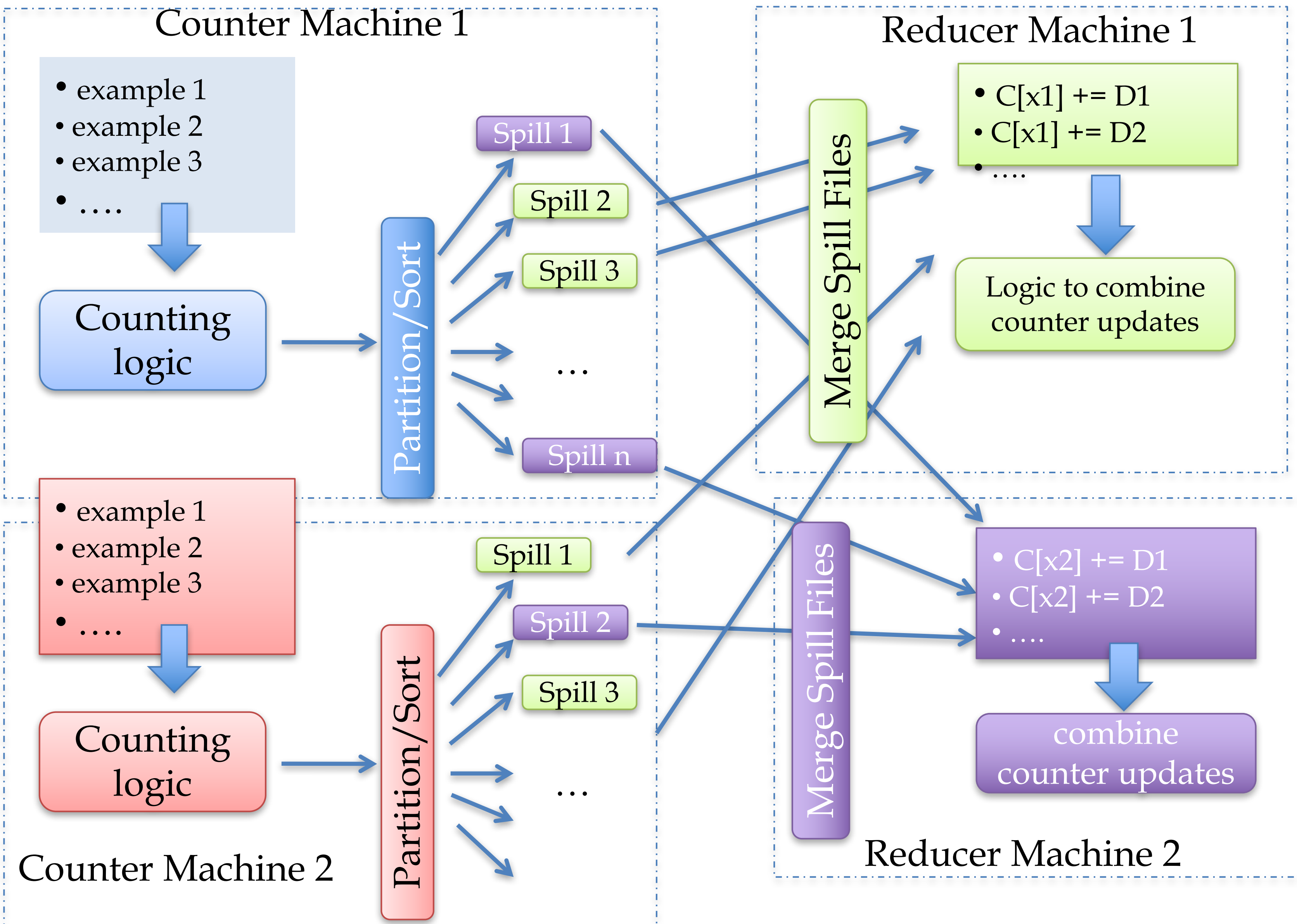
10-605: Map-Reduce Workflows

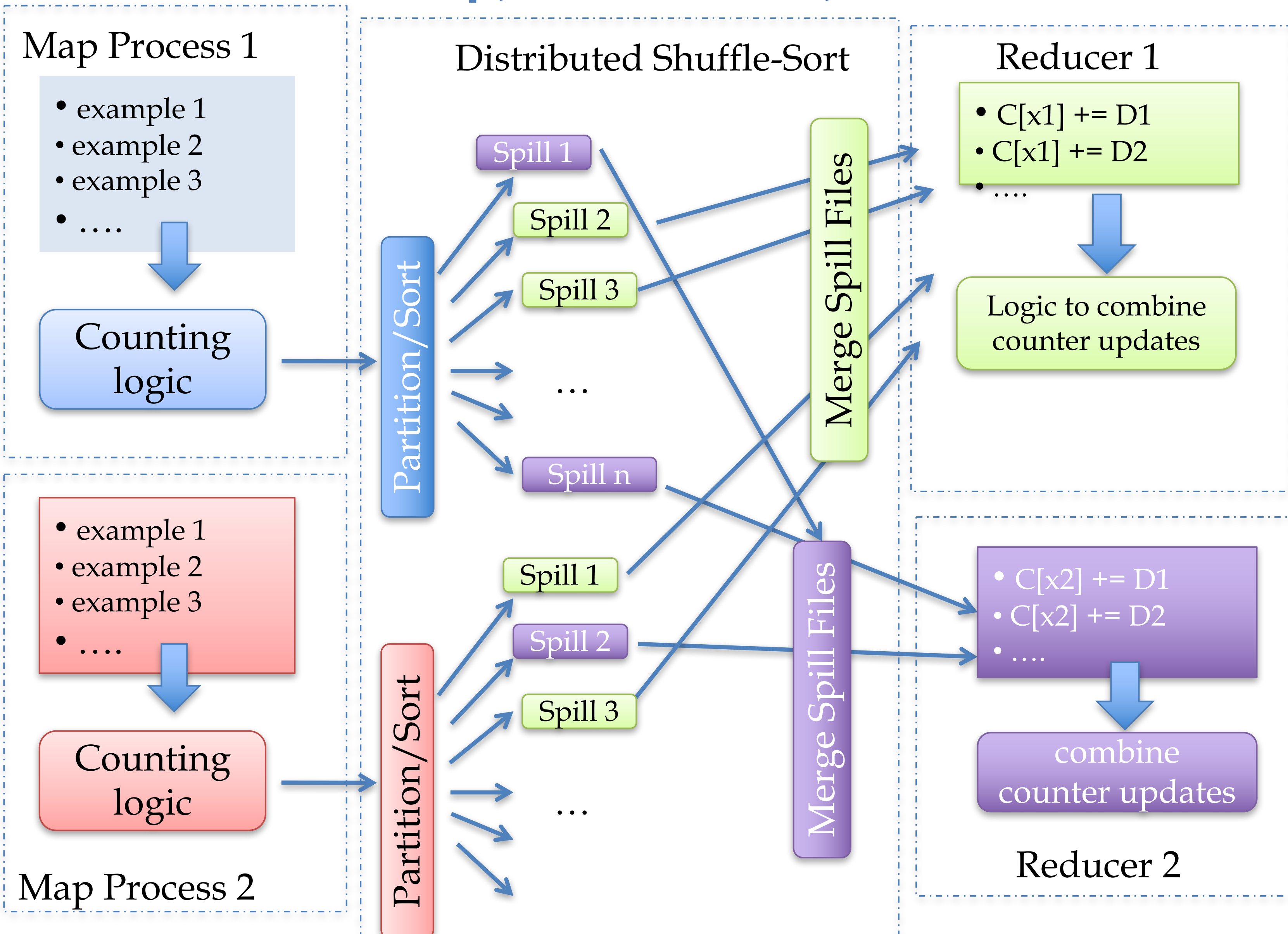
William Cohen

PARALLELIZING STREAM AND SORT

Stream and Sort Counting → Distributed Counting



Distributed Stream-and-Sort: Map, Shuffle-Sort, Reduce



Combiners in Hadoop

Some of this is wasteful

- Remember - moving data around and writing to/reading from disk are very expensive operations
- No reducer can start until:
    - all mappers are done
    - data for its partition has been received
    - data in its partition has been sorted

How much does buffering help?

BUFFER_SIZE	Time	Message Size
none		1.7M *words*
100	47s	1.2M
1,000	42s	1.0M
10,000	30s	0.7M
100,000	16s	0.24M
1,000,000	13s	0.16M
limit		0.05M

Recall idea here: in stream-and-sort, use a buffer to *accumulate* counts in messages for common words *before* the sort so *sort input was smaller*

Combiners

- Sits between the map and the shuffle
  - Do some of the reducing while you're waiting for other stuff to happen
  - Avoid moving all of that data over the network
  - Eg, for wordcount: instead of sending (word,1) send (word,n) where n is a *partial count* (over data seen by that mapper)
    - Reducer still just sums the counts

- Only applicable when
  - order of reduce operations doesn't matter (since order is undetermined)
  - effect is cumulative

```java
  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {

 private final static IntWritable one = new IntWritable(1);
 private Text word = new Text();

 public void map(LongWritable key, Text value, Context context) throws <stuff> {
     String line = value.toString();
     StringTokenizer tokenizer = new StringTokenizer(line);
     while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         context.write(word, one);
     }
 }
}

  public static class Reduce extends Reducer<Text, IntWritable, Text,
  IntWritable> {

 public void reduce(Text key, Iterable<IntWritable> values, Context context)
   throws IOException, InterruptedException {
     int sum = 0;
     for (IntWritable val : values) {
         sum += val.get();
     }
     context.write(key, new IntWritable(sum));
 }
}
```

```java
public static void main(String[] args) throws Exception {

 Configuration conf = new Configuration();

 Job job = new Job(conf, "wordcount");

 job.setMapperClass(Map.class);
 job.setReducerClass(Reduce.class);

 job.setInputFormatClass(TextInputFormat.class);
 job.setOutputFormatClass(TextOutputFormat.class);

 job.setOutputKeyClass(Text.class);

 job.setOutputValueClass(IntWritable.class);

 FileInputFormat.addInputPath(job, new Path(args[0]));
 FileOutputFormat.setOutputPath(job, new Path(args[1]));

 job.waitForCompletion(true);
}
```

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context) throws <stuff> {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
          word.set(tokenizer.nextToken());
          context.wr
      }
  }
}

  public static c
  IntWritable> {

 public void reduc
   throws IOExcept
     int sum = 0;
     for (IntWrita
         sum += va
     }
     context.write
  }
}
```

```
public static void main(String[] args) throws Exception {

 Configuration conf = new Configuration();

 Job job = new Job(conf, "wordcount");

 job.setMapperClass(Map.class);
 job.setReducerClass(Reduce.class);
 job.setCombinerClass(Reduce.class);
 job.setInputFormatClass(TextInputFormat.class);
 job.setOutputFormatClass(TextOutputFormat.class);

 job.setOutputKeyClass(Text.class);

 job.setOutputValueClass(IntWritable.class);

 FileInputFormat.addInputPath(job, new Path(args[0]));
 FileOutputFormat.setOutputPath(job, new Path(args[1]));

 job.waitForCompletion(true);
}
```

Deja vu: Combiner = Reducer

- Often the combiner is the reducer.
  - like for word count
  - but not always

  - remember you have no control over when/whether the combiner is applied

Algorithms in Map-Reduce

(Workflows)

Scalable out-of-core classification (of large test sets)

can we do better
that the current approach?

Testing Large-vocab Naïve Bayes

[For assignment]

- For each example *id, y*, $x_1,\ldots,x_d$ in *train:*
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

collection of event counts is **big**

- Initialize a HashSet NEEDED and a hashtable C
- For each example *id, y*, $x_1,\ldots,x_d$ in *test:*
  – Add $x_1,\ldots,x_d$ to NEEDED

test is **small**

- For each *event, C(event)* in the summed counters
  – If *event* involves a NEEDED term *x* read it into C

- For each example *id, y*, $x_1,\ldots,x_d$ in *test:*
  – For each *y'* in *dom(Y):*
    - Compute log $\Pr(y',x_1,\ldots,x_d)$ = ….

Can we do better?

Test data

id_1 $w_{1,1}\,w_{1,2}\,w_{1,3}$ …. $w_{1,k1}$
id_2 $w_{2,1}\,w_{2,2}\,w_{2,3}$ ….
id_3 $w_{3,1}\,w_{3,2}$ ….
id_4 $w_{4,1}\,w_{4,2}$ …
id_5 $w_{5,1}\,w_{5,2}$ ….
..

Event counts

$X=w_1$^Y=*sports*	5245
$X=w_1$^Y=*worldNews*	1054
X=..	2120
$X=w_2$^Y=…	37
X=…	3
…	…

What we'd like

id_1 $w_{1,1}\,w_{1,2}\,w_{1,3}$ …. $w_{1,k1}$	$C[X=w_{1,1}$^Y=*sports*$]=5245$, $C[X=w_{1,1}$^$Y=..]$,$C[X=w_{1,2}$^…]
id_2 $w_{2,1}\,w_{2,2}\,w_{2,3}$ ….	$C[X=w_{2,1}$^$Y=….]=1054$,…, $C[X=w_{2,k2}$^…]
id_3 $w_{3,1}\,w_{3,2}$ ….	$C[X=w_{3,1}$^$Y=….]=…$
id_4 $w_{4,1}\,w_{4,2}$ …	…

Can we do better?

Step 1: group counters by word *w*

How:

- Stream and sort:
  - for each C[X=w^Y=y]=n
    - print "w C[Y=y]=n"
  - sort and build a *list* of values associated with each key *w*

    Like an inverted index

Event counts

$X=w_1$^$Y=sports$	*5245*
$X=w_1$^$Y=worldNews$	*1054*
X=..	*2120*
$X=w_2$^$Y=\ldots$	*37*
X=…	*3*
…	…

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worldNews]=564
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldNews]=4464

If these records were in a key-value DB we would know what to do….

Test data

id_1 $w_{1,1}\, w_{1,2}\, w_{1,3}$ …. $w_{1,k1}$
id_2 $w_{2,1}\, w_{2,2}\, w_{2,3}$ ….
id_3 $w_{3,1}\, w_{3,2}$ ….
id_4 $w_{4,1}\, w_{4,2}$ …
id_5 $w_{5,1}\, w_{5,2}$ ….
..

Record of all event counts for each word

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worl
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldN

Classification logic

Step 2: stream through and for each test case

id_i $w_{i,1}\, w_{i,2}\, w_{i,3}$ …. $w_{i,ki}$

request the event counters needed to classify id_i from the event-count DB, then classify using the answers

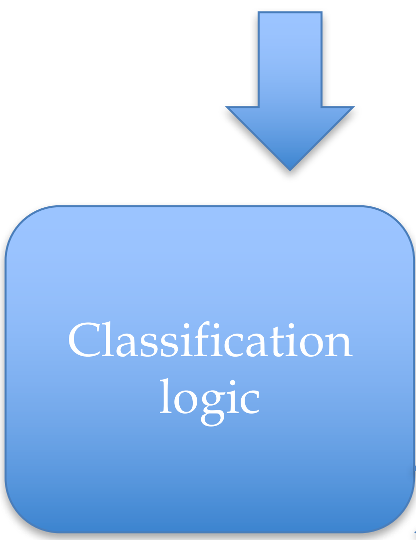
Is there a stream-and-sort analog of this request-and-answer pattern?

Test data

$id_1\ \ w_{1,1}\, w_{1,2}\, w_{1,3}\ \ldots.\ w_{1,k1}$
$id_2\ \ w_{2,1}\, w_{2,2}\, w_{2,3}\ \ldots.$
$id_3\ \ w_{3,1}\, w_{3,2}\ \ \ldots.$
$id_4\ \ w_{4,1}\, w_{4,2}\ \ldots$
$id_5\ \ w_{5,1}\, w_{5,2}\ \ldots.$
..

Record of all event counts for each word

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worl
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldN

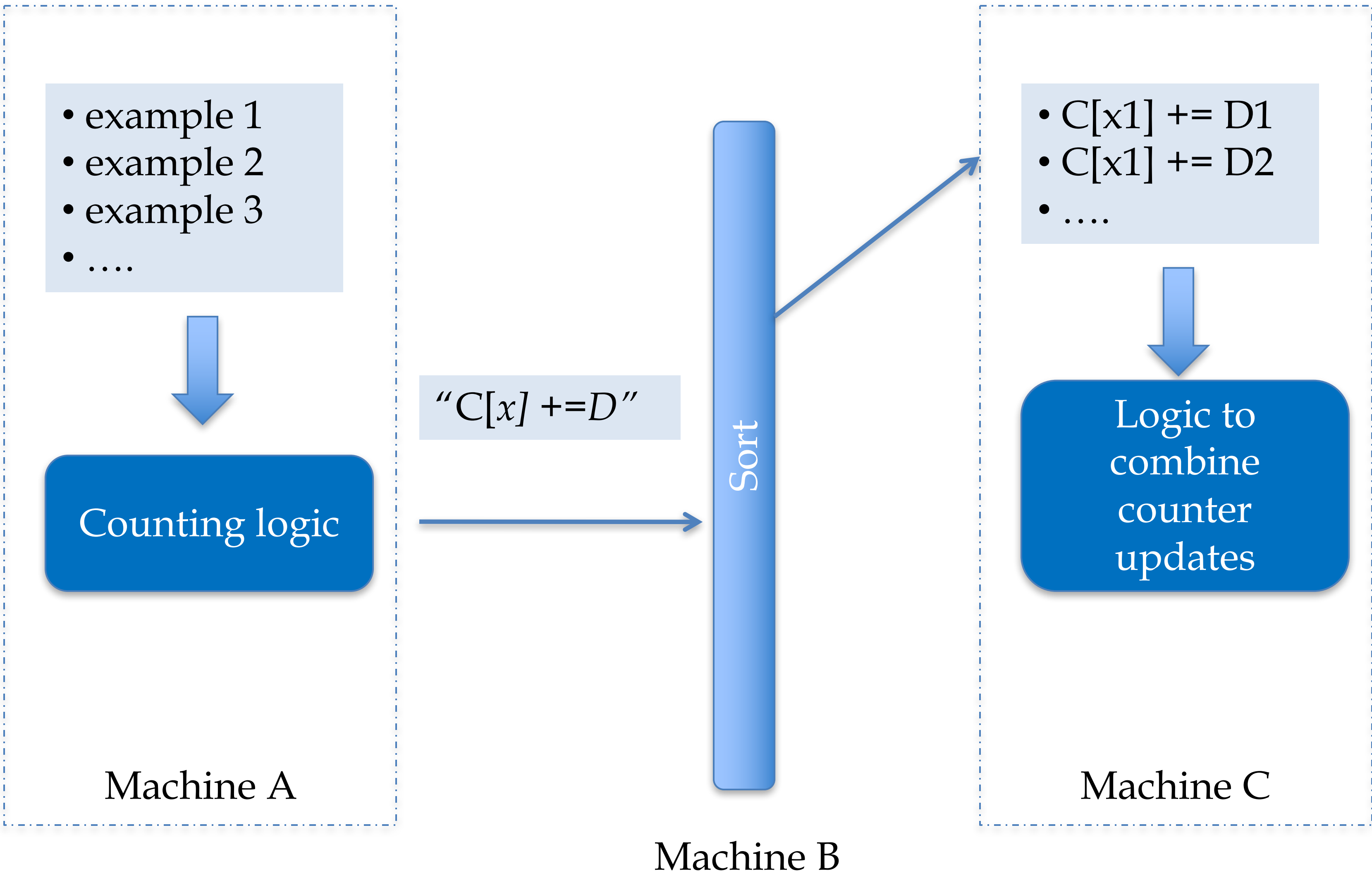


Step 2: stream through and for each test case

$id_i\ \ w_{i,1}\, w_{i,2}\, w_{i,3}\ \ldots.\ w_{i,ki}$

request the event counters needed to classify id_i from the event-count DB, then classify using the **answers**

Recall: Stream and Sort Counting: sort messages so the recipient can stream through them



Is there a stream-and-sort analog of this request-and-answer pattern?

Test data

id_1 $w_{1,1}\, w_{1,2}\, w_{1,3}$ …. $w_{1,k1}$
id_2 $w_{2,1}\, w_{2,2}\, w_{2,3}$ ….
id_3 $w_{3,1}\, w_{3,2}$ ….
id_4 $w_{4,1}\, w_{4,2}$ …
id_5 $w_{5,1}\, w_{5,2}$ ….
..

Record of all event counts for each word

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worl
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldN

Classification logic

$W_{1,1}$ counters to id_1
$W_{1,2}$ counters to id_1
…
$W_{i,j}$ counters to id_i
…

Is there a stream-and-sort analog of this request-and-answer pattern?

Test data

id$_1$ *found an aarvark in zynga's farmville today!*
id$_2$ …
id$_3$ ….
id$_4$ …
id$_5$ …
..

Record of all event counts for each word

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worl
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldN

Classification logic

found ctrs to *id*$_1$
aardvark ctrs to *id*$_1$
…
today ctrs to *id*$_1$
…

Is there a stream-and-sort analog of this request-and-answer pattern?

Test data

id_1 *found an aarvark in zynga's farmville today!*
id_2 …
id_3 ….
id_4 …
id_5 …
..

Record of all event counts for each word

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worl
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldN

Classification logic

found ~ctrs to id_1
aardvark ~ctrs to id_1
…
today ~ctrs to id_1
…

~ is the last ascii character

% export LC_COLLATE=C

means that it will sort *after* anything else with unix sort

Is there a stream-and-sort analog of this request-and-answer pattern?

Test data

id_1 found an aardvark in zynga's farmville today!
id_2 …
id_3 ….
id_4 …
id_5 …
..

Record of all event counts for each word

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worl
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldN

Counter records

Classification logic

found ~ctr to id_1
aardvark ~ctr to id_2
…
today ~ctr to id_i
…

requests

Combine and sort

A stream-and-sort analog of the request-and-answer pattern…

Record of all event counts for each word

w	Counts
aardvark	C[w^Y=sports]=2
agent	…
…	
zynga	…

Counter records

found ~ctr to id_1
aardvark ~ctr to id_1
…
today ~ctr to id_1
…

requests

Combine and sort

w	Counts
aardvark	C[w^Y=sports]=2
aardvark	~ctr to id1
agent	C[w^Y=sports]=…
agent	~ctr to id345
agent	~ctr to id9854
…	~ctr to id345
agent	~ctr to id34742
…	
zynga	C[…]
zynga	~ctr to id1

Request-handling logic

A stream-and-sort analog of the request-and-answer pattern…

- previousKey = somethingImpossible
- For each (*key*,*val*) in input:
  - If *key*==previousKey
    - Answer(recordForPrevKey,*val)*
  - Else
    - previousKey = *key*
    - recordForPrevKey = *val*

define Answer*(record,request):*

- find *id* where "*request* = ~ctr to *id*"
- print "*id* ~ctr for *request* is *record*"

w	Counts
aardvark	C[w^Y=sports]=2
aardvark	~ctr to id1
agent	C[w^Y=sports]=…
agent	~ctr to id345
agent	~ctr to id9854
…	~ctr to id345
agent	~ctr to id34742
…	
zynga	C[…]
zynga	~ctr to id1

Combine and sort

requests

Request-handling logic

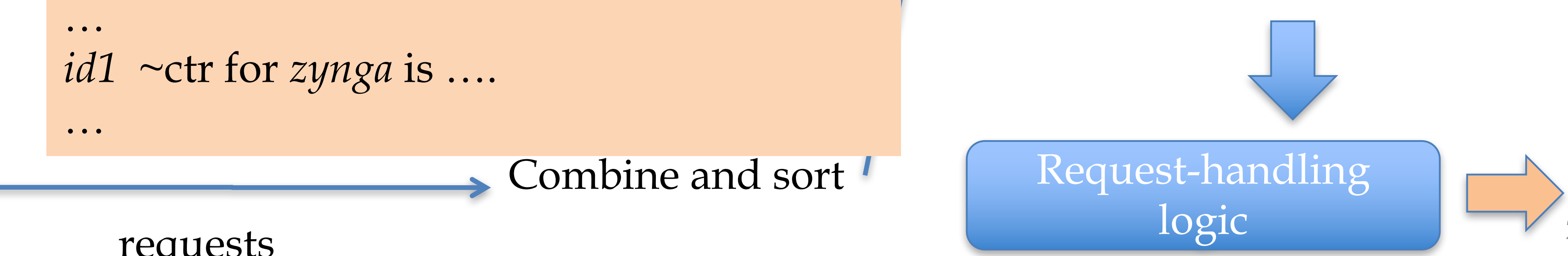
A stream-and-sort analog of the request-and-answer pattern…

- previousKey = somethingImpossible
- For each (*key*,*val*) in input:
  - If *key*==previousKey
    - Answer(recordForPrevKey,*val)*
  - Else
    - previousKey = *key*
    - recordForPrevKey = *val*

define Answer*(record,request):*
- find *id* where "*request* = ~ctr to *id*"
- print "*id* ~ctr for *request* is *record*"

Output:
id1 ~ctr for *aardvark* is C[w^Y=sports]=2
…
id1 ~ctr for *zynga* is ….
…

w	Counts
aardvark	C[w^Y=sports]=2
aardvark	~ctr to id1
agent	C[w^Y=sports]=…
agent	~ctr to id345
agent	~ctr to id9854
…	~ctr to id345
agent	~ctr to id34742
…	
zynga	C[…]
zynga	~ctr to id1



A stream-and-sort analog of the request-and-answer pattern…

w	Counts
aardvark	C[w^Y=sports]=2
aardvark	~ctr to id1
agent	C[w^Y=sports]=…
agent	~ctr to id345
agent	~ctr to id9854
…	~ctr to id345
agent	~ctr to id34742
…	
zynga	C[…]
zynga	~ctr to id1

Output:
id1 ~ctr for *aardvark* is C[w^Y=sports]=2
…
id1 ~ctr for *zynga* is ….
…

id_1 *found an aardvark in zynga's farmville today!*
id_2 …
id_3 ….
id_4 …
id_5 …
..

Request-handling logic

Combine and sort

????

What we'd wanted

$id_1\ w_{1,1}\, w_{1,2}\, w_{1,3}\ ....\ w_{1,k1}$	$C[X{=}w_{1,1}\wedge Y{=}sports]{=}5245,\ C[X{=}w_{1,1}\wedge Y{=}..],C[X{=}w_{1,2}\wedge\ldots]$
$id_2\ w_{2,1}\, w_{2,2}\, w_{2,3}\ ....$	$C[X{=}w_{2,1}\wedge Y{=}....]{=}1054,\ldots,\ C[X{=}w_{2,k2}\wedge\ldots]$
$id_3\ w_{3,1}\, w_{3,2}\ ....$	$C[X{=}w_{3,1}\wedge Y{=}....]{=}\ldots$
$id_4\ w_{4,1}\, w_{4,2}\ \ldots$	…

What we ended up with … and it's good enough!

Key	**Value**
id1	*found aardvark zynga farmville today*
	~ctr for *aardvark* is C[w^Y=sports]=2
	~ctr for *found* is C[w^Y=sports]=1027,C[w^Y=worldNews]=564
	…
id2	$w_{2,1}\, w_{2,2}\, w_{2,3}\ ....$
	~ctr for $w_{2,1}$ is …
…	…

Implementation summary

```
java CountForNB train.dat … > eventCounts.dat
java CountsByWord eventCounts.dat | sort
 | java CollectRecords   > words.dat

java requestWordCounts  test.dat
 | cat - words.dat | sort | java answerWordCountRequests
 | cat - test.dat| sort | testNBUsingRequests
```

train.dat

id_1 $w_{1,1}$ $w_{1,2}$ $w_{1,3}$ …. $w_{1,k1}$
id_2 $w_{2,1}$ $w_{2,2}$ $w_{2,3}$ ….
id_3 $w_{3,1}$ $w_{3,2}$ ….
id_4 $w_{4,1}$ $w_{4,2}$ …
id_5 $w_{5,1}$ $w_{5,2}$ ….
..

counts.dat

X=w1^Y=sports	*5245*
X=w1^Y=worldNews	*1054*
X=..	*2120*
X=w2^Y=…	*37*
X=…	*3*
…	…

Implementation summary

```
java CountForNB train.dat … > eventCounts.dat
java CountsByWord eventCounts.dat | sort
 | java CollectRecords   > words.dat

java requestWordCounts  test.dat
 | cat - words.dat | sort | java answerWordCountRequests
 | cat - test.dat| sort | testNBUsingRequests
```

words.dat

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worldNews]=564
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldNews]=4464

Implementation summary

java CountForNB train.dat … > eventCounts.dat
java CountsByWord eventCounts.dat | sort
| java CollectRecords > words.dat

java requestWordCounts test.dat
| cat - **words.dat** | sort | java answerWordCountRequests
| cat - test.dat| sort | testNBUsingRequests

output looks like this

found ~ctr to id_1
aardvark ~ctr to id_2
…
today ~ctr to id_i
…

words.dat

w	Counts
aardvark	C[w^Y=sports]=2
agent	…
…	
zynga	…

input looks like this

w	Counts
aardvark	C[w^Y=sports]=2
aardvark	~ctr to id1
agent	C[w^Y=sports]=…
agent	~ctr to id345
agent	~ctr to id9854
…	~ctr to id345

Implementation summary

java CountForNB train.dat … > eventCounts.dat
java CountsByWord eventCounts.dat | sort
| java CollectRecords > words.dat

java requestWordCounts test.dat
| cat - **words.dat** | sort | java answerWordCountRequests
| cat - **test.dat** | sort | testNBUsingRequests

Output looks like this

Output:
id1 ~ctr for *aardvark* is C[w^Y=sports]=2
…
id1 ~ctr for *zynga* is ….
…

test.dat

id_1 found an aardvark in zynga's farmville today!
id_2 …
id_3 ….
id_4 …
id_5 …
..

Implementation summary

```
java CountForNB train.dat … > eventCounts.dat
java CountsByWord eventCounts.dat | sort
 | java CollectRecords   > words.dat

java requestWordCounts  test.dat
 | cat - words.dat | sort | java answerWordCountRequests
 | cat -test.dat | sort | testNBUsingRequests
```

Input looks like this

Key	Value
id1	found aardvark zynga farmville today
	~ctr for aardvark is C[w^Y=sports]=2
	~ctr for found is C[w^Y=sports]=1027,C[w^Y=worldNews]=564
	…
id2	$w_{2,1}$ $w_{2,2}$ $w_{2,3}$ ….
	~ctr for $w_{2,1}$ is …
…	…

ABSTRACTIONS FOR MAP-REDUCE

Abstractions On Top Of Map-Reduce

- Some obvious streaming processes:
  - for each row in a table
    - Transform it and output the result
    - Decide if you want to keep it with some boolean test, and copy out only the ones that pass the test

Example: stem words in a stream of word-count pairs:

("aardvarks",1) ➔ ("aardvark",1)

Proposed syntax: *f(row)* ➔ *row'*

$table2 = \text{MAP } table1 \text{ TO } \lambda\, row : f(row))$

Example: apply stop words

("aardvark",1) ➔ ("aardvark",1)
("the",1) ➔ *deleted*

Proposed syntax: *f(row)* ➔ *{true,false}*

$table2 = \text{FILTER } table1 \text{ BY } \lambda\, row : f(row))$

Abstractions On Top Of Map-Reduce

- A non-obvious? streaming processes:
  - for each row in a table
    - Transform it to a list of items
    - Splice all the lists together to get the output table (**flatten**)

Proposed syntax: *f(row)* ➔ *list of rows*

table2 = FLATMAP *table1* TO λ *row* : *f*(*row*))

Example: tokenizing a line

"I found an aardvark" ➔ ["i", "found","an","aardvark"]
"We love zymurgy" ➔ ["we","love","zymurgy"]

..but final table is one word per row

"i"
"found"
"an"
"aardvark"
"we"
"love"
...

NB Test Step

How:

- Stream and sort:
  - for each C[X=w^Y=y]=n
    - print "w C[Y=y]=n"
  - sort and build a *list* of values associated with each key *w*
    Like an inverted index

Event counts

$X\!=\!w_1\text{^}Y\!=\!sports$	*5245*
$X\!=\!w_1\text{^}Y\!=\!worldNews$	*1054*
X=..	*2120*
$X\!=\!w_2\text{^}Y\!=\!\ldots$	*37*
X=…	*3*
…	…

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worldNews]=564
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldNews]=4464

NB Test Step

<u>The general case:</u>
We're taking rows from a table

- In a particular format *(event,count)*

Applying a function to get a new value

- The *word* for the event

And *grouping* the rows of the table by this new value

→ **Grouping operation**
Special case of a map-reduce

Event counts

$X=w_1$^$Y=sports$	*5245*
$X=w_1$^$Y=worldNews$	*1054*
X=..	*2120*
$X=w_2$^$Y=\ldots$	*37*
X=…	*3*
…	…

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worldNews]=564
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldNews]=4464

Proposed syntax: *f(row)→field*

GROUP *table* BY λ *row* : *f*(*row*)

Could define *f* via: a function, a field of a defined *record* structure, …

NB Test Step

<u>The general case:</u>
We're taking rows from a table
- In a particular format *(event,count)*

Applying a function to get a new value
- The *word* for the event

And *grouping* the rows of the table by this new value

➔ **Grouping operation**
Special case of a map-reduce

Proposed syntax: *f(row)* ➔ *field*

GROUP *table* BY λ *row* : *f*(*row*)

Could define *f* via: a function, a field of a defined *record* structure, …

Aside: you guys know how to implement this, right?

1. Output pairs (f(row),row) with a map/streaming process
2. Sort pairs by key – which is f(row)
3. Reduce and aggregate by *appending together* all the values associated with the same key

Abstractions On Top Of Map-Reduce

- And another example from the Naïve Bayes test program…

Request-and-answer

Test data

id_1 $w_{1,1}\, w_{1,2}\, w_{1,3}$ …. $w_{1,k1}$
id_2 $w_{2,1}\, w_{2,2}\, w_{2,3}$ ….
id_3 $w_{3,1}\, w_{3,2}$ ….
id_4 $w_{4,1}\, w_{4,2}$ …
id_5 $w_{5,1}\, w_{5,2}$ ….
..

Record of all event counts for each word

w	Counts associated with W
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worl
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldN

Classification logic

Step 2: stream through and for each test case

id_i $w_{i,1}\, w_{i,2}\, w_{i,3}$ …. $w_{i,ki}$

request the event counters needed to classify id_i from the event-count DB, then classify using the answers

Request-and-answer

- Break down into stages
  - Generate the data being requested (indexed by key, here a word)
    - Eg with group … by
  - Generate the requests as (key, requestor) pairs
    - Eg with flatmap … to
  - **Join** these two tables by key
    - Join: conceptually defined as (1) cross-product and (2) filter out pairs with different values for keys
    - Join: implemented by concatenating two different tables of key-value pairs, and reducing them together
  - Postprocess the joined result

w	Request
found	~ctr to id1
aardvark	~ctr to id1
…	
zynga	~ctr to id1
…	~ctr to id2

w	Counters
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worldNews
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldNews]=

w	Counters	Requests
aardvark	C[w^Y=sports]=2	~ctr to id1
agent	C[w^Y=sports]=…	~ctr to id345
agent	C[w^Y=sports]=…	~ctr to id9854
agent	C[w^Y=sports]=…	~ctr to id345
…	C[w^Y=sports]=…	~ctr to id34742
zynga	C[…]	~ctr to id1
zynga	C[…]	…

w	Request
found	id1
aardvark	id1
…	
zynga	id1
…	id2

w	Counters
aardvark	C[w^Y=sports]=2
agent	C[w^Y=sports]=1027,C[w^Y=worldNews
…	…
zynga	C[w^Y=sports]=21,C[w^Y=worldNews]=

Examples:

JOIN *wordInDoc* BY *word, wordCounters* BY *word --- if word(row) defined correctly*

JOIN wordInDoc BY lambda (word,docid):word,
wordCounters BY lambda (word,counters):word – *using python syntax for functions*

Proposed syntax:

JOIN *table1* BY $\lambda\, row : f(row)$,
table2 BY $\lambda\, row : g(row)$

	C[w^Y=sports]=…	id345
	C[w^Y=sports]=…	id34742
	C[…]	id1
	C[…]	…

Abstract Implementation: [TF]IDF

1/2

data = *pairs (docid, term) where term is a word appears in document with id docid*
opera
• DIS
• GRO
ce st

key	value
found	~~(d123,found),(d134,found),~~… 2456
aardvark	~~(d123,aardvark),~~… 7

docId	term
d123	found
d123	aardvark

docFreq = DISTINCT data
| GROUP BY λ(docid,term):term REDUCING T

key	value
1	12451

docIds = MAP DATA BY=λ(docid,term):docid | DISTINCT
numDocs = GROUP docIds BY λdocid:1 REDUCING TO count /* *(1,numDocs)* */

question – how many reducers should I use here?

dataPlusDF =
JOIN data BY λ(docid, term):term, docFreq BY λ(term, df):term
| MAP λ((docid,term),(term,df)):(docId,term,df) /* *(docId,term,document-freq)* */

unnormalizedDocVecs = JOIN dataPlusDF by λrow:1, numDocs by λrow:1
| MAP λ((docId,term,df),(dummy,numDocs)): (docId,term,log(numDocs/df))
/* *(docId, term, weight-before-normalizing) : **u*** */

Abstract Implementation: [TF]IDF

data = pairs (docid, term) where term is a word appears in document with id docid

opera

• DIS

• GR

ce st

key	value
found	~~(d123,found),(d134,found),~~… 2456
aardvark	~~(d123,aardvark),~~… 7

docId	term
d123	found
d123	aardvark

docFreq = DISTINCT data
| GROUP BY λ(docid,term):term REDUCING T

key	value
1	12451

docIds = MAP DATA BY=λ(docid,term):docid | DISTINCT
numDocs = GROUP docIds BY λdocid:1 REDUCING TO count /* *(1,numDocs)* */

question – how many reducers should I use here?

dataPlusDF =
JOIN data BY λ(docid, term):term, docFreq BY λ(term, df):term
| MAP λ((docid,term),(term,df)):(docId,term,df) /* *(docId,term,document-freq)* */

unnormalizedDocVecs = JOIN dataPlusDF by λrow:1, numDocs by λrow:1
| MAP λ((docId,term,df),(dummy,numDocs)): (docId,term,log(numDocs/df))
/* *(docId, term, weight-before-normalizing) : u* */

question – how many reducers should I use here?

Abstract Implementation: TFIDF

2/2

normalizers =
  GROUP unnormalizedDocVecs BY λ(docId,term,w):docid
  RETAINING λ(docId,term,w): w^2
  REDUCING TO sum */* (docid,sum-of-square-weights) */*

key	
d1234	(d1234,found,1.542), (d1234,aardvark,13.23),… 37.234
d3214	…. 29.654

docVec = JOIN unnormalizedDocVecs BY λ(docId,term,w):docid,
    normalizers BY λ(docId,norm):docid
  | MAP λ((docId,term,w), (docId,norm)): (docId,term,w/sqrt(norm))
  / (docId, term, weight) */*

docId	term	w
d1234	found	1.542
d1234	aardvark	13.23

docId	w
d1234	37.234
d1234	37.234

Two ways to join

- Reduce-side join
- Map-side join

Two ways to join

- Reduce-side join for A,B

A

term	df
found	2456
aardvark	7
…	

B

term	docId
aardvark	d15
…	…
found	d7
found	d23
found	…
…	

concat and sort →

do the join →

A	B
(aardvark, 7)	(aardvark, d15)
…	…
(found,2456)	(found,d7)
(found,2456)	(found,d23)
…	…

Two ways to join

- Reduce-side join for A,B

	df
	2456
ark	7

	docId
ark	d15
	...
	d7
	d23
	…

concat and sort

term		df
found	A	2456
aardvark	A	7
…		

term		docId
aardvark	B	d15
…		**...**
found	B	d7
found	B	d23
found	B	…
…		

tricky bit: need **sort** by **first two** values (aardvark, *AB*) – we want the DF's to come first

but all tuples with key "aardvark" should go to **same worker**

do the join

A	B
(aardvark, 7)	(aardvark, d15)
…	...
(found,2456)	(found,d7)
(found,2456)	(found,d23)
…	…

Two ways to join

- Reduce-side join for A,B

	df
	2456
ark	7

	docId
ark	d15
	…
	d7
	d23
	…

concat
and sort

term		df
found	A	2456
aardvark	A	7
…		

term		docId
aardvark	B	d15
…		…
found	B	d7
found	B	d23
found	B	…
…		

tricky bit: need **sort** by **first two** values (aardvark, *AB*) – we want the DF's to come first

but all tuples with key "aardvark" should go to **same worker**

custom sort (secondary sort key): Writeable with your own Comparator

custom Partitioner (specified for job like the Mapper, Reducer, ..)

Two ways to join

- Map-side join
  - write the smaller relation out to disk
  - send it to each Map worker
    - DistributedCache
  - when you **initialize** each Mapper, load in the small relation
    - Configure(…) is called at initialization time
  - map through the larger relation and do the join
  - faster but requires one relation to go in memory

Two ways to join

- Map-side join for A (small) and B (large)

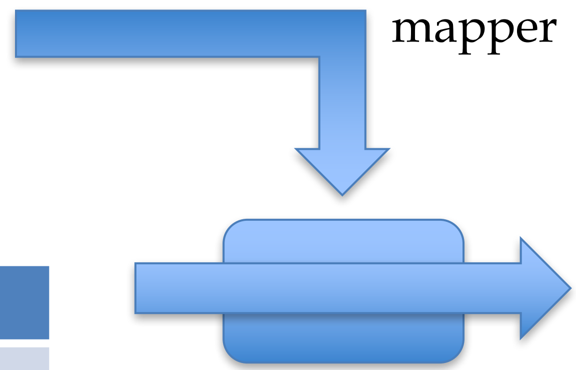
A

term	df
found	2456
aardvark	7
…	

B

term	docId
aardvark	d15
…	…
found	d7
found	d23
found	…
…	

Load into mapper



Join as you map

A	B
(aardvark, 7)	(aardvark, d15)
…	…
(found,2456)	(found,d7)
(found,2456)	(found,d23)
…	…

Two ways to join

- Map-side join for A (small) and B (large)

A

term	df
found	2456
aardvark	7
…	

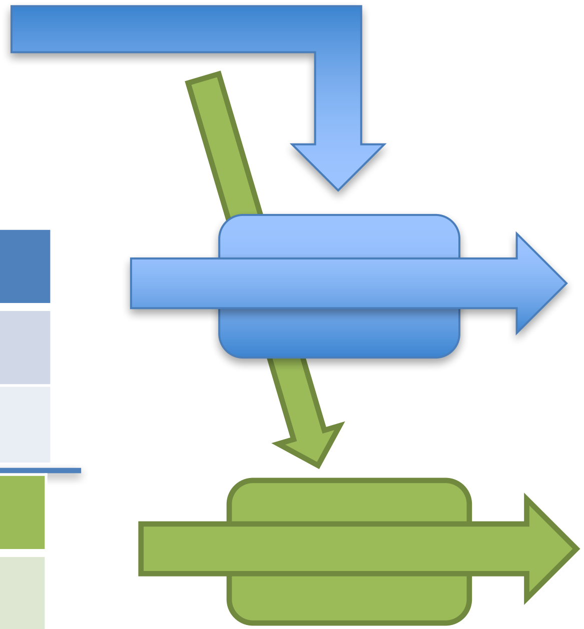
Duplicate and load into **every** mapper

B1

term	docId
aardvark	d15
…	…

B2

found	**d7**
found	d23
found	…
…	



Join as you map

A	B
(aardvark, 7)	(aardvark, d15)
…	…

(found,2456)	**(found,d7)**
(found,2456)	(found,d23)
…	…

PIG: A WORKFLOW LANGUAGE

PIG: word count example

```
A = load '/tmp/bible+shakes.nopunc';
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;
C = filter B by word matches '\w+';
D = group C by word;
E = foreach D generate COUNT(C) as count, group as word;
F = order E by count desc;
store F into '/tmp/wc';
```

PIG program is a bunch of **assignments** where every LHS is a **relation**.
No loops, conditionals, etc allowed.

```
A = load '/tmp/bible+shakes.nopunc';
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;
C = filter B by word matches '\w+';
D = group C by word;
E = foreach D generate COUNT(C) as count, group as word;
F = order E by count desc;
store F into '/tmp/wc';
```

Tokenize – built-in functio

Built-in regex matching ….

Flatten – special keyword, which applies to the **next** step in the process – so output is a stream of words w/o document boundaries

```
A = load '/tmp/bible+shakes.nopunc';
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;
C = filter B by word matches '\w+';
D = group C by word;
E = foreach D generate COUNT(C) as count, group as word;
F = order E by count desc;
store F into '/tmp/wc';
```

Group produces a stream of **bags** of identical words… bags, tuples, ictionaries are primitive types

Group by … foreach generate count(…) will be optimized into a single map-reduce