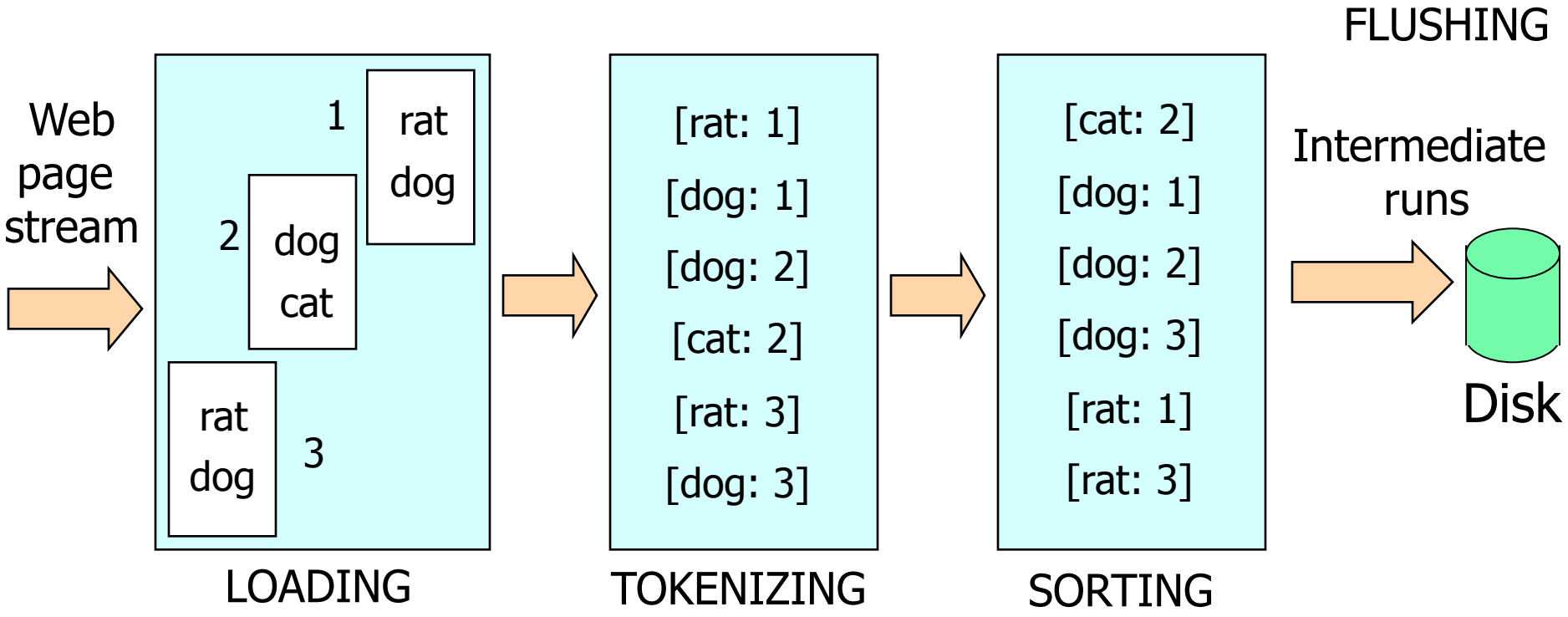
CS 347
Distributed Databases and Transaction Processing
Distributed Data Processing Using MapReduce

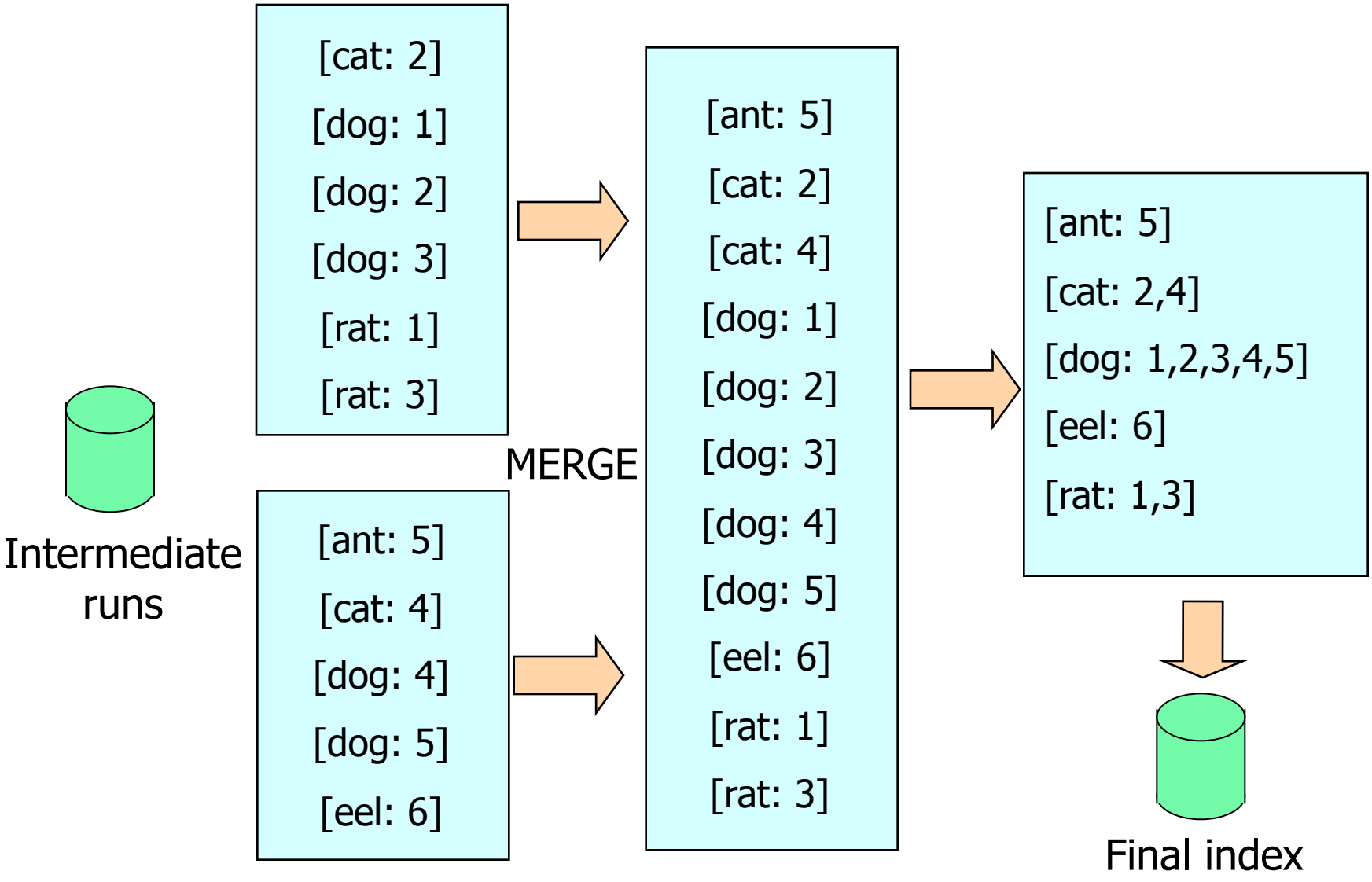
Hector Garcia-Molina

Zoltan Gyongyi

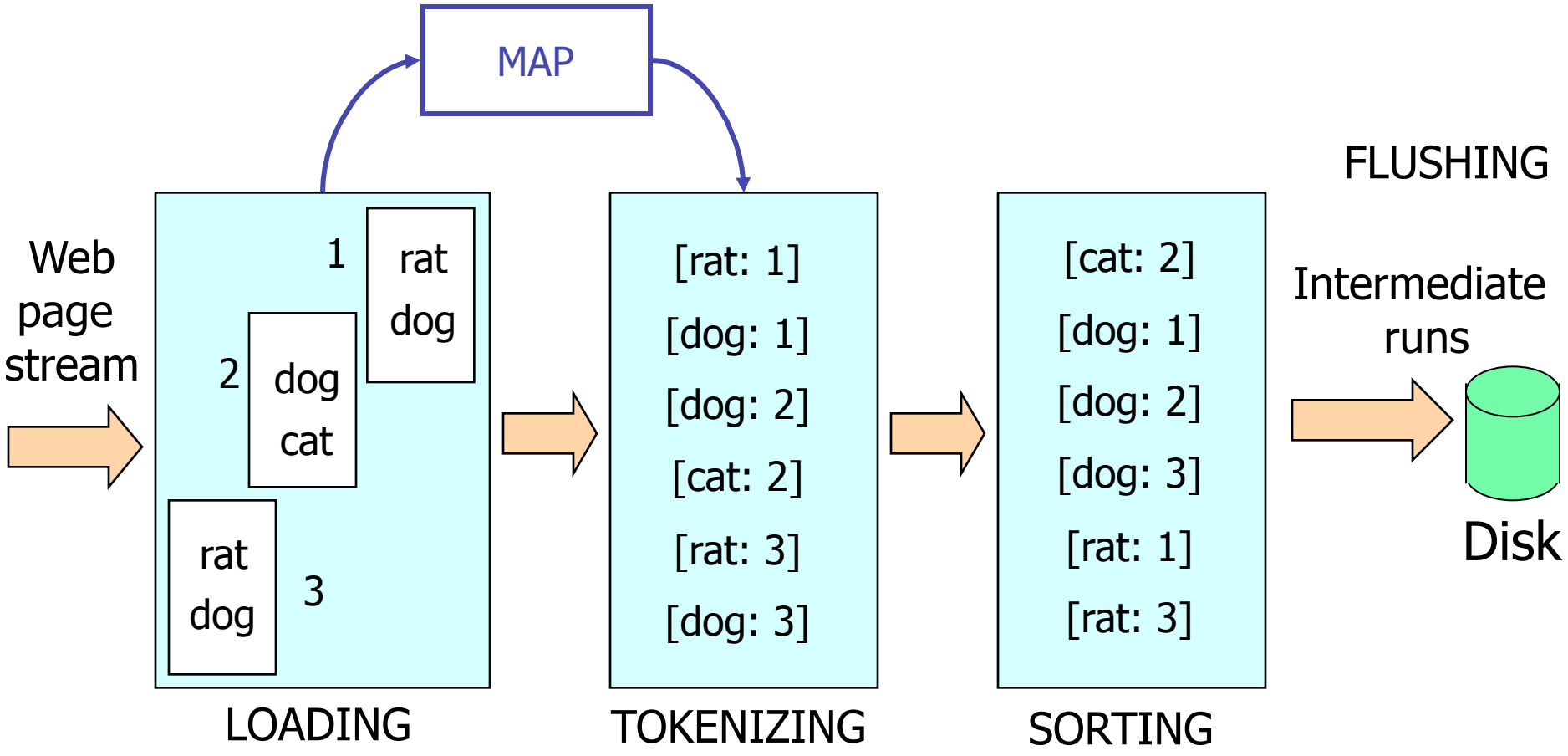
Motivation: Building a Text Index



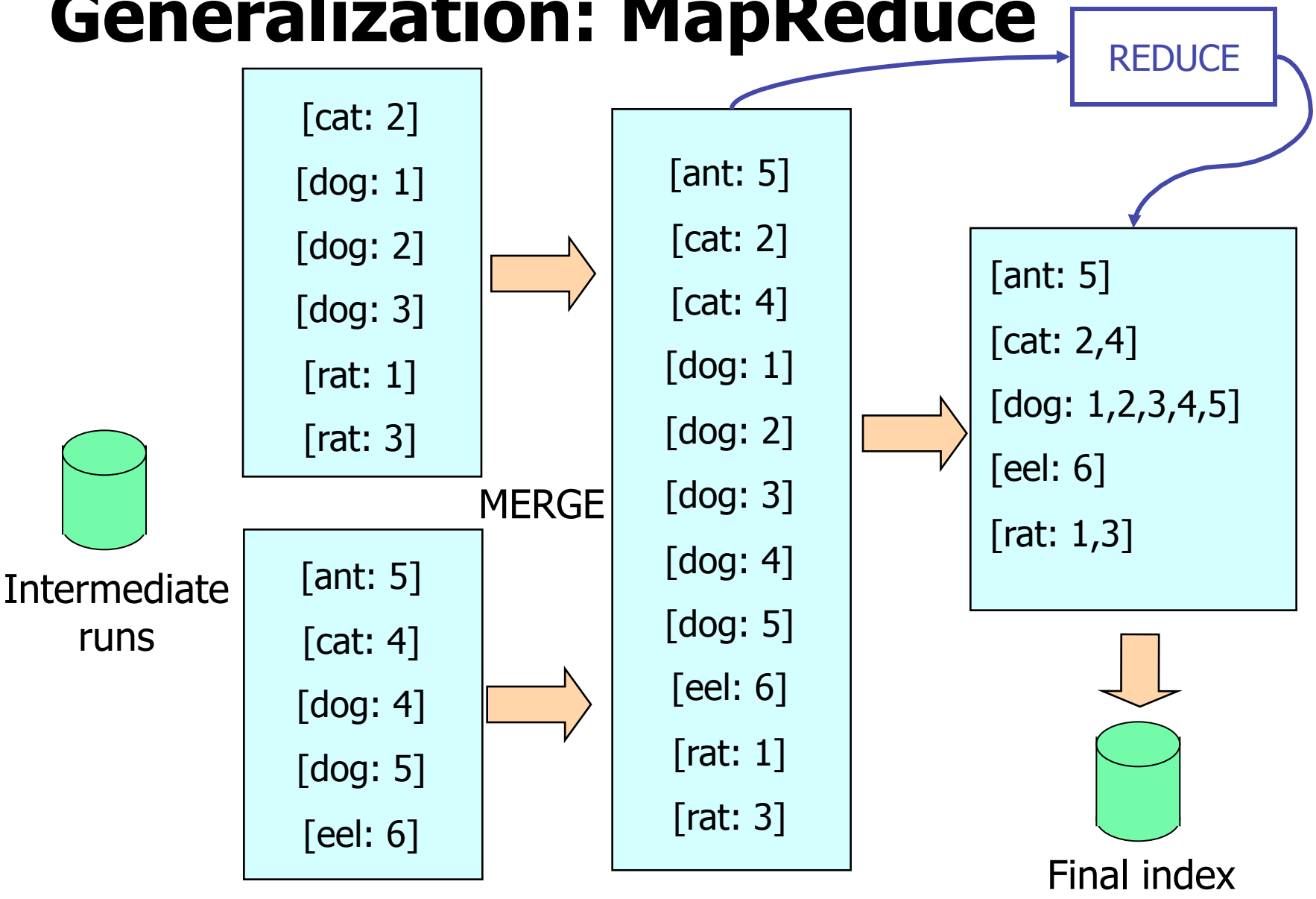
Motivation: Building a Text Index



Generalization: MapReduce



Generalization: MapReduce



MapReduce

{{ … }} = set
{ … } = bag

- Input
  - R = { r_1, r_2, …, r_n }
  - Functions ***M***, ***R***
  - ***M*** (r_i) → { [k_1: v_1], [k_2: v_2], … }
  - ***R*** (k_i, value bag) → "new" value for k_i
- Let
  - S = { [k: v] | [k: v] ∈ ***M*** (r) for some r ∈ R }
  - K = {{ k | [k: v] ∈ S, for any v }}
  - ***V*** (k) = { v | [k: v] ∈ S }
- Output
  - O = {{ [k: t] | k ∈ K, t = ***R*** (k, ***V*** (k)) }}

Example: Counting Word Occurrences

- Map(string *key*, string *value*):

  // *key* is the document ID

  // *value* is the document body

  for each word *w* in *value* :

  EmitIntermediate(*w*, '1')

- Example: Map('29875', 'cat dog cat bat dog') emits [cat: 1], [dog: 1], [cat: 1], [bat: 1], [dog: 1]
- Why does Map() have two parameters?

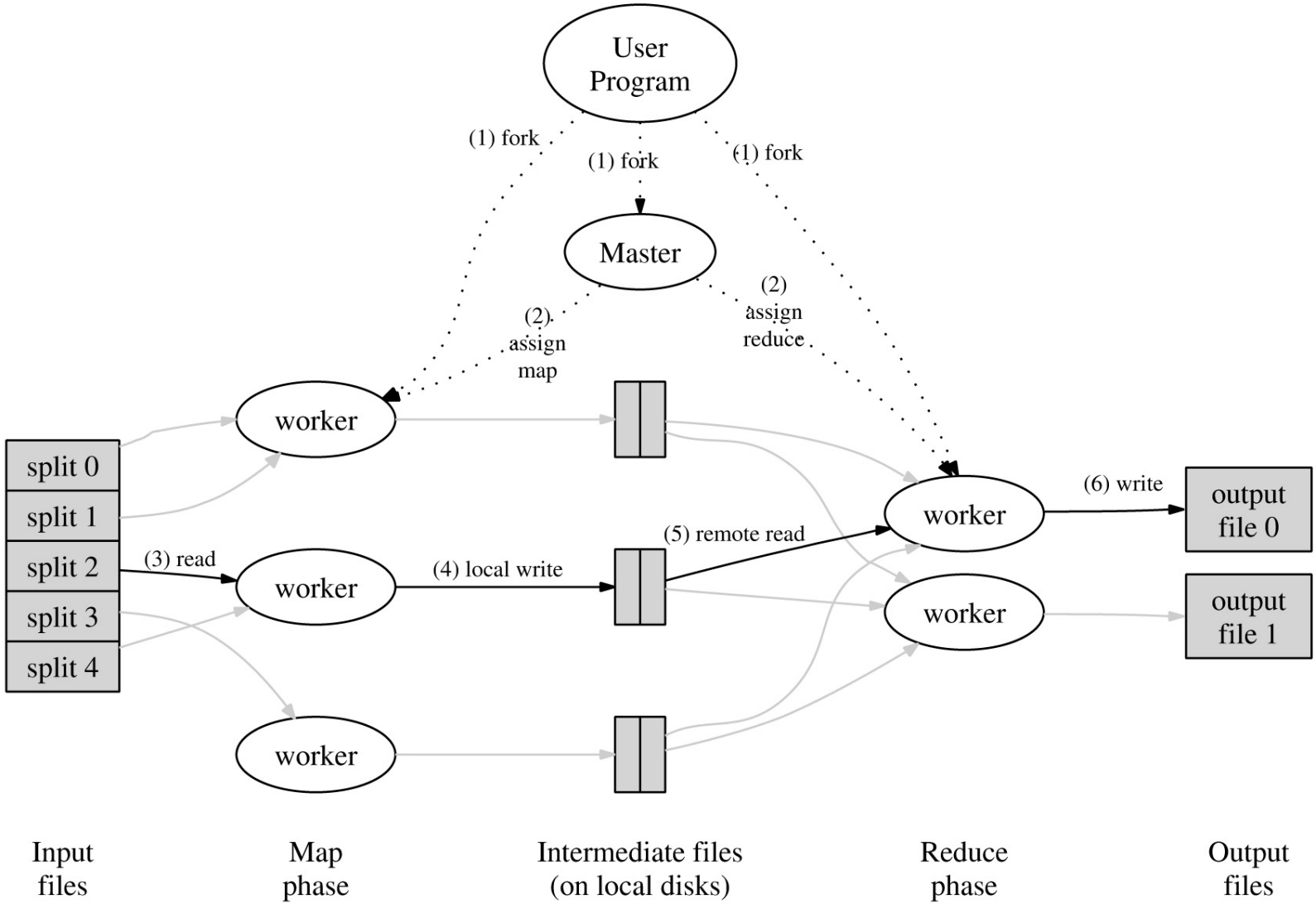
Example: Counting Word Occurrences

- Reduce(string *key*, string iterator *values*):

```
// key  is a word
// values  is a list of counts
int result = 0
for each value v in values :
    result  += ParseInteger(v)
EmitFinal(ToString(result))
```

- Example: Reduce('dog', { '1', '1', '1', '1' }) emits '4'

Google MapReduce Architecture



Implementation Issues

- File system
- Data partitioning
- Joined inputs
- Combine functions
- Result ordering
- Failure handling
- Backup tasks

File system

- All data transfer between workers occurs through distributed file system
  - Support for split files
  - Workers perform local writes
  - Each **map** worker performs *local or remote read of one or more* input splits
  - Each **reduce** worker performs *remote read of multiple* intermediate splits
  - Output is left in as many splits as reduce workers

Data partitioning

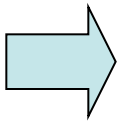
- Data partitioned (split) by hash on key
- Each worker responsible for certain hash bucket(s)
- How many workers/splits?
  - Best to have multiple splits per worker
    - Improves load balance
    - If worker fails, splits could be re-distributed across multiple other workers
  - Best to assign splits to "nearby" workers
  - Rules apply to both map and reduce workers

Joined inputs

- Two or more map inputs, partitioned (split) by same hash on key
- Map(string *key*, string iterator *values*)
  - Each value from one of the inputs
  - For some inputs, value may be empty if key is not present in the input
  - SQL "equivalent": full outer join on key

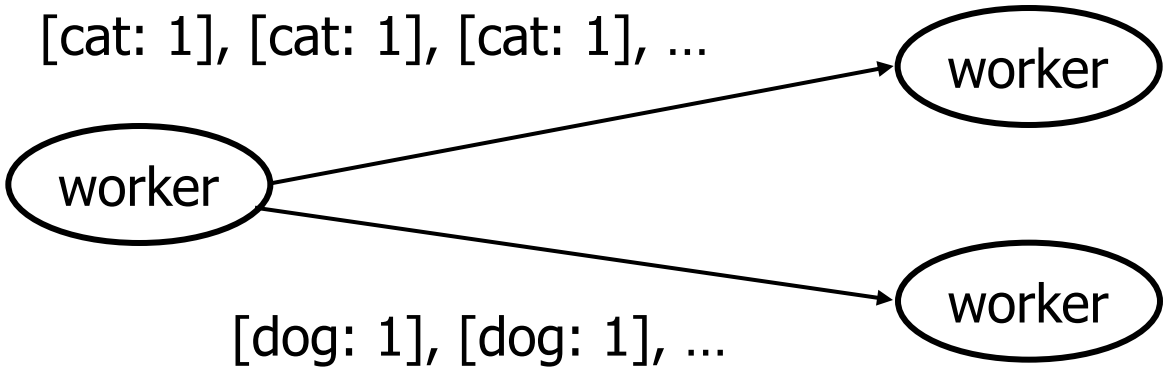
Input 1
[cat: 1]
[cow: 3]
[eel: 4]

Input 2
[ant: 6]
[cow: 3]

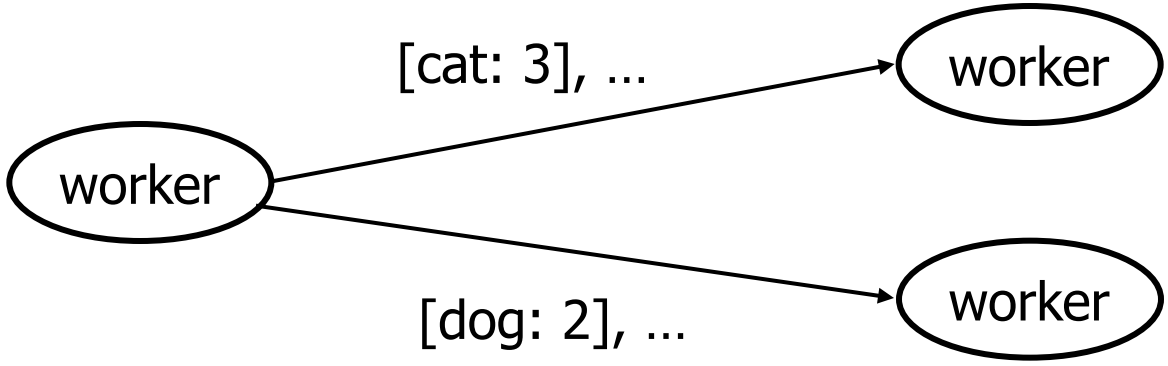


Calls
Map('ant', { ⊘, '6' })
Map('cat', { '1', ⊘ })
Map('cow', { '3', '3' })
Map('eel', { '4', ⊘ })

Combine functions

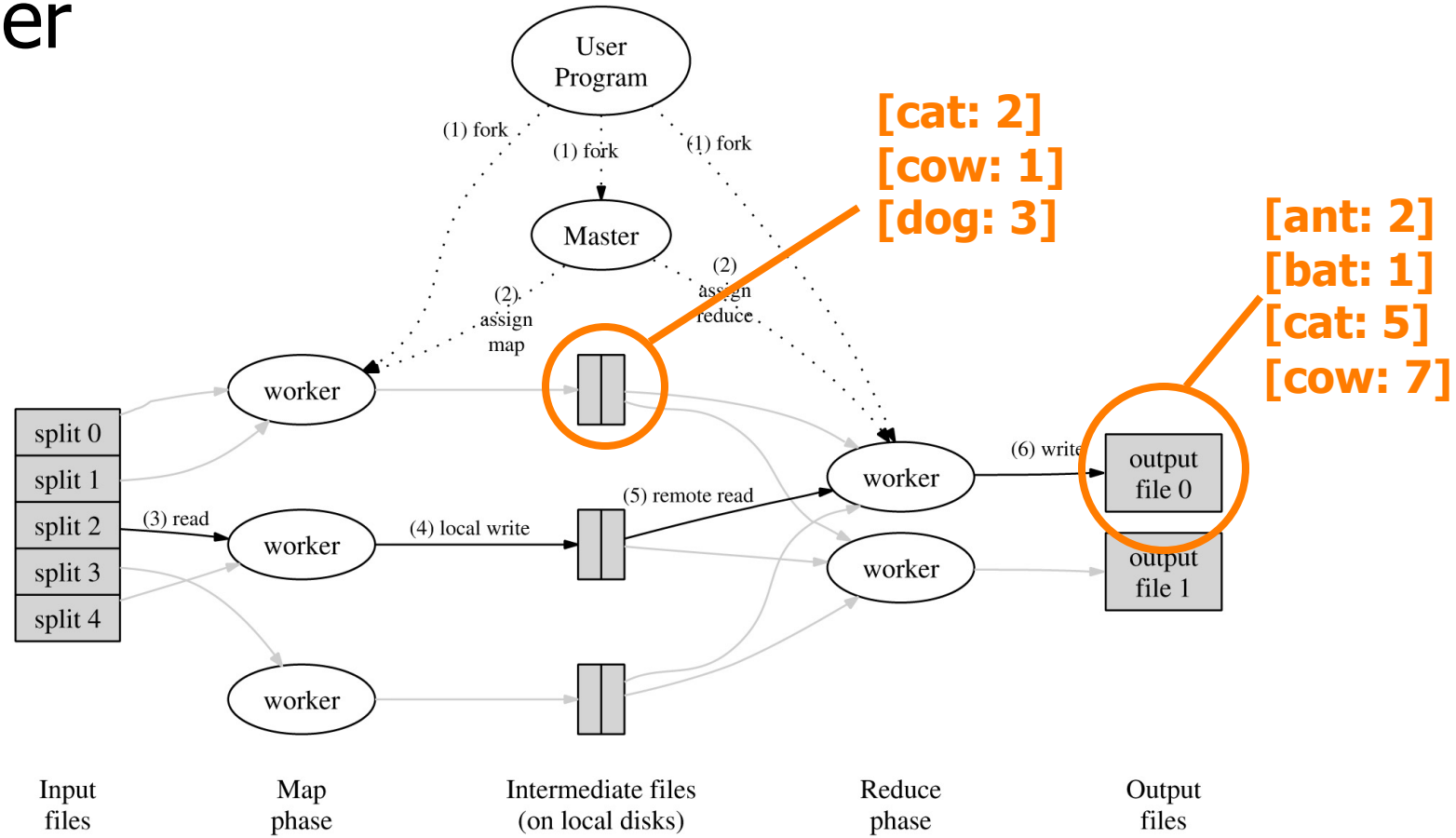


Combine is like a local reduce applied (at map worker) before storing/distributing intermediate results:



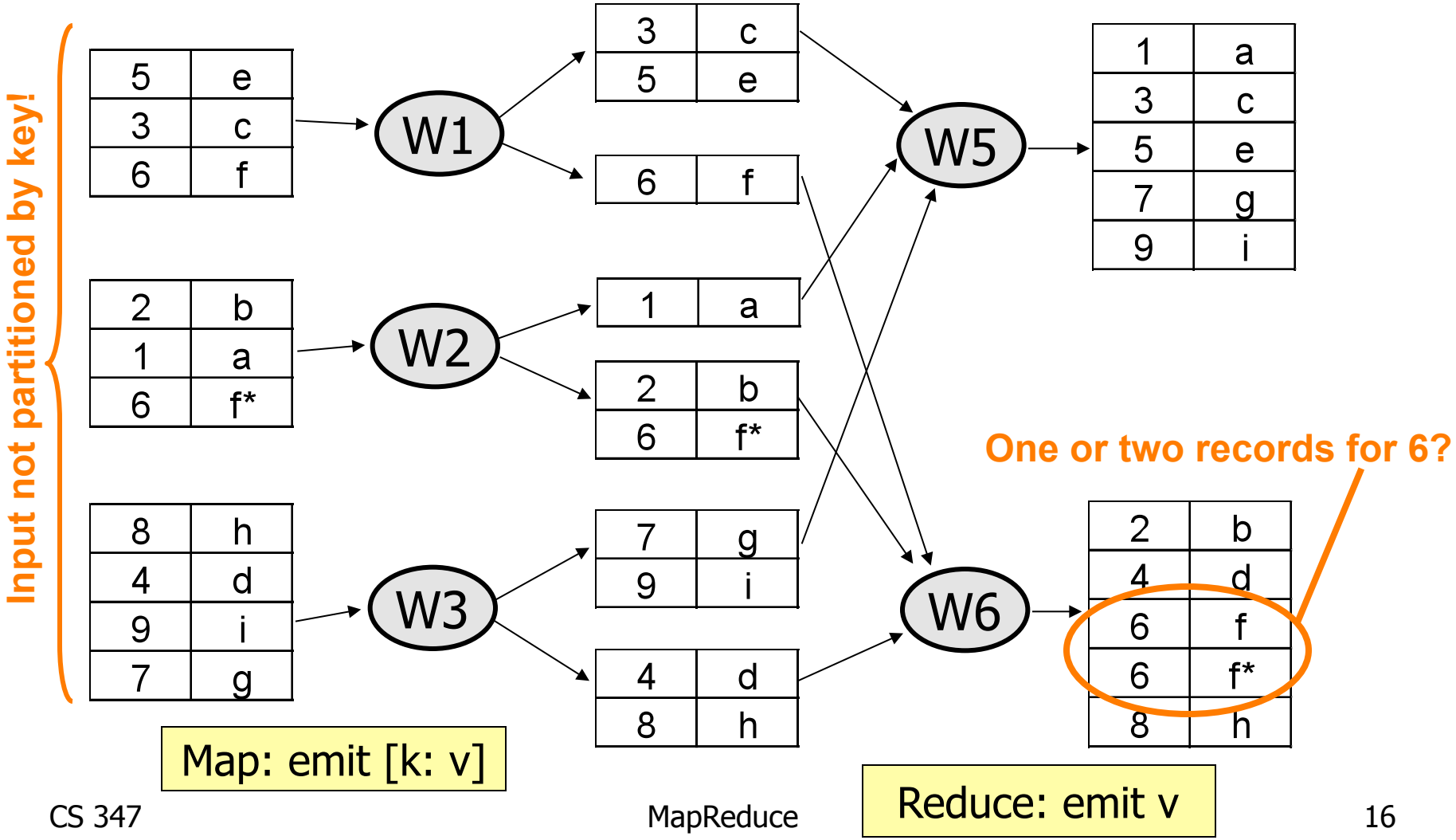
Result ordering

- Results produced by workers are in key order



Result ordering

- Example: sorting records



Failure handling

- Worker failure
  - Detected by master through periodic pings
  - Handled via re-execution
    - Redo in-progress or completed map tasks
    - Redo in-progress reduce tasks
    - Map/reduce tasks committed through master
- Master failure
  - Not covered in original implementation
  - Could be detected by user program or monitor
  - Could recover persistent state from disk

Backup tasks

- Straggler: worker that takes unusually long to finish task
  - Possible causes include bad disks, network issues, overloaded machines
- Near the end of the map/reduce phase, master spawns backup copies of remaining tasks
  - Use workers that completed their task already
  - Whichever finishes first "wins"

Other Issues

- Handling bad records
  - Best is to debug and fix data/code
  - If master detects at least 2 task failures for a particular input record, skips record during 3rd attempt
- Debugging
  - Tricky in a distributed environment
  - Done through log messages and counters

MapReduce Advantages

- Easy to use
- General enough for expressing many practical problems
- Hides parallelization and fault recovery details
- Scales well, way beyond thousands of machines and terabytes of data

MapReduce Disadvantages

- One-input two-phase data flow rigid, hard to adapt
  - Does not allow for stateful multiple-step processing of records
- Procedural programming model requires (often repetitive) code for even the simplest operations (e.g., projection, filtering)
- Opaque nature of the map and reduce functions impedes optimization

Questions

- Could MapReduce be made more declarative?
- Could we perform (general) joins?
- Could we perform grouping?
  - As done through GROUP BY in SQL

Pig & Pig Latin

- Layer on top of MapReduce (Hadoop)
  - Hadoop is an open-source implementation of MapReduce
  - Pig is the system
  - Pig Latin is the language, a hybrid between
    - A high-level declarative query language, such as SQL
    - A low-level procedural language, such as C++/ Java/Python typically used to define Map() and Reduce()

Example: Average score per category

- Input table: pages(url, category, score)
- Problem: find, for each sufficiently large category, the average score of high-score web pages in that category
- SQL solution:

```
SELECT category, AVG(score)
FROM pages
WHERE score > 0.5
GROUP BY category HAVING COUNT(*) > 1M
```

Example: Average score per category

- SQL solution:

```
SELECT category, AVG(score)
FROM pages
WHERE score > 0.5
GROUP BY category HAVING COUNT(*) > 1M
```

- Pig Latin solution:

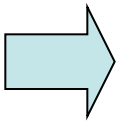
```
topPages = FILTER pages BY score > 0.5;
groups = GROUP topPages BY category;
largeGroups = FILTER groups
                BY COUNT(topPages) > 1M;
output = FOREACH largeGroups
           GENERATE category, AVG(topPages.score);
```

Example: Average score per category

```
topPages = FILTER pages BY score >= 0.5;
```

pages:
url, category, score

(cnn.com,	com,	0.9)
(yale.edu,	edu,	0.5)
(berkeley.edu,	edu,	0.1)
(nytimes.com,	com,	0.8)
(stanford.edu,	edu,	0.6)
(irs.gov,	gov,	0.7)



topPages:
url, category, score

(cnn.com,	com,	0.9)
(yale.edu,	edu,	0.5)
(nytimes.com,	com,	0.8)
(stanford.edu,	edu,	0.6)
(irs.gov,	gov,	0.7)

Example: Average score per category

```
groups = GROUP topPages BY category;
```

topPages:
url, category, score

(cnn.com,	com,	0.9)
(yale.edu,	edu,	0.5)
(nytimes.com,	com,	0.8)
(stanford.edu,	edu,	0.6)
(irs.gov,	gov,	0.7)

groups:
category, topPages

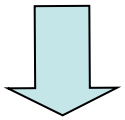
(com,	{(cnn.com, com, 0.9), (nytimes.com, com, 0.8)}	)
(edu,	{(yale.edu, edu, 0.5), (stanford.edu, edu, 0.6)}	)
(gov,	{(irs.gov, gov, 0.7)}	)

Example: Average score per category

```
largeGroups = FILTER groups BY COUNT(topPages) > 1;
```

groups:
category, topPages

(com, {(cnn.com, com, 0.9), (nytimes.com, com, 0.8)})
(edu, {(yale.edu, edu, 0.5), (stanford.edu, edu, 0.6)})
(gov, {(irs.gov, gov, 0.7)})



largeGroups:
category, topPages

(com, {(cnn.com, com, 0.9), (nytimes.com, com, 0.8)})
(edu, {(yale.edu, edu, 0.5), (stanford.edu, edu, 0.6)})

Example: Average score per category

```
output = FOREACH largeGroups GENERATE category, AVG(topPages.score);
```

largeGroups:
category, topPages

(com, {(cnn.com, com, 0.9), (nytimes.com, com, 0.8)})
(edu, {(yale.edu, edu, 0.5), (stanford.edu, edu, 0.6)})

output:
category, topPages

(com, 0.85)
(edu, 0.55)

Pig (Latin) Features

- Similar to specifying a query execution plan (i.e., data flow graph)
  - Makes it easier for programmers to understand and control execution
- Flexible, fully nested data model
- Ability to operate over input files without schema information
- Debugging environment

Execution control: good or bad?

- Example:

```
spamPages = FILTER pages BY isSpam(url);
culpritPages = FILTER spamPages BY score > 0.8;
```

- Should system reorder filters?
  - Depends on selectivity

Data model

- **Atom**, e.g., 'alice'
- **Tuple**, e.g., ('alice', 'lakers')
- **Bag**, e.g.,
  { ('alice', 'lakers'), ('alice', ('iPod', 'apple')) }
- **Map**, e.g.,
  ['fan of' → { ('lakers'), ('iPod') },
    'age' → 20]

Expressions

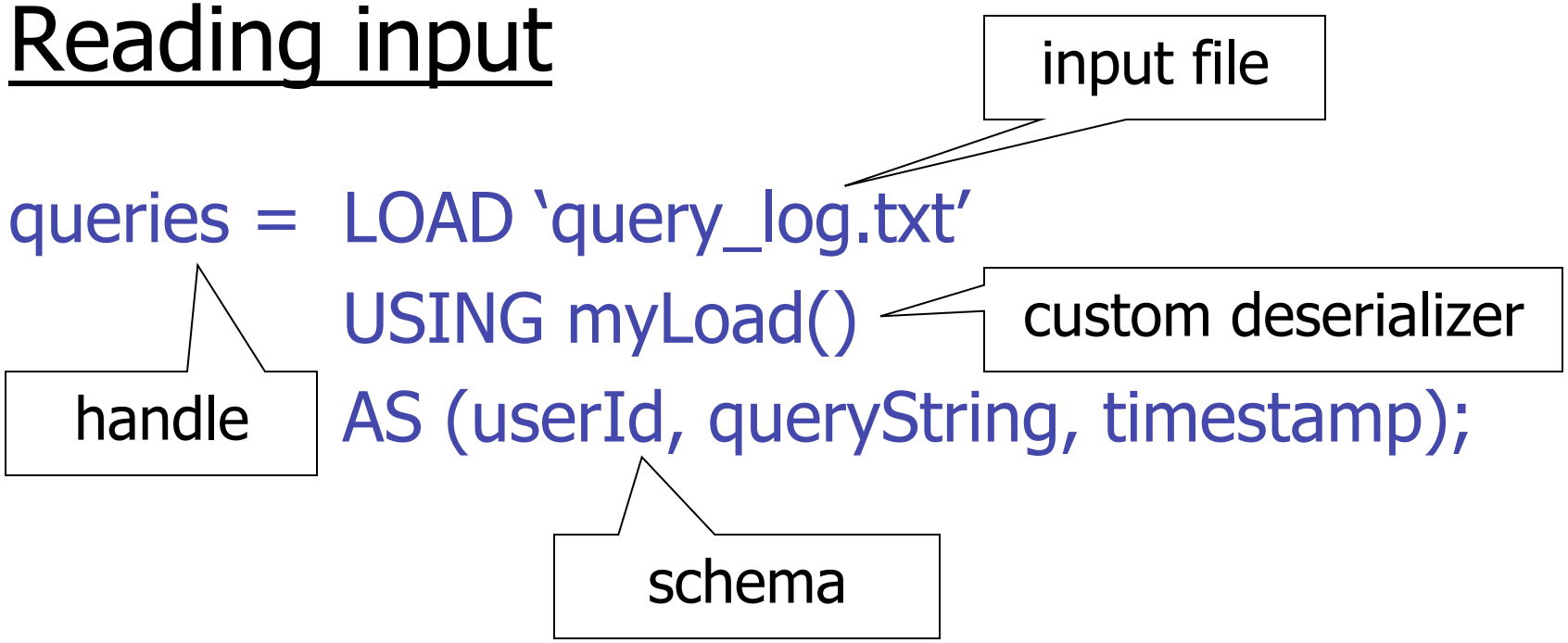
$$\texttt{t} = \left(\text{`alice'}, \left\{ \begin{array}{c} (\text{`lakers'}, 1) \\ (\text{`iPod'}, 2) \end{array} \right\}, [\text{`age'} \rightarrow 20]\right)$$

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`

two tuples!

Reading input



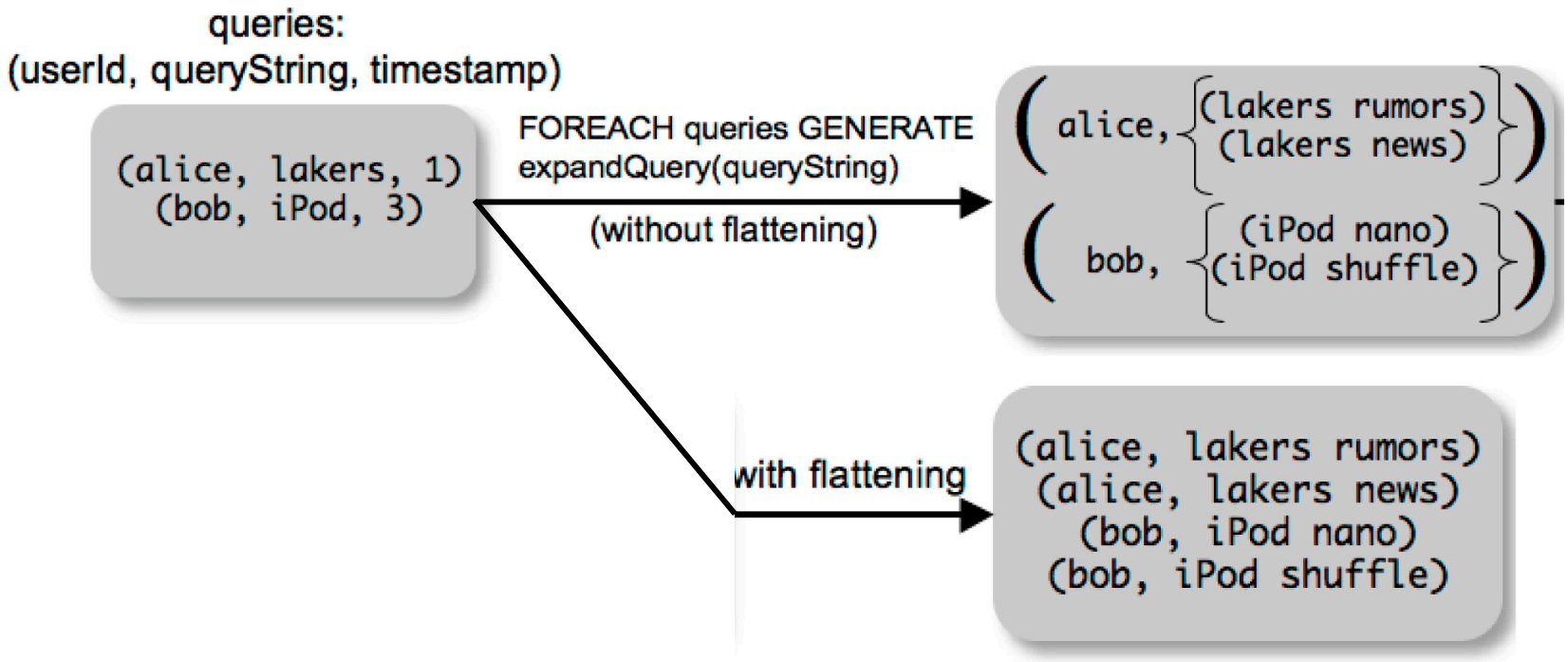
For each

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

- Each tuple is processed independently ← good for parallelism
- Can flatten output to remove one level of nesting:

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

For each


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)

Flattening example

x	*a*	*b*	*c*
	(a1,	{(b1, b2), (b3, b4), (b5)},	{(c1), (c2)})
	(a2,	{(b6, (b7, b8))},	{(c3), (c4)})

y = FOREACH x GENERATE a, FLATTEN(b), c;

Flattening example

x	*a*	*b*	*c*
	a1	{(b1, b2), (b3, b4), (b5)}	{(c1), (c2)}
	a2	{(b6, (b7, b8))}	{(c3), (c4)}

y = FOREACH x GENERATE a, FLATTEN(b), c;

(a1,	b1,	b2,	{(c1), (c2)})
(a1,	b3,	b4,	{(c1), (c2)})
(a1,	b5,	**?**	{(c1), (c2)})
(a2,	b6,	(b7, b8),	{(c3), (c4)})

Flattening example

- Also flattening *c* (in addition to *b*) yields:

  (a1, b1, b2, c1)

  (a1, b1, b2, c2)

  (a1, b3, b4, c1)

  (a1, b3, b4, c2)

  …

Filter

```
realQueries = FILTER queries BY userId NEQ 'bot';

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

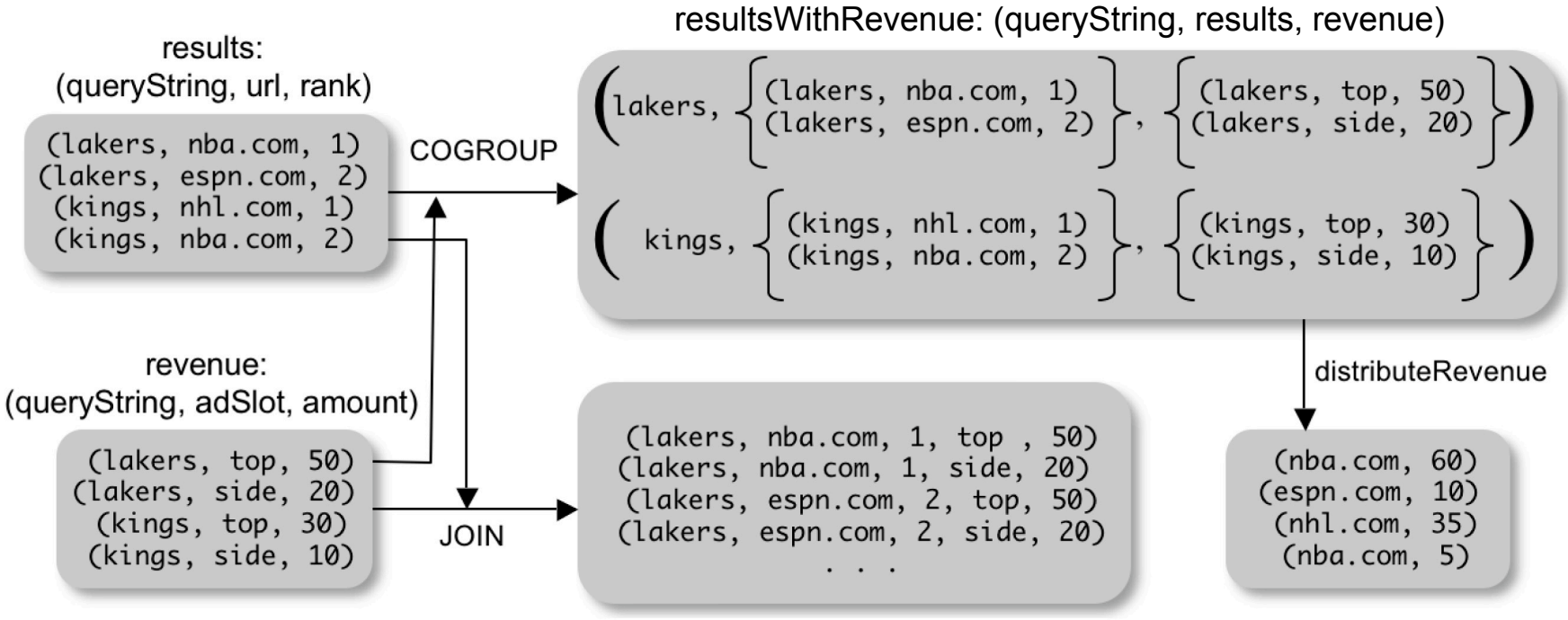
Co-group

- Two input tables:
  - results(queryString, url, position)
  - revenue(queryString, adSlot, amount)

```
resultsWithRevenue =
   COGROUP results BY queryString,
                revenue BY queryString;
revenues = FOREACH resultsWithRevenue GENERATE
   FLATTEN(distributeRevenue(results, revenue));
```

- More flexible than SQL joins

Co-group


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)
COGROUP
JOIN
resultsWithRevenue: (queryString, results, revenue)
(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)})
(lakers, nba.com, 1, top , 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)
. . .
distributeRevenue
(nba.com, 60)
(espn.com, 10)
(nhl.com, 35)
(nba.com, 5)

Group

- Simplified co-group (single input)

```
groupedRevenue = GROUP revenue BY queryString;
queryRevenues = FOREACH groupedRevenue
   GENERATE queryString,
                SUM(revenue.amount) AS total;
```

Co-group example 1

x	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

y	*a*	*b*	*d*
	1	1	d1
	1	2	d2
	2	1	d3
	2	2	d4

s = GROUP x BY a;

Co-group example 1

x

a	*b*	*c*
1	1	c1
1	1	c2
2	2	c3
2	2	c4

y

a	*b*	*d*
1	1	d1
1	2	d2
2	1	d3
2	2	d4

s = GROUP x BY a;

s

a	*x*
1	{(1, 1, c1), (1, 1, c2)}
2	{(2, 2, c3), (2, 2, c4)}

Group and flatten

s = GROUP x BY a;

s	*a*	*x*
	1	{(1, 1, c1), (1, 1, c2)}
	2	{(2, 2, c3), (2, 2, c4)}

z = FOREACH s GENERATE FLATTEN(x);

z	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

Co-group example 2

x	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

y	*a*	*b*	*d*
	1	1	d1
	1	2	d2
	2	1	d3
	2	2	d4

t = GROUP x BY (a, b);

Co-group example 2

x	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

y	*a*	*b*	*d*
	1	1	d1
	1	2	d2
	2	1	d3
	2	2	d4

t = GROUP x BY (a, b);

t	*a/b?*	*x*
	(1, 1)	{(1, 1, c1), (1, 1, c2)}
	(2, 2)	{(2, 2, c3), (2, 2, c4)}

Co-group example 3

x	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

y	*a*	*b*	*d*
	1	1	d1
	1	2	d2
	2	1	d3
	2	2	d4

```
u = COGROUP x BY a, y BY a;
```

Co-group example 3

x	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

y	*a*	*b*	*d*
	1	1	d1
	1	2	d2
	2	1	d3
	2	2	d4

u = COGROUP x BY a, y BY a;

u	*a*	*x*	*y*
	1	{(1, 1, c1), (1, 1, c2)}	{(1, 1, d1), (1, 2, d2)}
	2	{(2, 2, c3), (2, 2, c4)}	{(2, 1, d3), (2, 2, d4)}

Co-group example 4

x	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

y	*a*	*b*	*d*
	1	1	d1
	1	2	d2
	2	1	d3
	2	2	d4

```
v = COGROUP x BY a, y BY b;
```

Co-group example 4

x	*a*	*b*	*c*
	1	1	c1
	1	1	c2
	2	2	c3
	2	2	c4

y	*a*	*b*	*d*
	1	1	d1
	1	2	d2
	2	1	d3
	2	2	d4

v = COGROUP x BY a, y BY b;

v	*a/b?*	*x*	*y*
	1	{(1, 1, c1), (1, 1, c2)}	{(1, 1, d1), (2, 1, d3)}
	2	{(2, 2, c3), (2, 2, c4)}	{(1, 2, d2), (2, 2, d4)}

Join

- Syntax:

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

- Shorthand for:

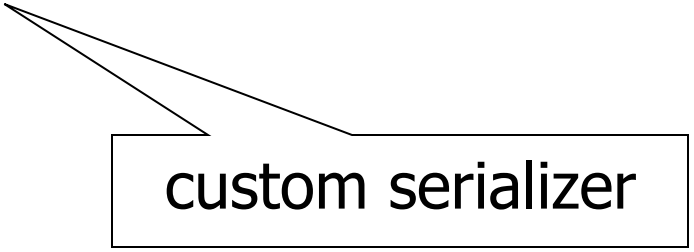
```
temp = COGROUP results BY queryString,
               revenue BY queryString;
joinedResults = FOREACH temp GENERATE
   FLATTEN(results), FLATTEN(revenue);
```

MapReduce in Pig Latin

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

Storing output

```
STORE queryRevenues INTO 'output.txt'
   USING myStore();
```



Pig on Top of MapReduce

- Pig Latin program can be "compiled" into a sequence of mapreductions
- Load, for each, filter: can be implemented as map functions
- Group, store: can be implemented as reduce functions (given proper intermediate data)
- Cogroup and join: special map functions that handle multiple inputs split using the same hash function
- Depending on sequence of operations, include identity mapper and reducer phases as needed

Hive & HiveQL

- Data warehouse on top of Hadoop
  - Hive is the system
  - HiveQL is the language
    - Fully declarative, SQL-like
    - Most SQL features present
    - Supports custom mapreduce scripts
  - MetaStore system catalog
    - Table schemas and statistics
    - Also for keeping track of underlying distributed file structure

HiveQL Features

- Data model: relations with cells containing
  - Atoms
  - Lists, maps, and structs that may be nested
- Table creation

```
CREATE TABLE t(
    x string,
    y list<map<struct<a: string, b:int>>>)
```

sample expression:
t.y[3]['cat'].a

  - Default serializers and deserializes
  - Incorporate "external" data using *SerDe* Java interface

HiveQL Features

- Table updates
  - UPDATE and DELETE not supported
  - INSERT overwrites entire table

    INSERT OVERWRITE t SELECT * FROM s
- Joins
  - Only equality-based SQL joins
    - Equijoin, Cartesian product, left/right/full outer join
  - Explicit syntax

    SELECT t.a AS x, s.b AS y

    FROM t **JOIN** s **ON** (t.a = s.b)

HiveQL Features

- MapReduce scripts
  - Word count example:

```
FROM (
  FROM docs
  MAP text USING 'python wc_map.py' AS (word, count)
  CLUSTER BY word
) temp
REDUCE word, count USING `python wc_reduce.py'
```

  - May have SELECT instead of MAP or REDUCE
    - Order of FROM and SELECT/MAP/REDUCE keywords is interchangeable

Hive Query Processing

1. Parsing: query → abstract syntax tree (AST)
2. Type checking and semantic analysis: AST → query block tree (QBT)
3. Optimization: QBT → optimized operator DAG
   - Map operators: TableScan, Filter, ReduceSink
   - Reduce operators: GroupBy, Join, Select, FileSink
   - Extensible optimization logic (set of heuristics) through *Transform* Java interface
   - No explicit notion of cost or search of plan space
4. Generation of physical plan: operator DAG → Hadoop mapreduce sequence

Optimization heuristics

- **Column pruning**: add projections to remove unneeded columns
- **Partition pruning**: eliminate unneeded file partitions (splits)
- **Predicate pushdown**: early filtering of input records
- **Map-side joins**: hash-join in mapper if one table is small
  - Triggered explicitly by HiveQL hint

    SELECT **/*+ MAPJOIN(t) */** t.a, s.b …

Sawzall

- Procedural scripting language and interpreter on top of MapReduce
- Simplifies the formulation of one mapreduction
  - Record-oriented processing
- Word count example:

```
wc_table: table sum[word: string] of count: int;
most_freq: table top(100) of word: string;

words: array of string = tokenize(input);
for (i: int = 0; i < len(words); i++) {
    emit wc_table[words[i]] ← 1;
    emit most_freq ← words[i];
}
```

Sawzall

- Mapper executes script body for each **input** record
- Reducer generates table outputs by aggregating data **emit**ted by mapper
  - Table (aggregation) types
    - sum, maximum
    - collection: bag of values
    - unique(n): set of values (of max size n for efficiency)
    - sample(n): random sample of size n
    - top(n): n most frequent values
    - quantile(n): quantile thresholds (e.g., percentile for n=100)

Comparison

	Sawzall	Pig	Hive
Language	Procedural	Semi-declarative	Declarative
Schemas	Yes*	Yes (implicit)	Yes (explicit)
Nesting	Containers	Containers	Full
User-defined functions	No	Yes (Java)	Yes
Custom serialization/ deserialization	No	Yes	Yes
Joins	No	Yes+	Yes (equality)
MapReduce steps	Single	Multiple	Multiple

* Through *protocol buffers*, i.e., complex data type declaration

References

- MapReduce: Simplified Data Processing on Large Clusters (Dean and Ghemawat)

  http://labs.google.com/papers/mapreduce.html
- Pig Latin: A Not-so-foreign Language for Data Processing (Olston *et al.*)

  http://wiki.apache.org/pig/
- Hive: A Petabyte Scale Data Warehouse Using Hadoop (Thusoo *et al.*)

  http://i.stanford.edu/~ragho/hive-icde2010.pdf
- Interpreting the Data: Parallel Analysis with Sawzall (Pike *et al.*)

  http://labs.google.com/papers/sawzall.html

Summary

- MapReduce
  - Two phases: map and reduce
  - Transparent distribution, fault tolerance, and scaling
- Sawzall, Pig, and Hive
  - Various layers on top of MapReduce
  - Procedural, semi-declarative, and declarative "query" languages