Section 8

Pig Latin

Outline

- Based on *Pig Latin: A not-so-foreign language for data processing*, by Olston, Reed, Srivastava, Kumar, and Tomkins, 2008

Pig Engine Overview

- Data model = loosely typed *nested relations*
- Query model = a sql-like, dataflow language

- Execution model:
  - Option 1: run locally on your machine
  - Option 2: compile into sequence of map/reduce, run on a cluster supporting Hadoop

- Main idea: use Opt1 to debug, Opt2 to execute

Example

- Input: a table of urls:
  (url, category, pagerank)
- Compute the average pagerank of all sufficiently high pageranks, for each category
- Return the answers only for categories with sufficiently many such pages

First in SQL…

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

…then in 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)
```

Pig Latin combines
- high-level declarative querying in the spirit of SQL, and
- low-level, procedural programming a la map-reduce.

Types in Pig-Latin

- Atomic: string or number, e.g. ‘Alice’ or 55
- Tuple: (‘Alice’, 55, ‘salesperson’)
- Bag: {(‘Alice’, 55, ‘salesperson’),
  (‘Betty’,44, ‘manager’), …}
- Maps: we will try not to use these

Types in Pig-Latin

Bags can be nested !

- {(‘a’, {1,4,3}), (‘c’,{ }), (‘d’, {2,2,5,3,2})}

Tuple components can be referenced by number

- \$0, \$1, \$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

Loading data

- Input data = FILES !
  - Heard that before ?

- The LOAD command parses an input file into a bag of records

- Both parser (="deserializer") and output type are provided by user

Loading data

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

Loading data

- USING userfuction( ) -- is optional
  - Default deserializer expects tab-delimited file
- AS type – is optional
  - Default is a record with unnamed fields; refer to them as $0, $1, …
- The return value of LOAD is just a handle to a bag
  - The actual reading is done in pull mode, or parallelized

FOREACH

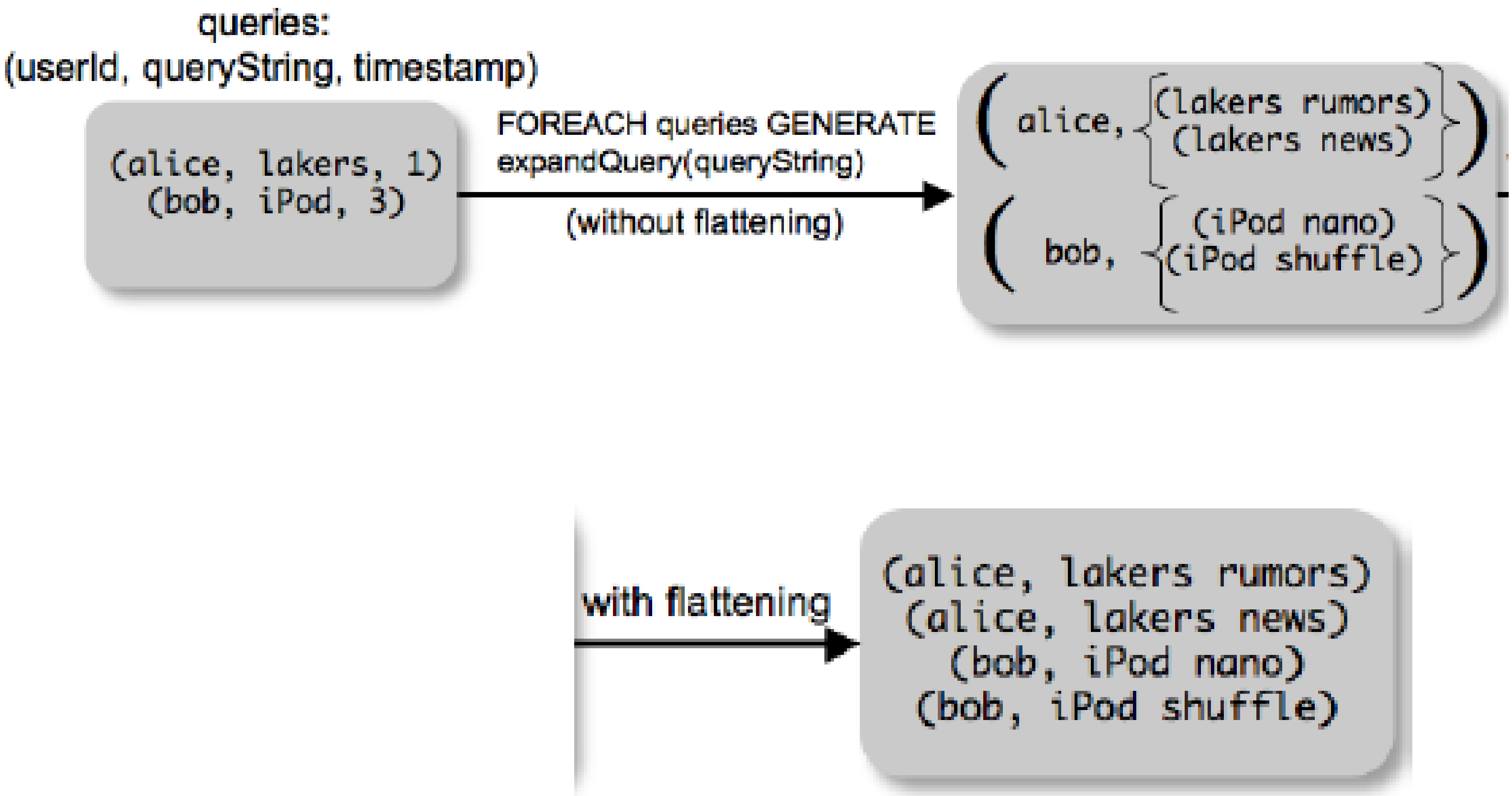
```
expanded_queries =
   FOREACH queries
   GENERATE userId, expandQuery(queryString)
```

expandQuery( ) is a UDF that produces likely expansions
Note: it returns a bag, hence expanded_queries is a nested bag

FOREACH

```
expanded_queries =
   FOREACH queries
   GENERATE userId,
                  flatten(expandQuery(queryString))
```

Now we get a flat collection



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)
with flattening
(alice, lakers rumors)
(alice, lakers news)
(bob, iPod nano)
(bob, iPod shuffle)

FLATTEN

Note that it is NOT a first class function !

- First class FLATTEN:
  - FLATTEN({{2,3},{5},{},{4,5,6}}) = {2,3,5,4,5,6}
  - Type: {{T}} → {T}
- Pig-latin FLATTEN
  - FLATTEN({4,5,6}) = 4, 5, 6
  - Type: {T} → T, T, T, …, T ?????

FILTER

Remove all queries from Web bots:

```
real_queries =  FILTER queries BY userId neq ‘bot’
```

Better: use a complex UDF to detect Web bots:

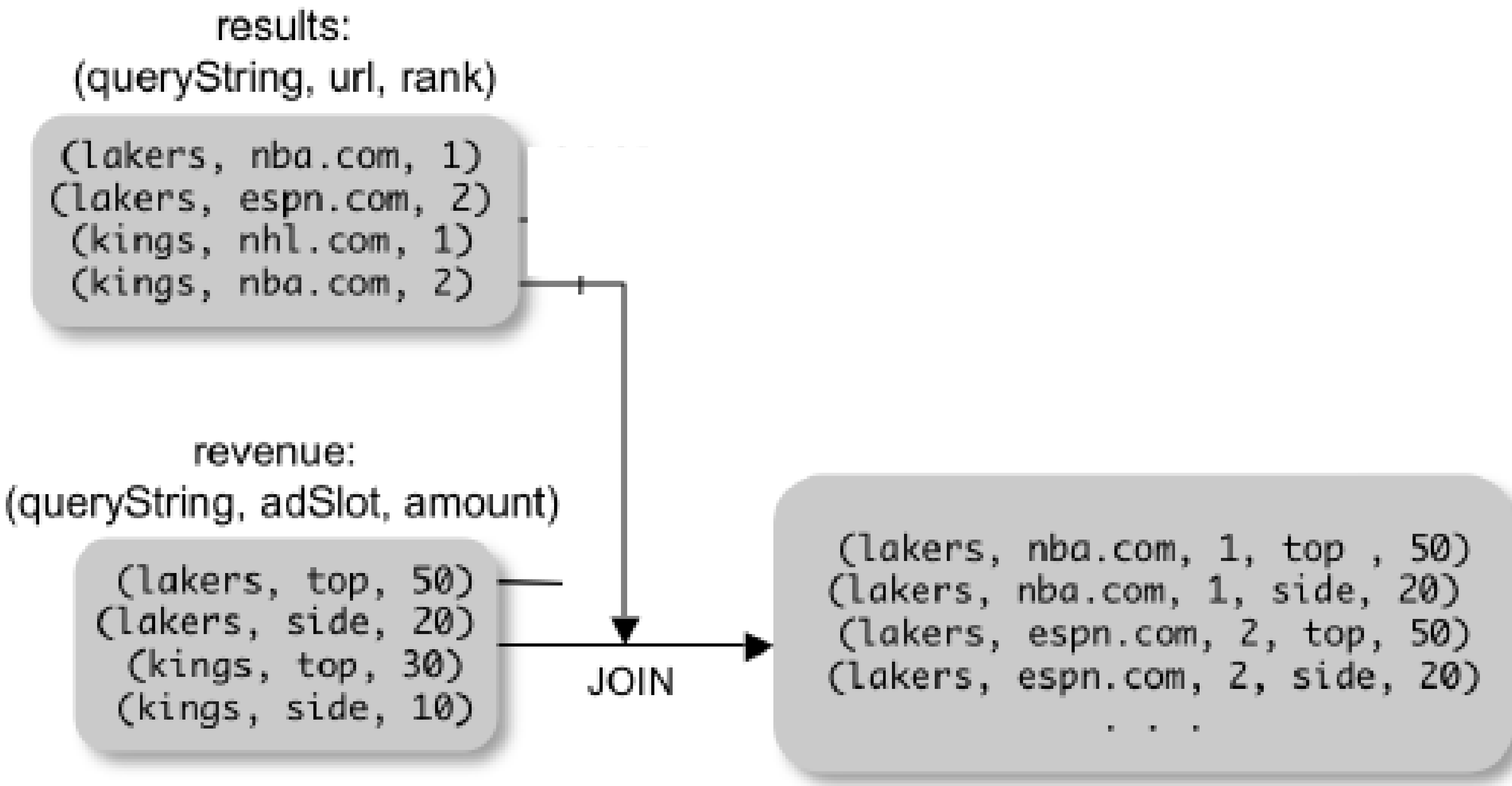
```
real_queries =  FILTER queries
                        BY NOT isBot(userId)
```

JOIN

results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

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

join_result : {(queryString, url, position, adSlot, amount)}


results:
(queryString, url, rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)
revenue:
(queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)
JOIN
(lakers, nba.com, 1, top , 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)
. . .

GROUP BY

revenue: {(queryString, adSlot, amount)}

```
grouped_revenue = GROUP revenue BY queryString
query_revenues =
    FOREACH grouped_revenue
    GENERATE queryString,
             SUM(revenue.amount) AS totalRevenue
```

grouped_revenue: {(queryString, {(adSlot, amount)})}
query_revenues: {(queryString, totalRevenue)}

Cogroup

- A generic way to group tuples from two datasets together

Co-Group

Dataset 1 results: {(queryString, url, position)}
Dataset 2 revenue: {(queryString, adSlot, amount)}

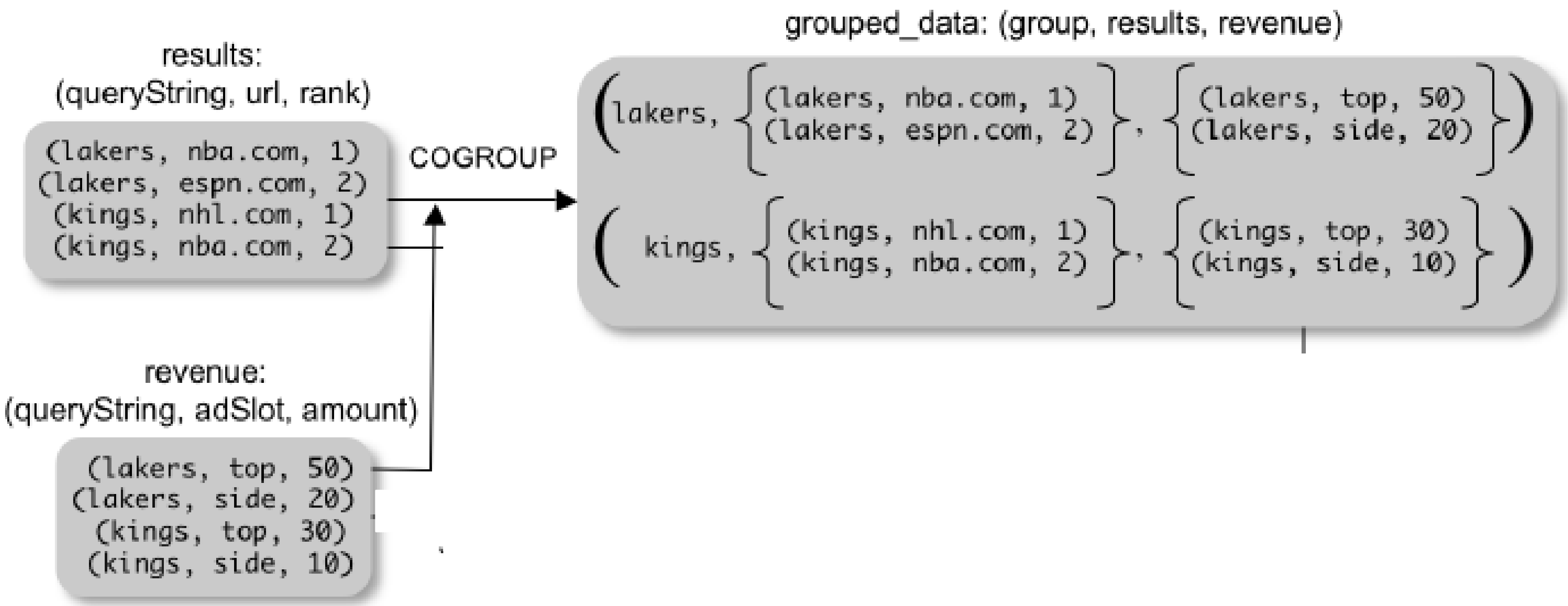
```
grouped_data =
      COGROUP results BY queryString,
                   revenue BY queryString;
```

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

What is the output type in general ?

{group_id, bag dataset 1, bag dataset 2}

Co-Group



grouped_data: (group, results, revenue)
results:
(queryString, url, rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)
COGROUP
revenue:
(queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)
(lakers, {(lakers, nba.com, 1) (lakers, espn.com, 2)}, {(lakers, top, 50) (lakers, side, 20)})
(kings, {(kings, nhl.com, 1) (kings, nba.com, 2)}, {(kings, top, 30) (kings, side, 10)})

Co-Group

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

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

distributeRevenue is a UDF that accepts search results and
revenue information for a query string at a time,
and outputs a bag of urls and the revenue attributed to them.

Co-Group v.s. Join

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

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

Result is the same as JOIN

Asking for Output: STORE

```
STORE query_revenues INTO `myoutput'
USING myStore();
```

Meaning: write query_revenues to the file 'myoutput'

This is when the entire query is finally executed!

First in SQL…

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

…then in 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)
```

Pig Latin combines

- high-level declarative querying in the spirit of SQL, and
- low-level, procedural programming a la map-reduce.

Another Example

raw (from, to, amount date)

raw2 (name, phonenumber)

In Pig, how would we write

SELECT from, SUM(amount) *

FROM transactions *

GROUP BY from

SQL

```
SELECT from, SUM(amount) *
FROM transactions *
GROUP BY from
```

PIG

```
grouped = GROUP raw BY from;
grouped2 = FOREACH grouped GENERATE $0 as from,
   SUM(raw.amount) as total;
```

Another Example Extended

In Pig, how would we write

SELECT from, SUM(amount) *

FROM transactions *

GROUP BY from

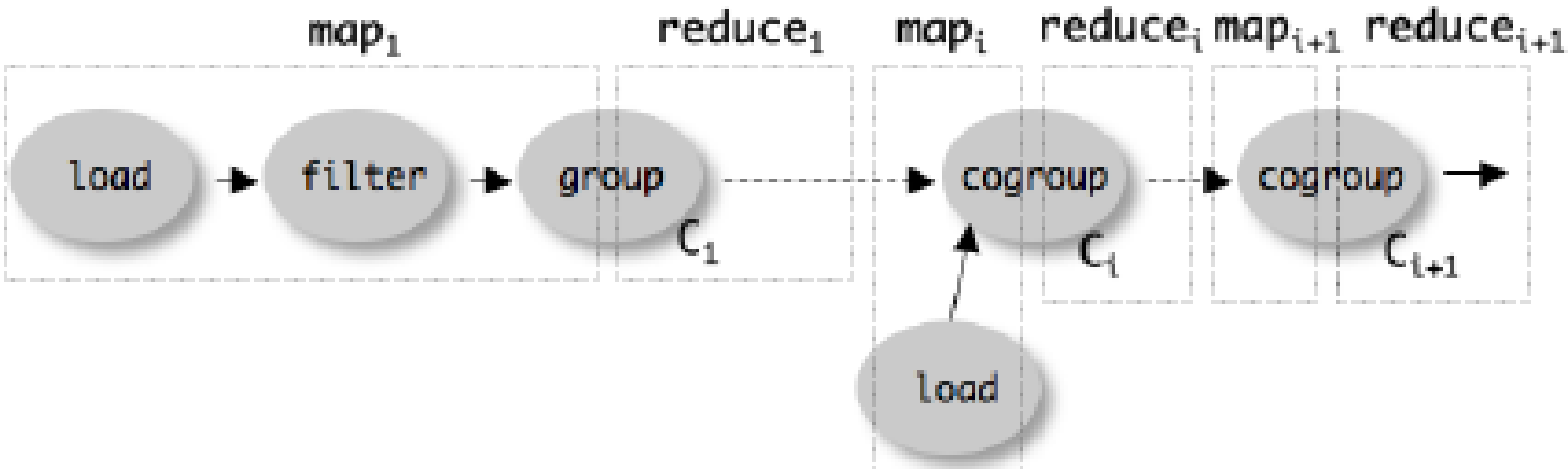
HAVING SUM(amount) >= 150 * ORDER BY SUM(amount) DESC;

```
grouped = GROUP raw BY from;
grouped2 = FOREACH grouped GENERATE $0 as
  from, SUM(raw.amount) as total;
grouped3 = FILTER grouped2 BY (total >= 150);
grouped4 = ORDER grouped3 BY total DESC;
```

Implementation

- Over Hadoop !
- Parse query:
  - All between LOAD and STORE → one logical plan
- Logical plan → ensemble of MapReduce jobs
  - Each (CO)Group becomes a MapReduce job
  - Other ops merged into Map or Reduce operators

Implementation



Query Processing Steps