred, snails.com, 11am, snails.com, 0.3) users: (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8), (Amy, frogs.com, 9am, frogs.com, 0.8) }) (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) }) useravg: (Amy, 0.8) (Fred, 0.3) answer: (Amy, 0.8)</pre>
--	---

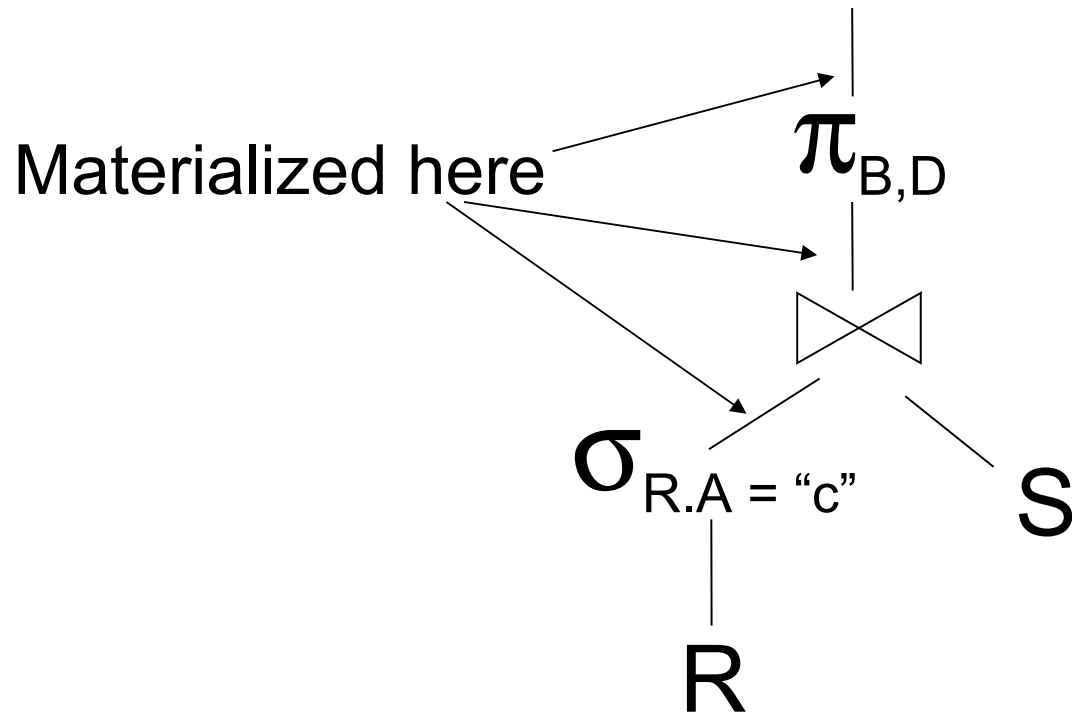


Recall Operator Plumbing

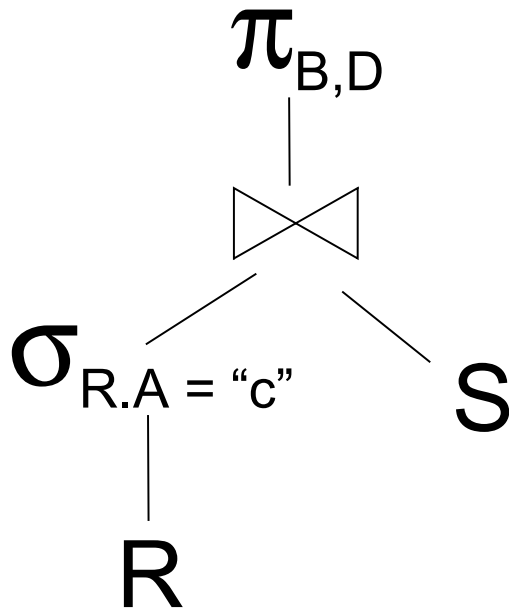


- **Materialization:** output of one operator written to disk, next operator reads from the disk
- **Pipelining:** output of one operator directly fed to next operator

Materialization



Iterators: Pipelining



→ Each operator supports:

- `Open()`
- `GetNext()`
- `Close()`

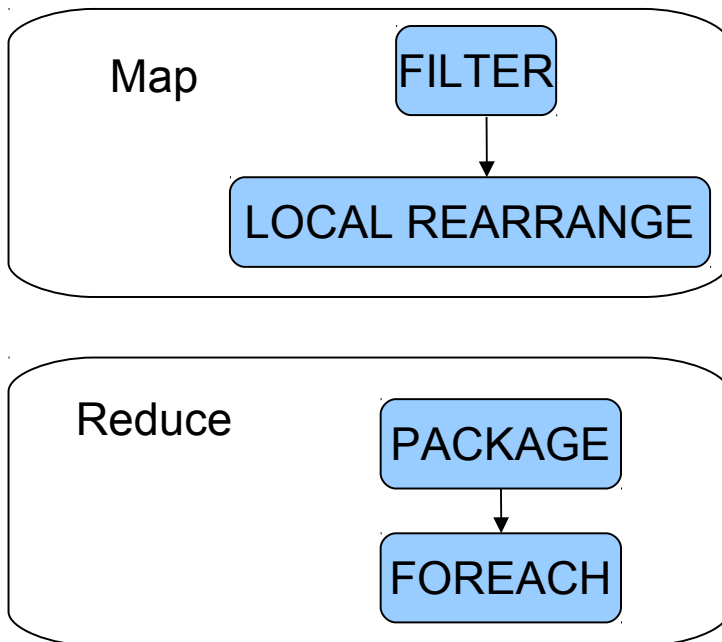
How do these operators

exe 1950 0 1 | Pig?

```
1950 22 1
1950 e 1
1949 111 1
1949 78 1
```

Hints (based on Reading 2):

- What will Hadoop's map function and reduce function calls do in this case?
- How does each operator work? What does each operator do? (Section 4.3)
- Outermost operator graph (Section 5)
- Iterator model (Section 5)



Branching Flows in Pig

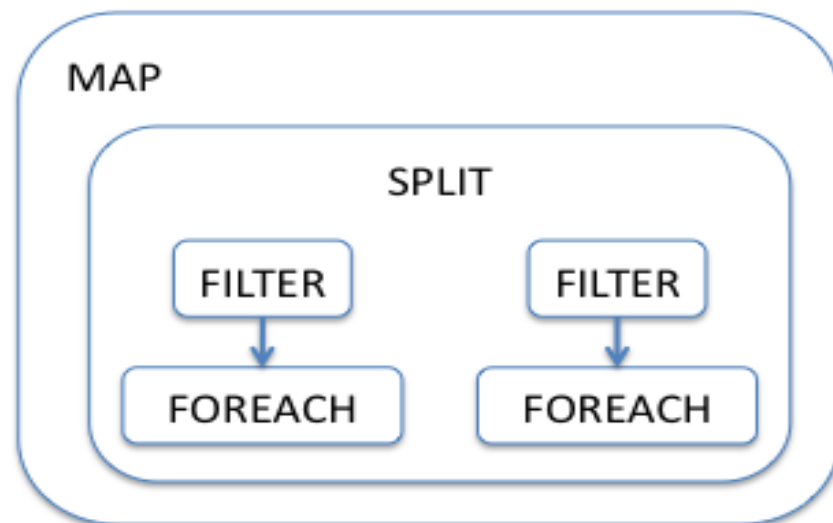
```
clicks = LOAD `clicks'  
AS (userid, pageid, linkid, viewedat);
```

```
SPLIT clicks INTO  
pages IF pageid IS NOT NULL,  
links IF linkid IS NOT NULL;
```

```
cpages = FOREACH pages GENERATE  
userid,  
CanonicalizePage(pageid) AS cpage,  
viewedat;
```

```
clinks = FOREACH links GENERATE userid,  
CanonicalizeLink(linkid) AS clink,  
viewedat;
```

```
STORE cpages INTO `pages`;  
STORE clinks INTO `links`;
```



- Hints (based on Reading 2, Section 5.1, last two paras before Section 5.1.1):
 - Outermost data flow graph
 - New pause signal for iterators

Branching Flows in Pig

- Draw the MapReduce plan for this query

```
clicks = LOAD `clicks`  
AS (userid, pageid, linkid, viewedat);  
  
byuser = GROUP clicks BY userid;  
  
result = FOREACH byuser {  
  
    uniqPages = DISTINCT clicks.pageid;  
  
    uniqLinks = DISTINCT clicks.linkid;  
  
    GENERATE group, COUNT(uniqPages),  
COUNT(uniqLinks);  
  
};
```


Branching Flows in Pig

- Draw the MapReduce plan for this query

```
clicks = LOAD `clicks`
AS (userid, pageid, linkid, viewedat);

byuser = GROUP clicks BY userid;

result = FOREACH byuser {

    fltrd = FILTER clicks BY viewedat IS NOT
NULL;

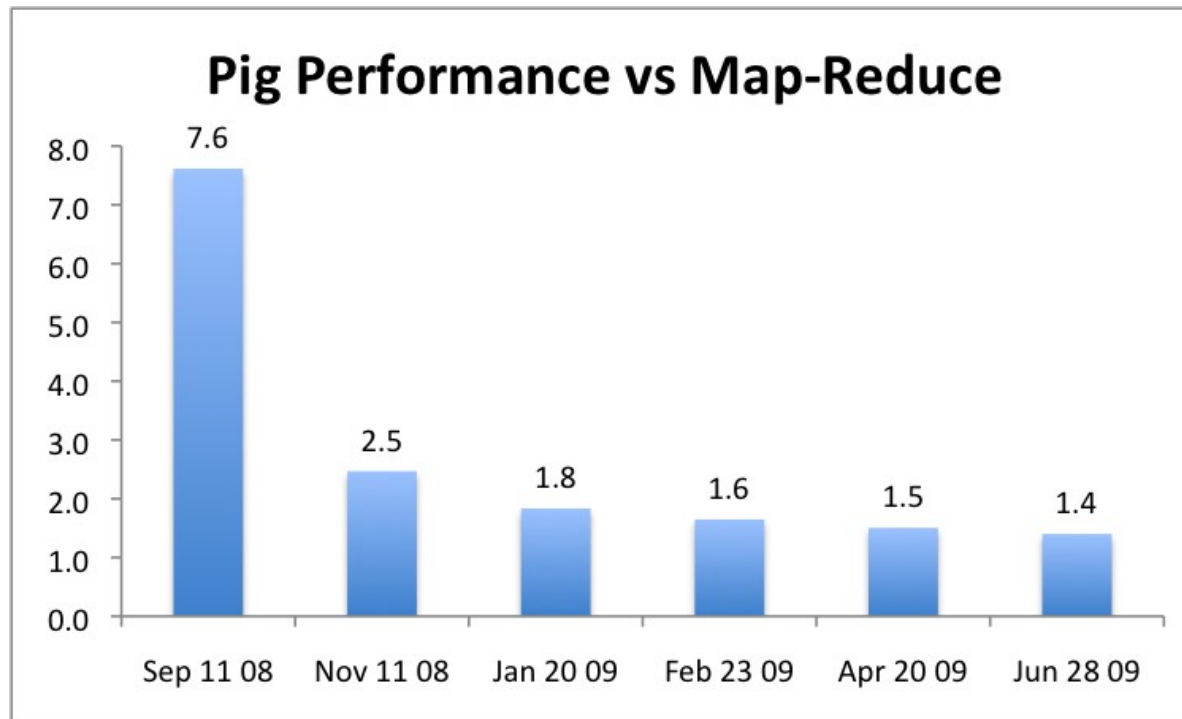
    uniqPages = DISTINCT fltrd.pageid;

    uniqLinks = DISTINCT fltrd.linkid;

    GENERATE group, COUNT(uniqPages),
COUNT(uniqLinks);
};
```

Performance and future improvement

Pig Performance



Images from
<http://wiki.apache.org/pig/PigTalksPapers>

Future Improvements

- Query optimization
 - Currently rule-based optimizer for plan rearrangement and join selection
 - Cost-based in the future
- Non-Java UDFs
- Grouping and joining on pre-partitioned/sorted data
 - Avoid data shuffling for grouping and joining
 - Building metadata facilities to keep track of data layout
- Skew handling
 - For load balancing

- Get more information at the Pig website
- You can work with the source code to implement something new in Pig
- Also take a look at Hive, a similar system from Facebook

References

- Some of the content come from the following presentations:
 - Introduction to data processing using Hadoop and Pig, by Ricardo Varela
 - Pig, Making Hadoop Easy, by Alan F. Gates
 - Large-scale social media analysis with Hadoop, by Jake Hofman
 - Getting Started on Hadoop, by Paco Nathan
 - MapReduce Online, by Tyson Condie and Neil Conway