SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (5/6)

- large systems
  - global scale data management
  - map/reduce
  - pig and pig latin
  - search engines
  - data warehouses and OLAP
- write-optimized system concepts
  - OLTP
  - publish/subscribe
  - streaming
  - moving objects
- management of geographical data
  - basic concepts
  - GIS, google maps

What is Pig Latin?

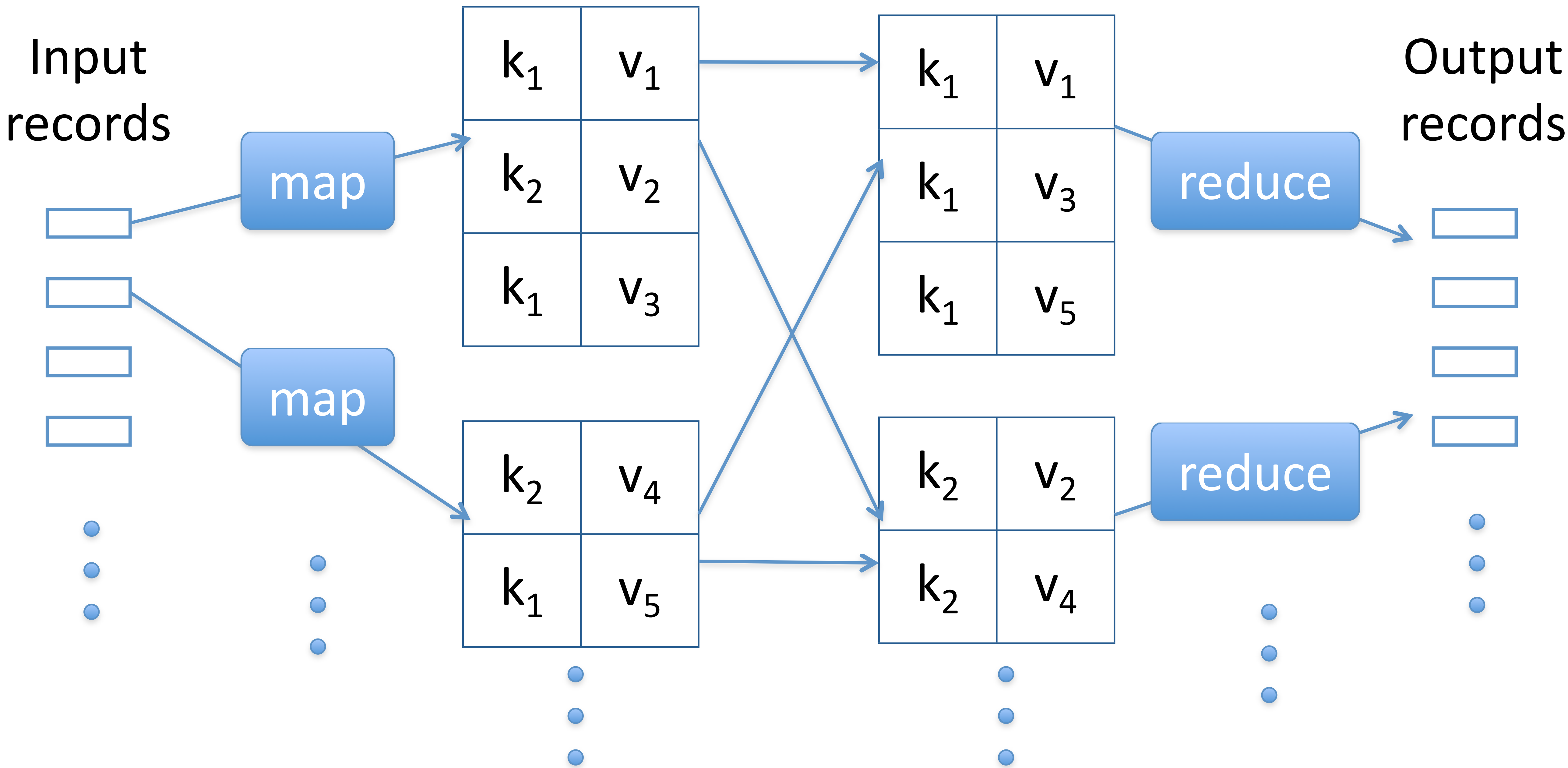
- game of alterations for English language played by children
- two major rules
- (1) if a word w starts with a consonant:
  move consonant to end of the word and add “-ay“
- examples:
  - beast → east-bay
  - happy → appy-hay
  - question → estion-quay
  - star → ar-stay
  - three → ee-thray

  perl → erl-pay
  python → ython-pay
  java → ava-jay
  database → database-ay
  flash → lash-fay
- (2) if a word starts with a vowel or silent consonant (like “h“):
  just end “-ay“ to the end of the word
- examples:
  - eagle → eagle-ay
  - America → America-ay
  - honor → honor-ay
- source: wikipedia

Pig and Pig Latin.

Introduction.

Recap: Map-Reduce



Just a group-by-aggregate?

The Map-Reduce Appeal

Scale

Scalable due to simpler design

- Only parallelizable operations
- No transactions

$

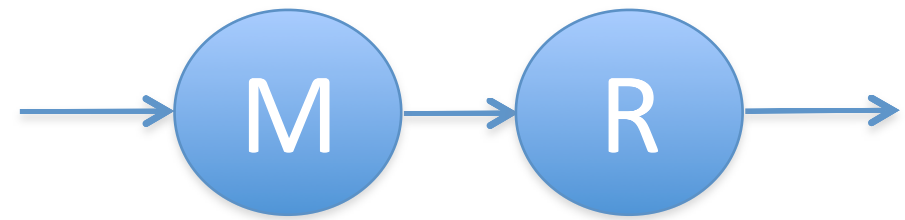
Runs on cheap commodity hardware

~~SQL~~

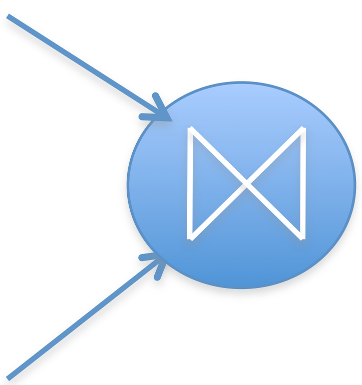
Procedural Control- a processing "pipe"

Disadvantages

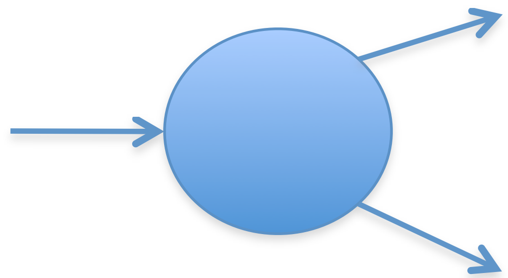
1. Extremely rigid data flow



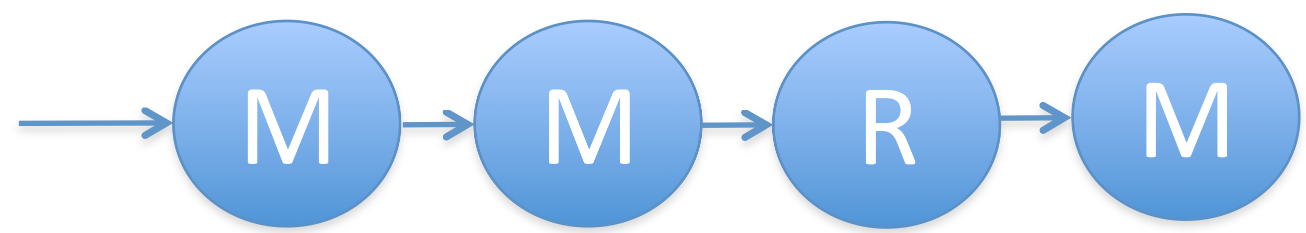
Other flows constantly hacked in



Join, Union



Split



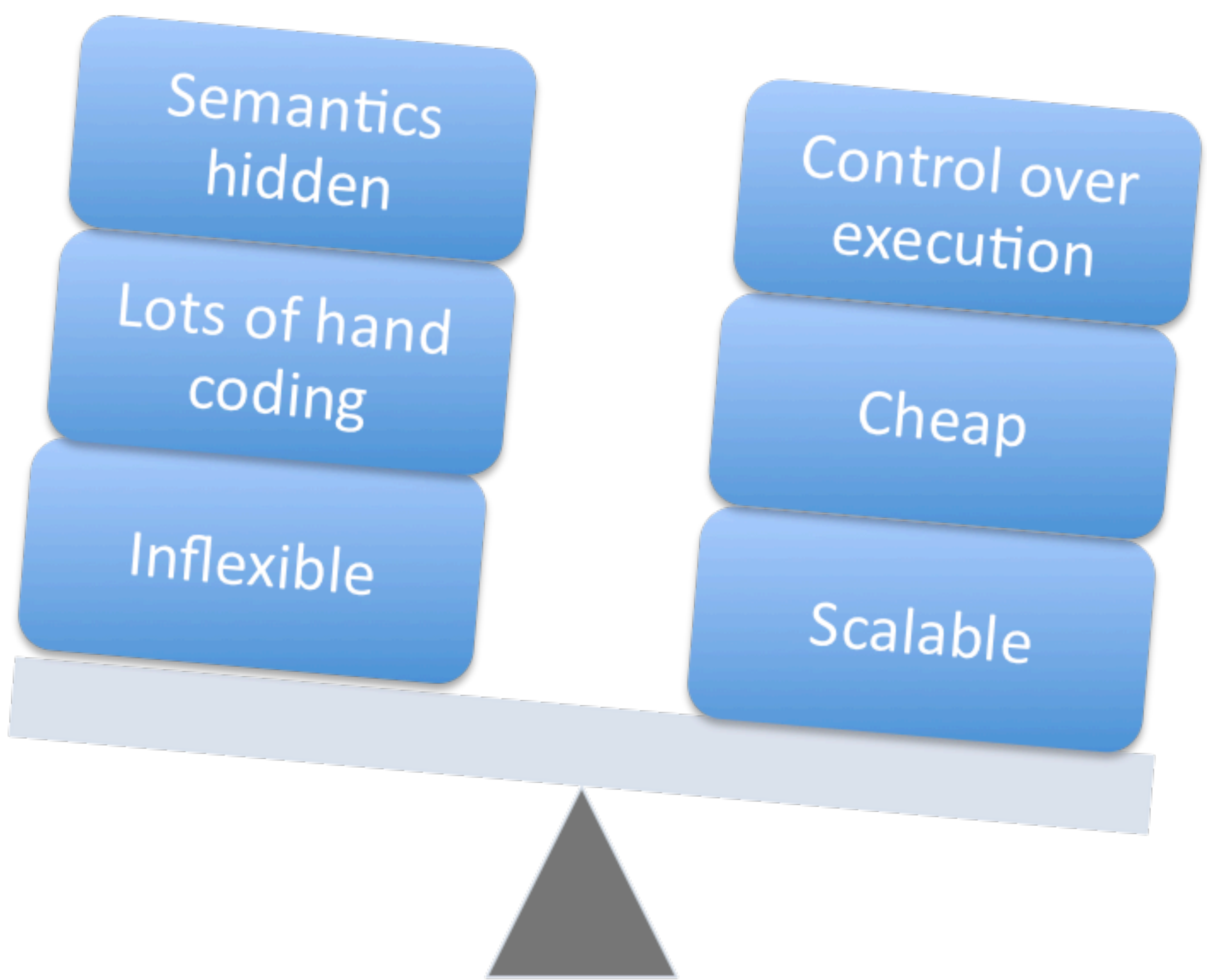
Chains

2. Common operations must be coded by hand
   - Join, filter, projection, aggregates, sorting, distinct

3. Semantics hidden inside map-reduce functions
   - Difficult to maintain, extend, and optimize

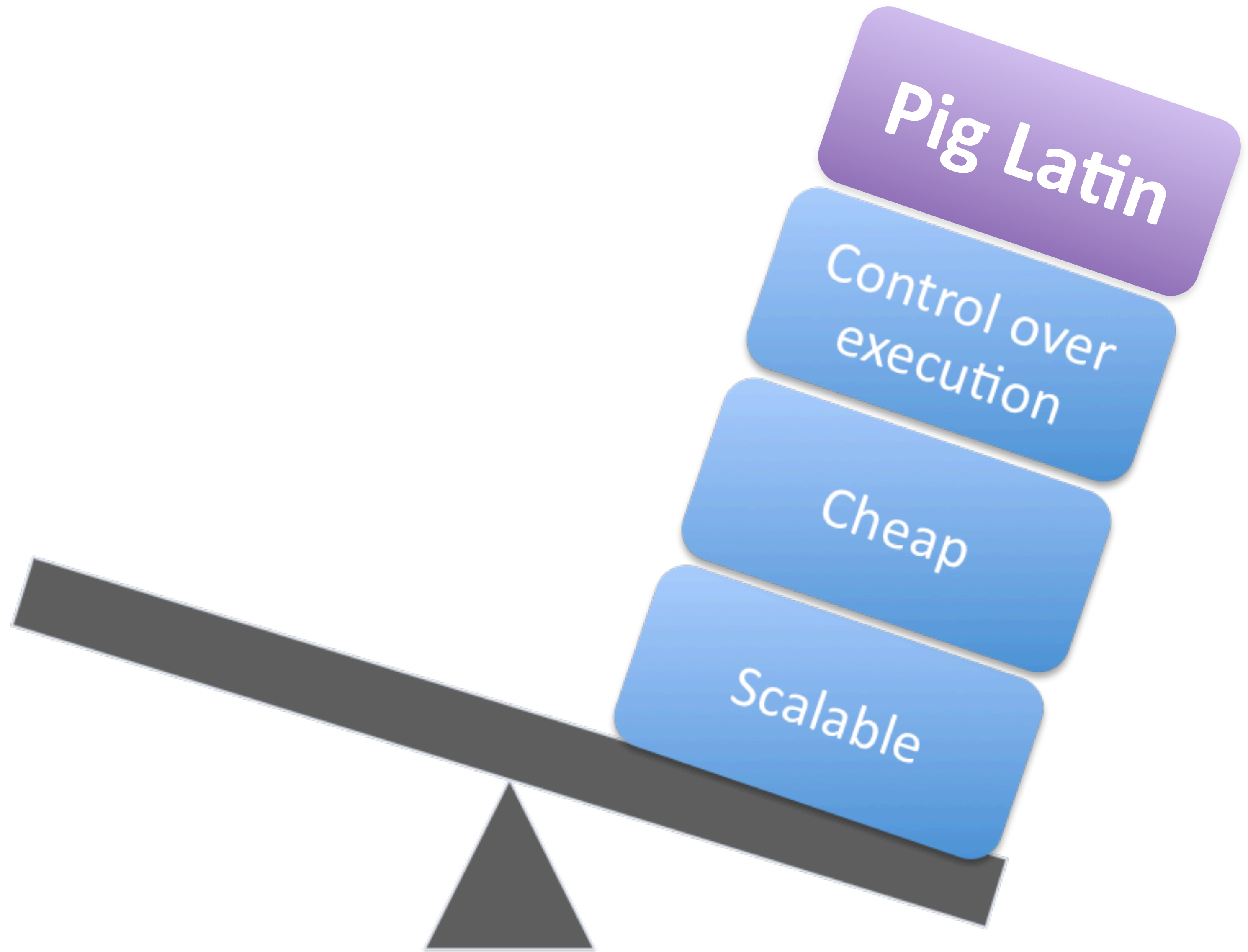
Pros And Cons

Need a high-level, general data flow language



Enter Pig Latin

Need a high-level, general data flow language



Other Approaches

- google's map/reduce -> sawzall
- Microsoft's dryad -> DryadLINQ
- hadoop -> Pig Latin
- Idea:
  - high-level declarative language
  - on top of actual query execution engine
  - translated into logical operators
  - then translated into physical map/reduce tasks
  - then executed on top of Hadoop (or any other map/reduce engine)
- Analogy:
  - Database Systems: SQL on top of query execution engine

Pig and Pig Latin.

Data Model.

Pig`s Data Model

- **atom**:
  - simple atomic value
  - e.g. string or number
  - three atoms: ‘alice‘, ‘peter‘, 42
- **tuple**:
  - sequence of fields
  - each field may be of any type
  - a tuple: (‘alice‘, ‘lakers‘)
- **bag**:
  - collection of tuples
  - possibly duplicates
  - tuples in a bag may have different types
  - tuples in a bag may have different number of fields

a bag: $\left\{ \begin{array}{c} (\text{‘alice’}, \text{‘lakers’}) \\ \big(\text{‘alice’}, (\text{‘iPod’}, \text{‘apple’})\big) \end{array} \right\}$

Data Model

- **map:**
  - collection of data items
  - associated key to look up
  - in other words: a mapping or index
  - items in map may have different type
  - keys need to be atoms
  - useful to model data sets where schemas might change over time

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

Data Model in Python

Terminal — Python — 79×22

```
[hsdpc00:~] jens% python
Python 2.5.1 (r251:54863, Jul 23 2008, 11:00:16)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> atom = 'alice'
>>> atom
'alice'
>>> tuple = (atom, 'lakers')
>>> tuple
('alice', 'lakers')
>>> bag = [tuple, tuple, (atom, 'whatever')]
>>> bag
[('alice', 'lakers'), ('alice', 'lakers'), ('alice', 'whatever')]
>>> map = {}
>>> map['fan of'] = tuple
>>> map['age'] = 20
>>> map[42] = 43
>>> map
{'fan of': ('alice', 'lakers'), 'age': 20, 42: 43}
>>>
```

Expressions in Pig Latin

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

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	$\left[\texttt{'age'} \rightarrow 20 \right]$
Projection	f2.$0	$\left\{ \begin{array}{c} (\texttt{'lakers'}) \\ (\texttt{'iPod'}) \end{array} \right\}$
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

Example Data Analysis Task

Find the top 10 most visited pages in each category

first input: Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

second input: Url Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Data Flow

Load Visits

“first input: Visits“

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

Group by url

“group visits on url“

Url	Url Group
cnn.com	{Amy, cnn.com, 8:00} {Fred, cnn.com, 12:00}
bbc.com	{Amy, bbc.com, 10:00}
flickr.com	{Amy, flickr.com, 10:05}

Foreach url generate count

“fold each url group into a single line: emit url and a count (size of the group)“

Url	Count
cnn.com	2
bbc.com	1
flickr.com	1

Load Url Info

second input: Url Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Join on url

“natural join on url“

Url	Count	Url	Category	PageRank
cnn.com	2	cnn.com	News	0.9
bbc.com	1	bbc.com	News	0.8
flickr.com	1	flickr.com	Photos	0.7

Group by category

“group urls that share the same category“

Category	Group
News	{cnn.com, 2, News, 0.9}, {bbc.com, 1, News, 0.8}
Photos	{flickr.com, 1, Photos, 0.7}

Foreach category generate top10 urls

“fold each category group into a single line: emit top10 Urls for each category“

Category	Group
News	{cnn.com, bbc.com}
Photos	{flickr.com}

Store topUrls

Same Example in Pig Latin

```
visits          = load '/data/visits' as (user, url, time);
gVisits         = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo         = load '/data/urlInfo' as (url, category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,
    10);

store topUrls into '/data/topUrls';
```

Same Example in Pig Latin

```
visits           = load '/data/visits' as (user, url, time);
gVisits          = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo          = load '/data/urlInfo' as (url, category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategorie                                                    ry;
topUrls = f                                                   (visitCounts,
    10);

store topUrls into '/data/topUrls';
```

Operates directly over files

Same Example in Pig Latin

```
visits          = load '/data/visits' as (user, url, time);
gVisits         = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo         = load '/data/urlInfo' as (url, category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategorie                                                   ;
topUrls = f                                              visitCounts,
    10);

store topUrls into '/data/topUrls';
```

Schemas optional;
Can be assigned dynamically

Same Example in Pig Latin

```
visits           = load '/data/visits' as (user, url, time);
gVisits
visitCo                                   l, count(visits);

urlInfo                                   category, pRank);
visitCounts  = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,
    10);

store topUrls into '/data/topUrls';
```

User-defined functions (UDFs) can be used in every construct

- Load, Store
- Group, Filter, Foreach

Similar Example using Pig Pen Front-end

Operators

LOAD | GROUP | COGROUP | FILTER | FOREACH | ORDER

= LOAD USING Default AS ( )

Generate Query

```
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';
```

```
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)
```

Pig and Pig Latin.

Language Definition.

Pig Latin: Principles

- pig latin program is a sequence of steps
- writing a pig latin program is similar to writing a query execution plan
- pig latin program = sequence of steps
- each step specifies a **single** high-level data transformation
- does **not preclude** query optimization
- data flow graph is just a logical description

Loading Data

```
queries = LOAD 'query_log.txt'
          USING myLoad()
          AS (userId, queryString, timestamp);
```

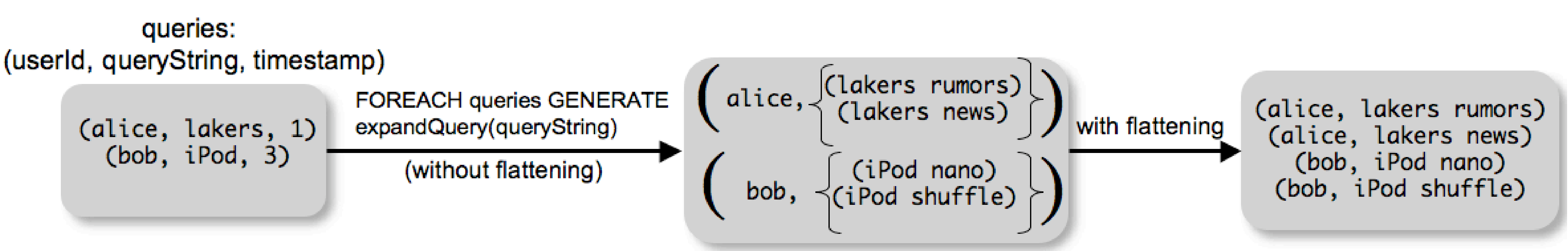
- input read from “query_log.txt“
- uses a custom myLoad() deserializer
- loaded tuples have three fields:
  (userId, queryString, timestamp)
- USING is optional
- will then try to parse tab separated fields
- query variable is called a **bag handle**
- just logical definition, nothing is executed when typing LOAD

Per-Tuple Processing: FOREACH ... GENERATE

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

- applies a function to each item
- here: expandQuery()
- result may be **nested**!
- alternative:

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



Per-Tuple Processing: Filter

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

- only elements where userID!=‘bot‘ will pass

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

- same filter using a user-defined function (UDF)

Getting Related Data Together: CoGroup

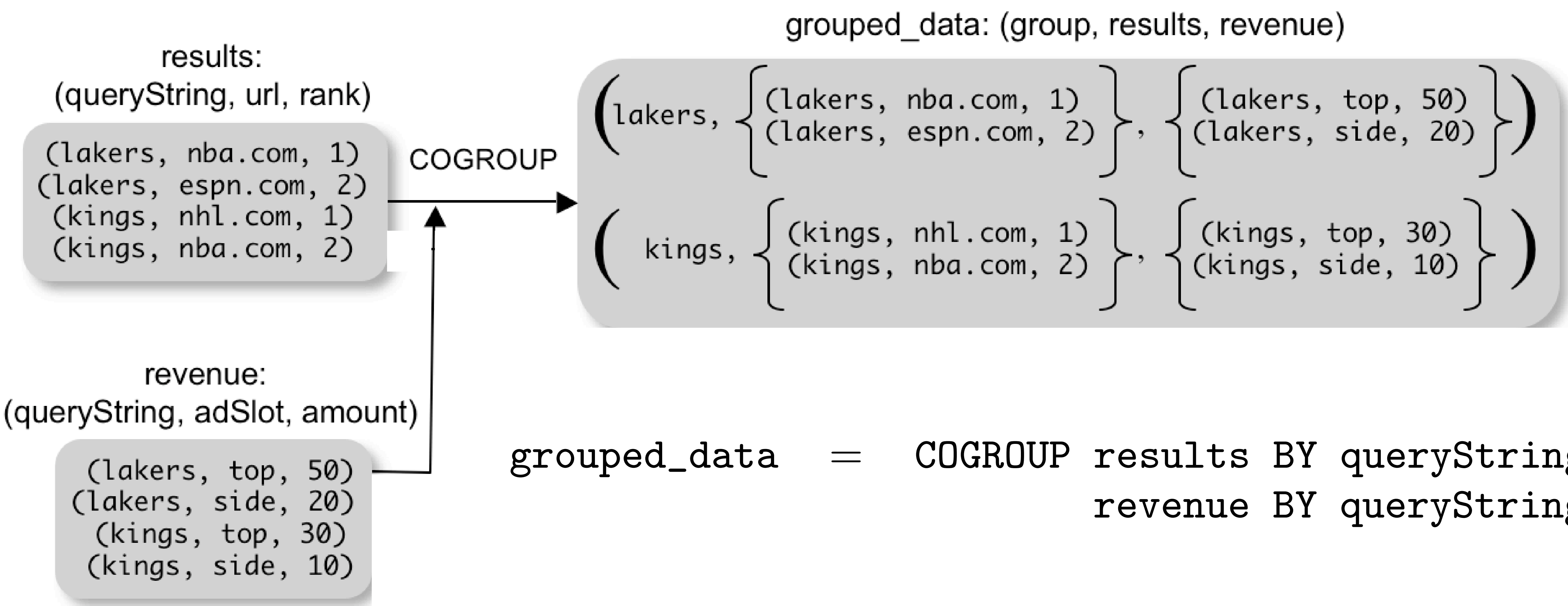
- clusters/groups similar data together
- input may be one or more data sets:
- for instance, assume two inputs:

```
results:  (queryString, url, position)
revenue:  (queryString, adSlot, amount)
```

- cogroup them with the command:

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

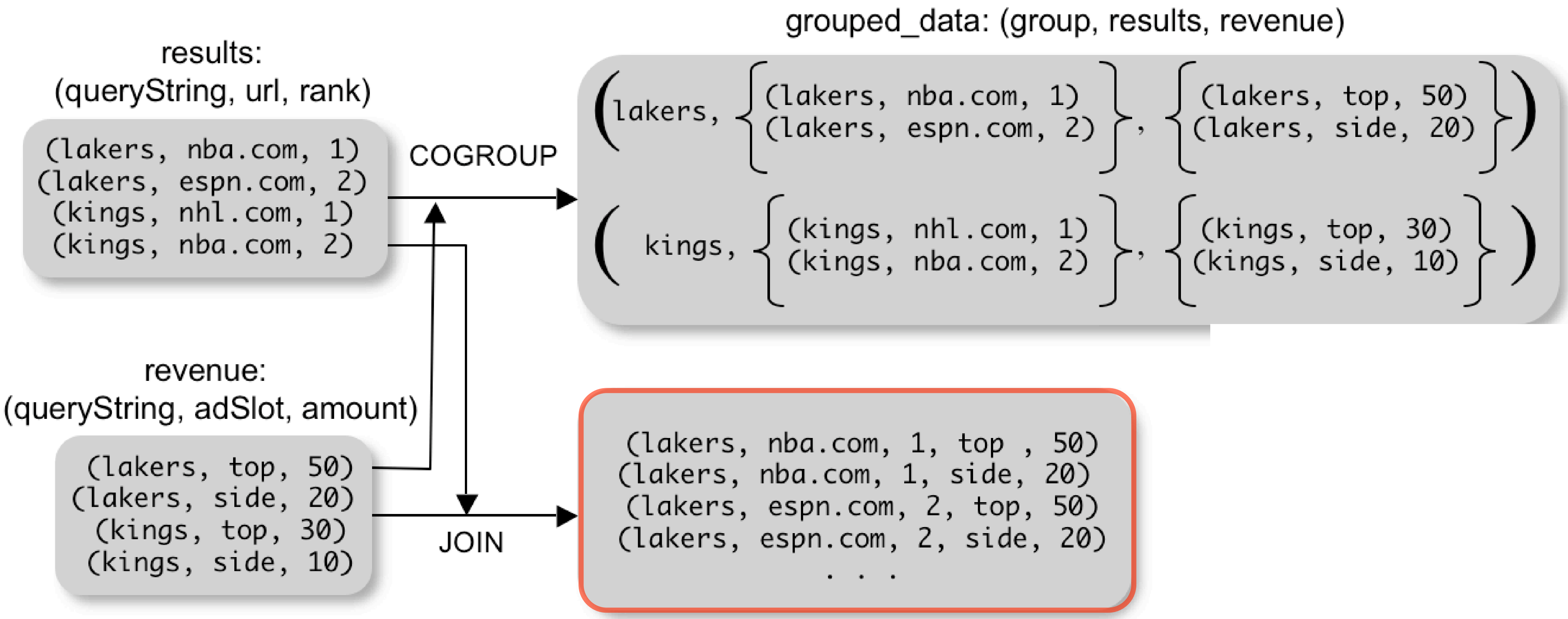
CoGroup




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

- cogroup generates **one tuple** for each group
- first field: group identifier, here: query string
- next field: bag from first input with occurrences
- i-th field: bag from i-th input with occurrences
- **cogrouping** of **multiple** data sets

CoGroup and Join




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

- cogroup similar to a join
- difference: cogroup does **not** compute cross product on bags!

Joins versus CoGroup

- writing an explicit join as:

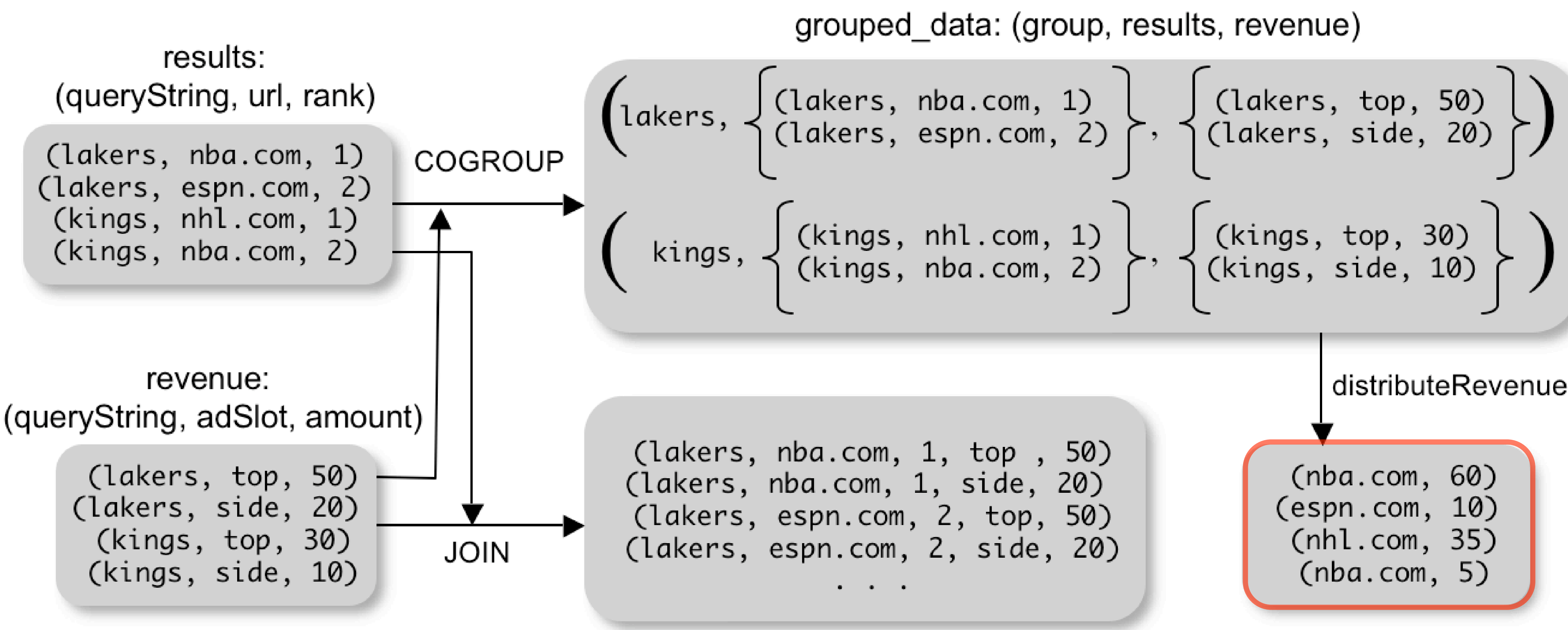
```
join_result = JOIN results BY queryString,
              revenue BY queryString;
```

- ...is just a syntactic shortcut for a COGROUP followed by flattening:

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

- flattening both groups returns cross product for each cogroup

So why make a Difference? Example:




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

- UDF distributeRevenue() takes two bags as its input
- UDF makes decisions based on seeing the **entire** bags
- UDF generates bag as a output that is the flattened

Special Case of CoGroup: Group

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

- special case of COGROUP
- just **one** data set involved!
- may also use GROUP revenue ALL to group everything

MapReduce in Pig Latin

```
map_result = FOREACH input GENERATE FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
    output = FOREACH key_groups GENERATE reduce(*);
```

=map()

=grouping

=reduce()

- * = all fields are passed
- therefore easy to express any map/reduce task in Pig Latin
- makes the intermediate grouping step explicit

Asking for Output

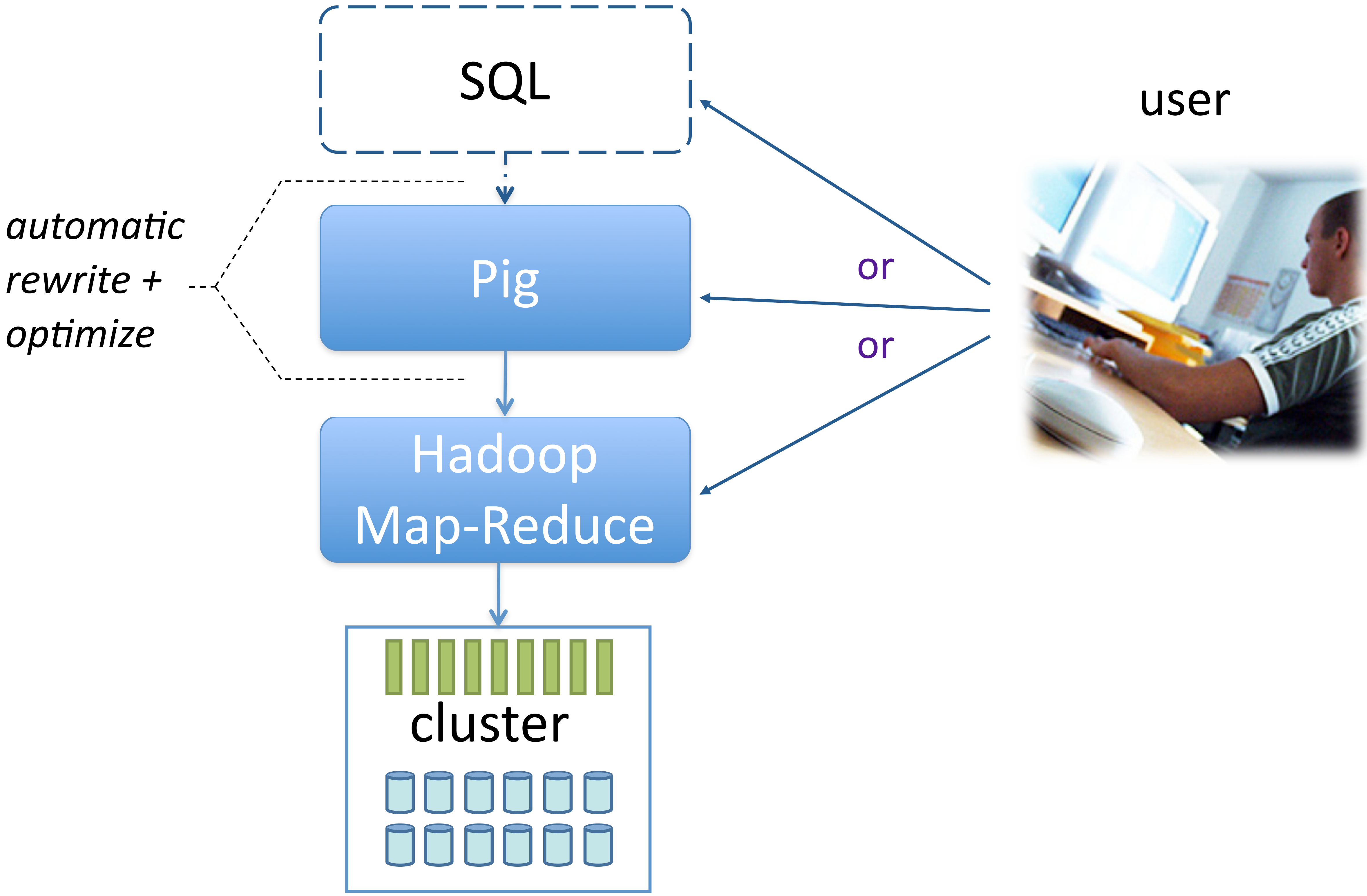
```
STORE query_revenues INTO ‘myoutput’
      USING myStore();
```

- puts output to a file ‘myoutput‘
- using optional custom serializer myStore()

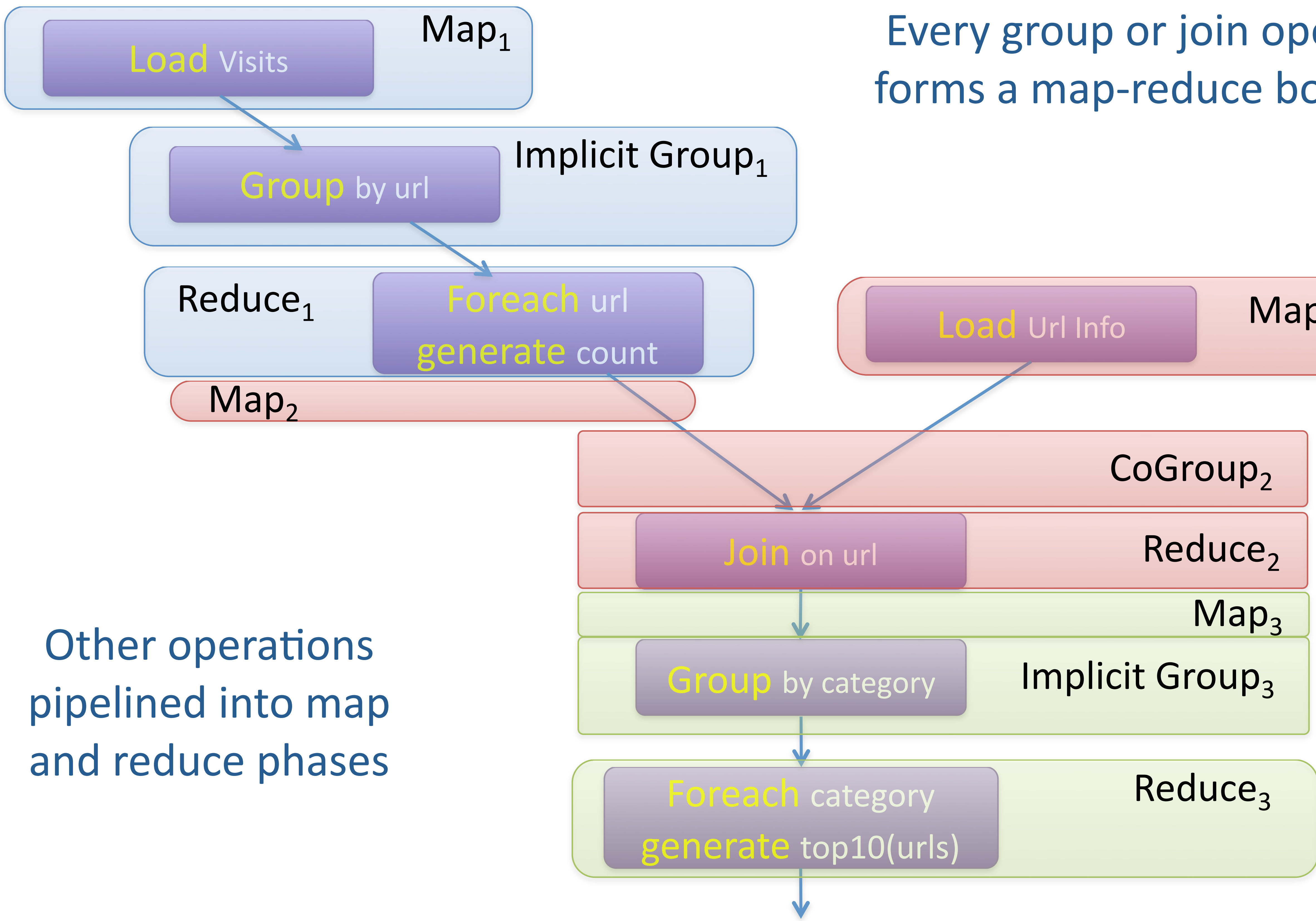
Pig and Pig Latin.

Pig Execution.

Implementation



Compilation into Map-Reduce



Every group or join operation forms a map-reduce boundary

Other operations pipelined into map and reduce phases

Pig versus DBMS

	DBMS	Pig
workload	Bulk and random reads & writes; indexes, transactions	Bulk reads & writes only
data representation	System controls data format Must pre-declare schema	Pigs eat anything
programming style	System of constraints	Sequence of steps
customizable processing	Custom functions second-class to logic expressions	Easy to incorporate custom functions

Pig versus SQL

- SQL declarative: users tells system **what** he wants to have
- Pig Latin data-flow-graph: user tells system **how** the result **could** be composed
  - close to imperative programming
  - programmers like it
  - data analysis incremental anyway
  - why not write the query incrementally?
- So Pig is equal to a hard-coded query execution plan?:
  - no!
  - does not preclude query optimization
  - data flow graph is just a logical description
  - may be executed differently

Conclusions

- Big demand for parallel data processing
  - Emerging tools that do not look like SQL DBMS
  - Programmers like dataflow pipes over static files
- Map-Reduce is too low-level and rigid
- pig provides a declarative interface on top
- mix of python and SQL
- open source implementation of pig available:
- pig is a hadoop sub-project
- http://hadoop.apache.org/pig/

Literature

- Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig latin: a not-so-foreign language for data processing. SIGMOD 2008:1099-1110
- http://hadoop.apache.org/pig/
- http://wiki.apache.org/pig/PigLatin
- http://wiki.apache.org/pig/PigTalksPapers