Lecture 09: Parallel Databases, Big Data, Map/Reduce, Pig-Latin

Wednesday, November 23rd, 2011

Overview of Today's Lecture

- Parallel databases
  - Chapter 22.1 – 22.5
- Big Data
  - Kumar et al. *The Web as a Graph*
- Cluster Computing
  - Map/reduce: Rajaraman&Ullman online book
  - Declarative layer: Pig-Latin, Tenzing (see papers)
- WILL NOT DISUCSS IN CLASS: mini-tutorial on Pig-Latin (see the last part of the slides)

Parallel Databases

Parallel v.s. Distributed Databases

- Parallel database system:
  - Improve performance through parallel implementation
  - Will discuss in class

- Distributed database system:
  - Data is stored across several sites, each site managed by a DBMS capable of running independently
  - Will not discuss in class

Parallel DBMSs

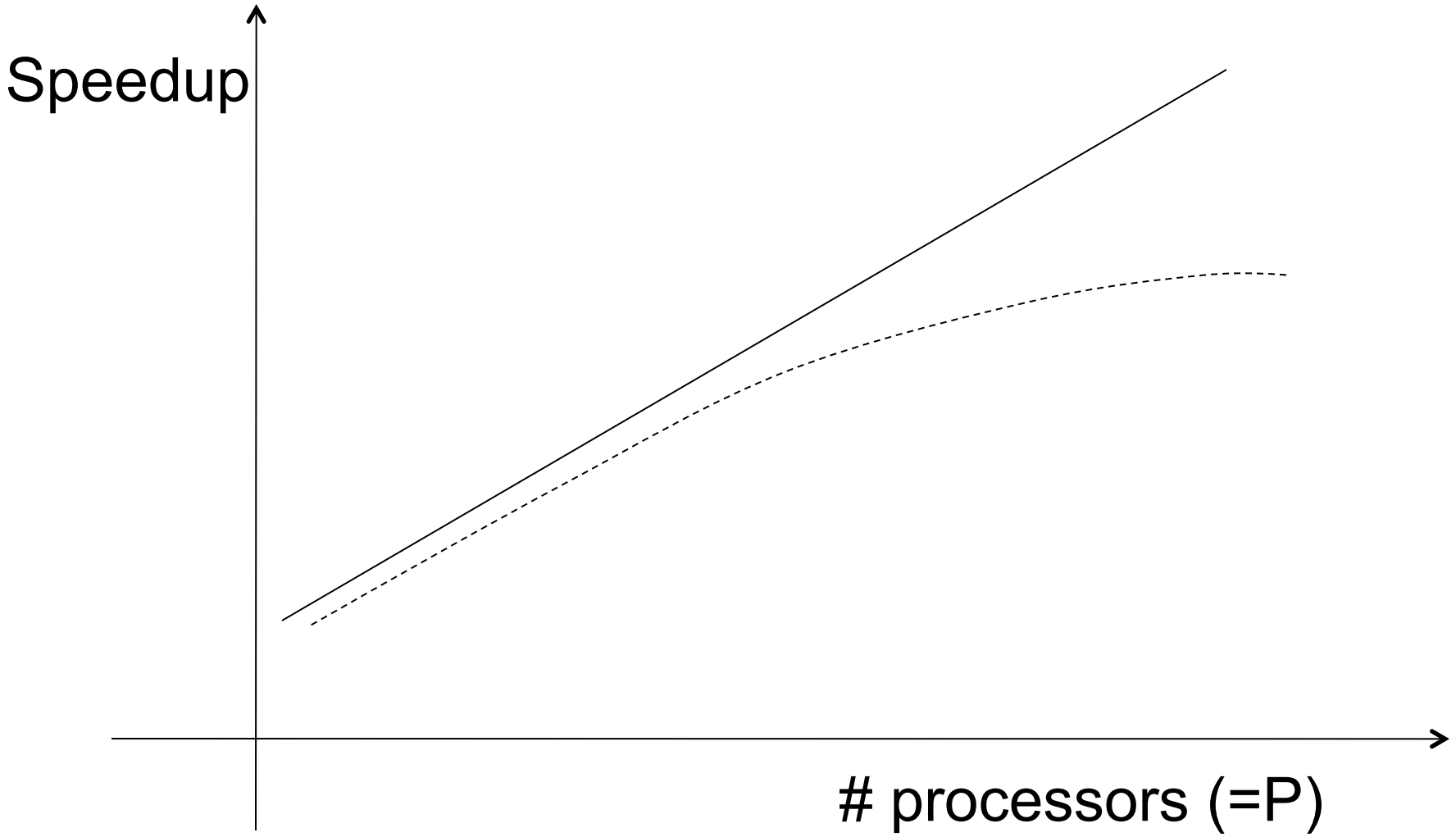
- Goal
  - Improve performance by executing multiple operations in parallel
- Key benefit
  - Cheaper to scale than relying on a single increasingly more powerful processor
- Key challenge
  - Ensure overhead and contention do not kill performance

Performance Metrics for Parallel DBMSs

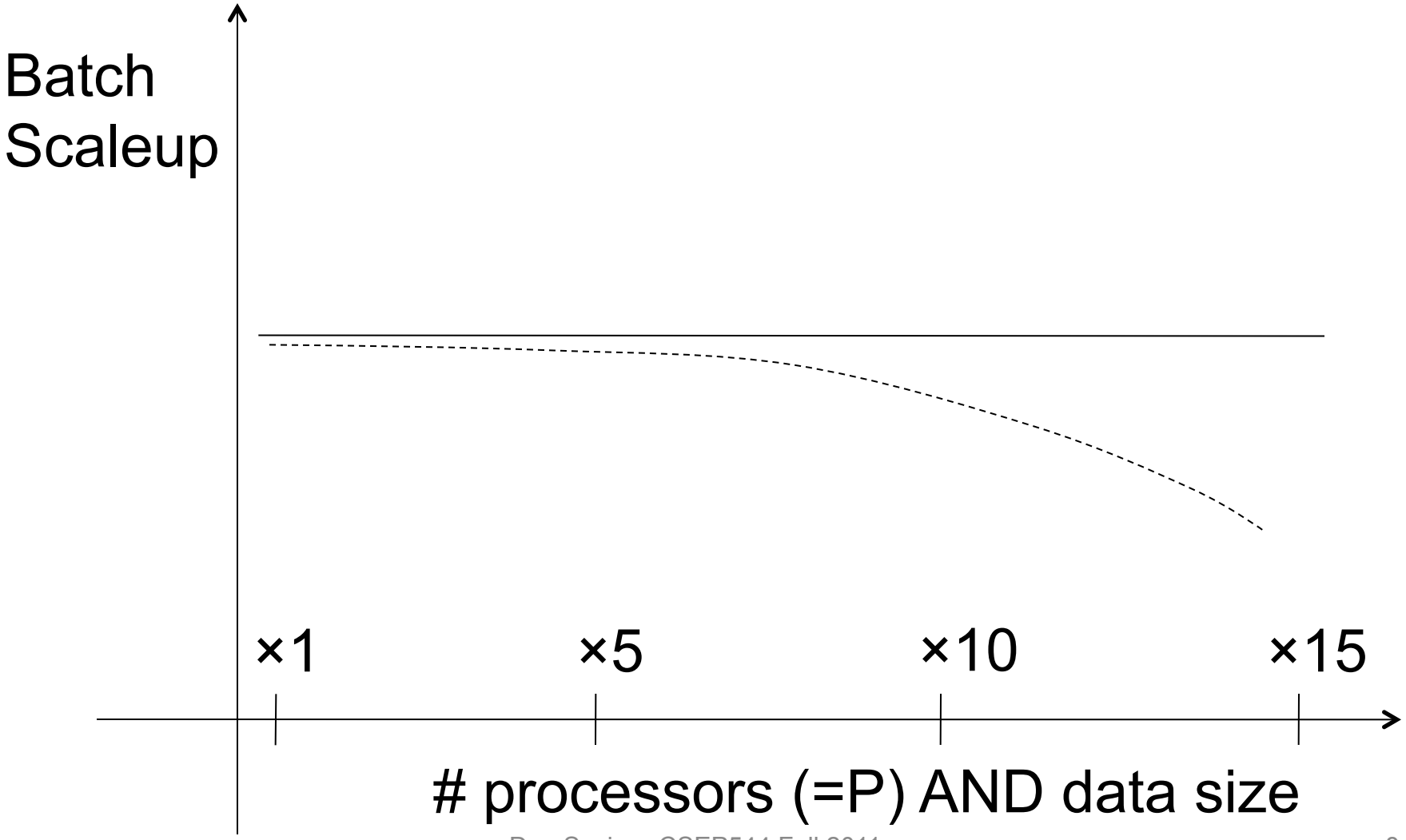
- Speedup
  - More processors ➔ higher speed
  - Individual queries should run faster
  - Should do more transactions per second (TPS)

- Scaleup
  - More processors ➔ can process more data
  - Batch scaleup
    - Same query on larger input data should take the same time
  - Transaction scaleup
    - N-times as many TPS on N-times larger database
    - But each transaction typically remains small

Linear v.s. Non-linear Speedup



Linear v.s. Non-linear Scaleup



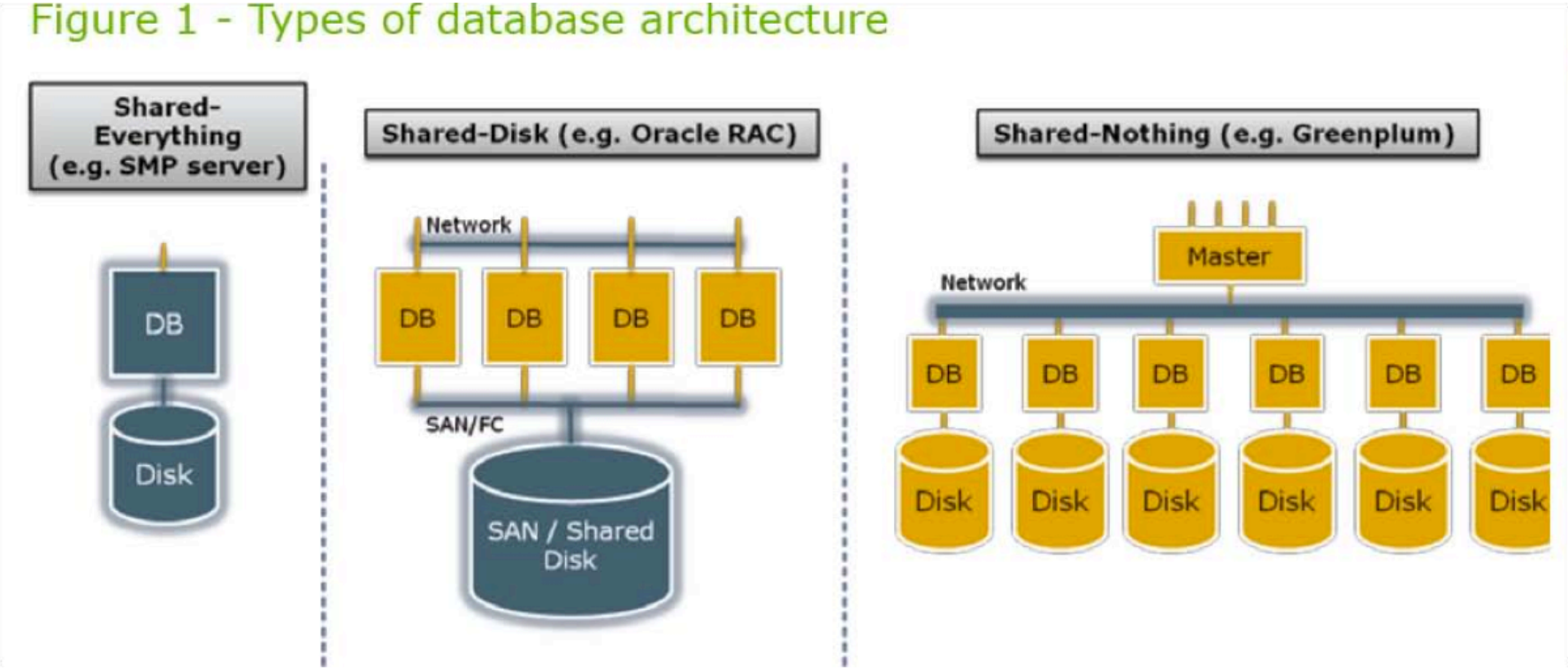
Challenges to Linear Speedup and Scaleup

- Startup cost
  - Cost of starting an operation on many processors
- Interference
  - Contention for resources between processors
- Skew
  - Slowest processor becomes the bottleneck

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Architectures for Parallel Databases



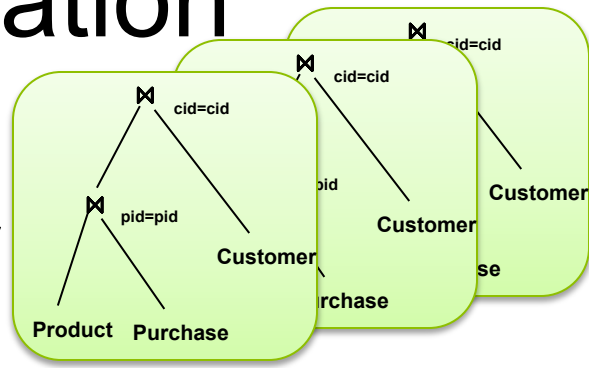
From: Greenplum Database Whitepaper

Shared Nothing

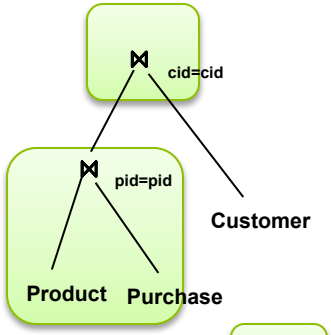
- Most scalable architecture
  - Minimizes interference by minimizing resource sharing
  - Can use commodity hardware
  - Processor = server = node
  - P = number of nodes
- Also most difficult to program and manage

Taxonomy for Parallel Query Evaluation

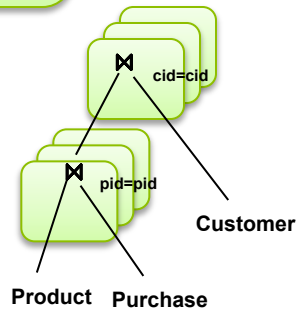
- Inter-query parallelism
  - Each query runs on one processor



- Inter-operator parallelism
  - A query runs on multiple processors
  - An operator runs on one processor



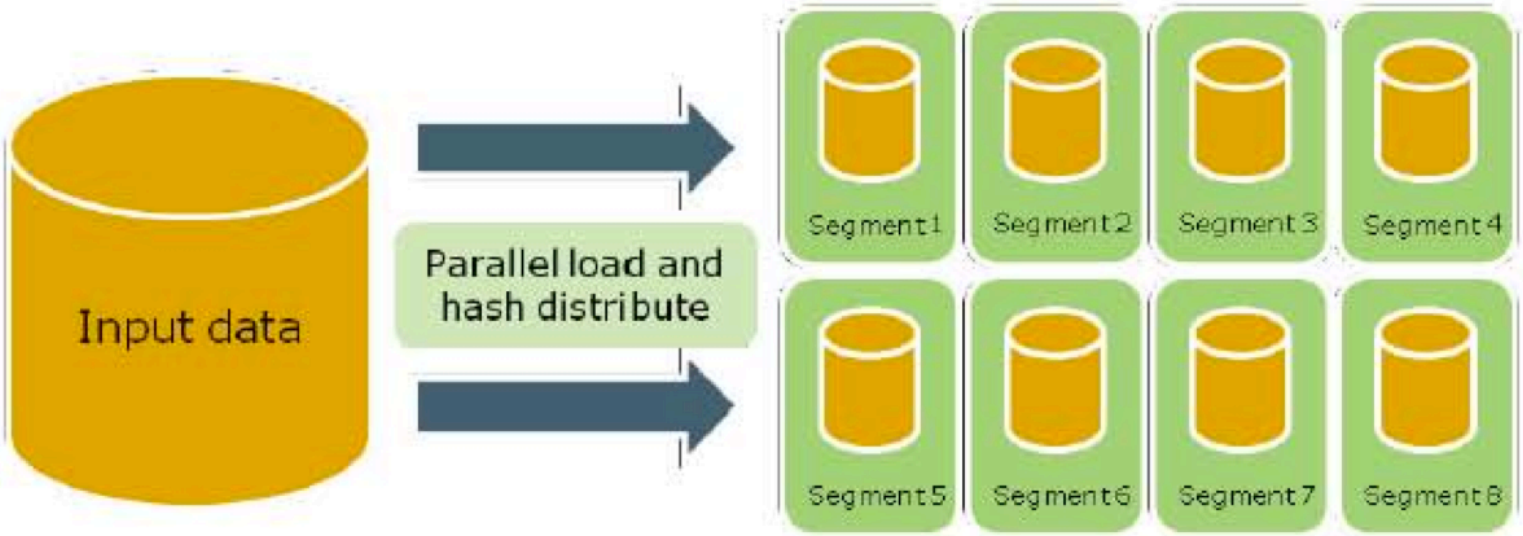
- Intra-operator parallelism
  - An operator runs on multiple processors



We study only intra-operator parallelism: most scalable

Query Evaluation

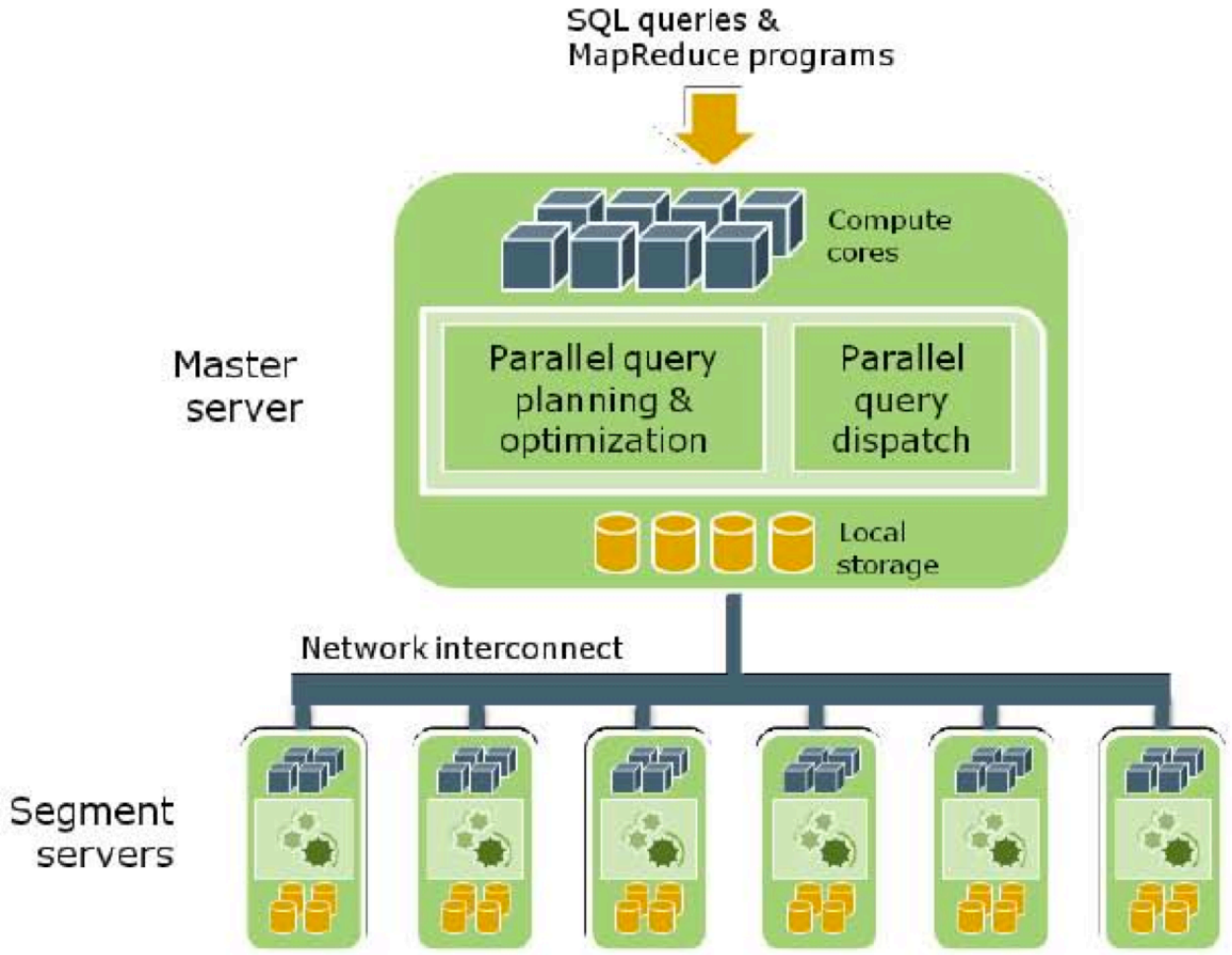
Figure 3 - Automatic hash-based data distribution



From: Greenplum Database Whitepaper

Query Evaluation

Figure 5 - Master server performs global planning and dispatch



From: Greenplum Database Whitepaper

Horizontal Data Partitioning

- Relation R split into P chunks R_0, …, R_{P-1}, stored at the P nodes
- Round robin: tuple t_i to chunk (i mod P)
- Hash based partitioning on attribute A:
  - Tuple t to chunk h(t.A) mod P
- Range based partitioning on attribute A:
  - Tuple t to chunk i if $v_{i-1} < t.A < v_i$

Parallel Selection

Compute $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- On a conventional database: cost = B(R)

- Q: What is the cost on a parallel database with P processors ?
  - Round robin
  - Hash partitioned
  - Range partitioned

Parallel Selection

- Q: What is the cost on a parallel database with P processors ?
- Round robin: all servers do the work
  - Parallel time = B(R)/P; total work = B(R)
  - Good load balance but needs to read all the data
- Hash:
  - $\sigma_{A=v}(R)$: Parallel time = total work = B(R)/P
  - $\sigma_{A\in[v1,v2]}(R)$: Parallel time = B(R)/P; total work = B(R)
- Range: one server only
  - Parallel time = total work = B(R)
  - Works well for range predicates but suffers from data skew

Parallel Group By

- Given R(A,B,C) , compute $\gamma_{A,\ sum(B)}(R)$
- Assume R is hash-partitioned on C

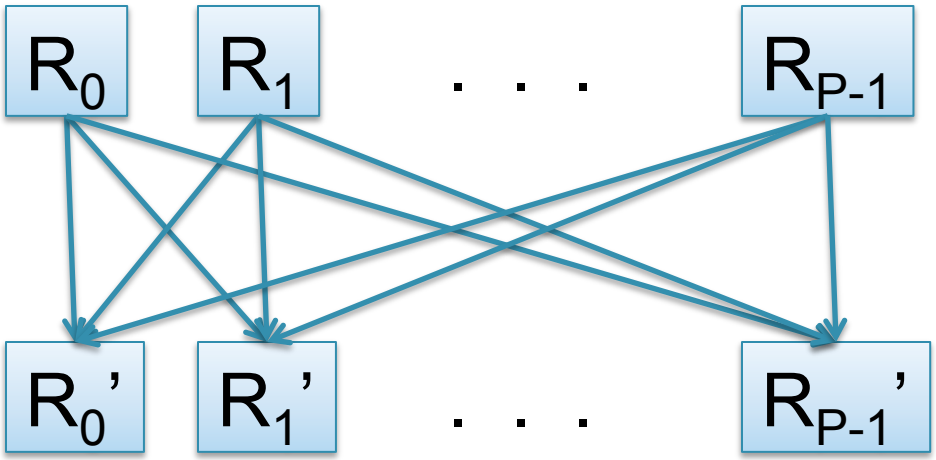
R_0 R_1 . . . R_{P-1}

- How do we compute $\gamma_{A,\ sum(B)}(R)$?

Parallel Group By

Compute $\gamma_{A,\ sum(B)}(R)$

- Step 1: server i hash-partitions chunk R_i using h(t.A):
  $R_{i0}, R_{i1}, \ldots, R_{i,P-1}$
- Step 2: server i sends partition R_{ij} to serve j
- Step 3: server j computes $\gamma_{A,\ sum(B)}(R_{0j} \cup R_{1j} \cup \ldots \cup R_{P-1,j})$

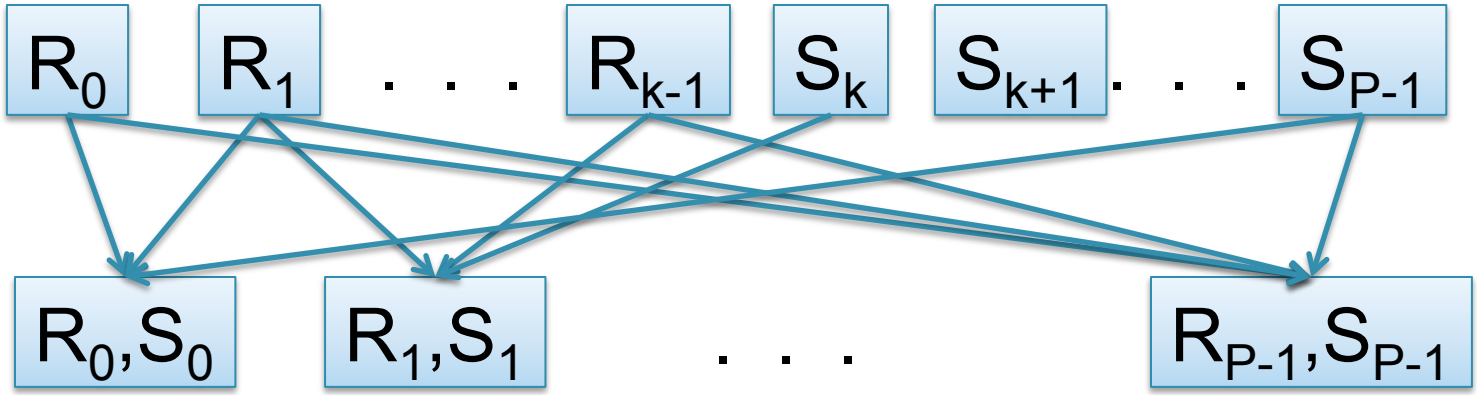


Parallel Join

- How do we compute $R(A,B) \bowtie S(B,C)$?

Parallel Join

- Step 1
  - For all servers in [0,k-1], server i partitions chunk R_i using a hash function h(t.A) mod P: R_{i0}, R_{i1}, …, $R_{i,P-1}$
  - For all servers in [k,P], server j partitions chunk S_j using a hash function h(t.A) mod P: S_{j0}, S_{j1}, …, $R_{j,P-1}$
- Step 2:
  - Servers i=0..k-1 send partition R_{iu} to server u
  - Servers j=k..P send partition S_{ju} to server u
- Steps 3: Server u computes the join of R_{iu} with S_{ju}



Parallel Dataflow Implementation

- Use relational operators unchanged
- Add special split and merge operators
  - Handle data routing, buffering, and flow control
- Example: exchange (or “shuffle”) operator
  - Inserted between consecutive operators in the query plan
  - Can act as either a producer or consumer
  - Producer pulls data from operator and sends to n consumers
    - Producer acts as driver for operators below it in query plan
  - Consumer buffers input data from n producers and makes it available to operator through getNext interface

Big Data

Big Data

A buzzword that means several things:

- Large data mining:
  - More and more corporations have data as the basis of their business, not just "nice to have”
- “External” data is increasingly mined too:
  - Blogs, emails, reviews, websites
- Sometimes it just means using cluster computing for large data analytics

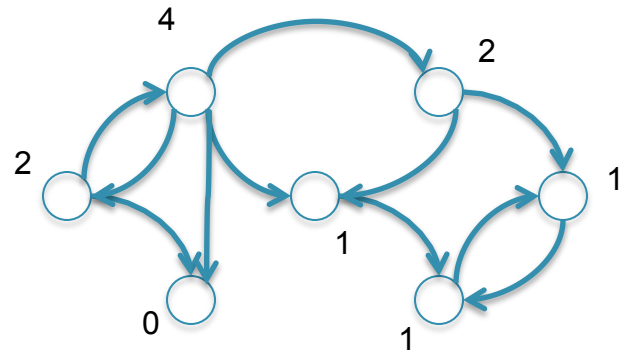
Famous Example of Big Data Analysis

Kumar et al., *The Web as a Graph*

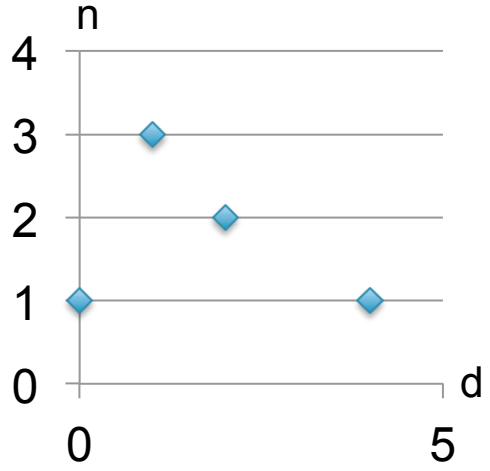
- Question 1: is the Web like a "random graph"?
  - Random Graphs introduced by Erdos and Reny in the 1940s
  - Extensively studied in mathematics, well understood
  - If the Web is a "random graph", then we have mathematical tools to understand it: clusters, communities, diameter, etc
- Question 2: how does the Web graph look like?

Histogram of a Graph

- Outdegree of a node = number of outgoing edges
- For each d, let n(d) = number of nodes with oudegree d
- The outdegree histogram of a graph = the scatterplot (d, n(d))

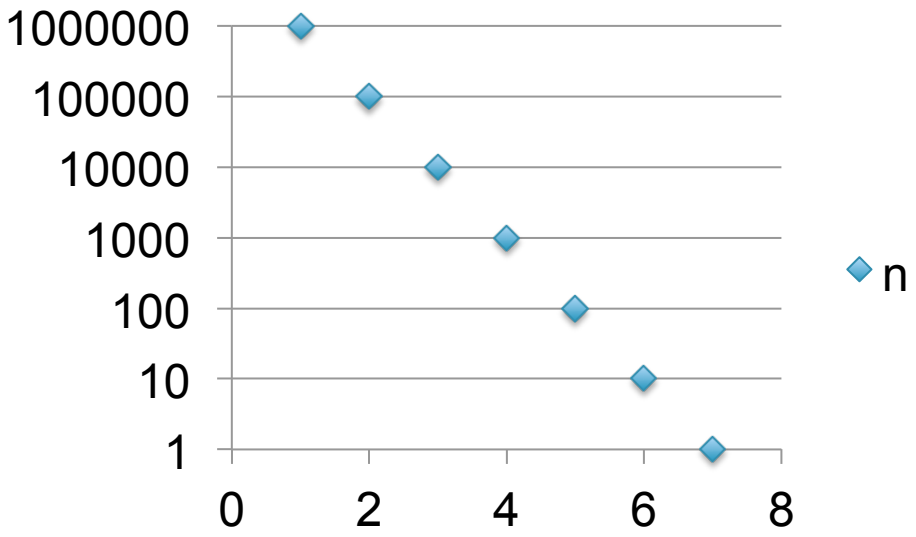


d	n(d)
0	1
1	3
2	2
3	0
4	1



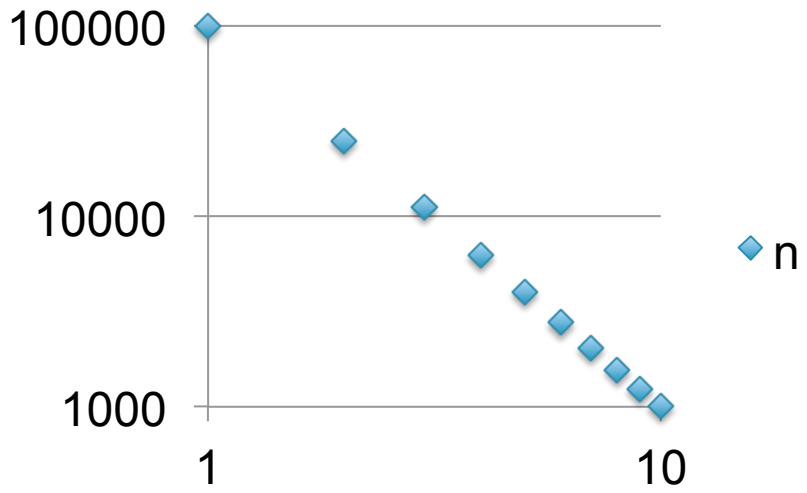
Exponential Distribution

- $n(d) \cong c/2^d$ (generally, cx^d, for some $x < 1$)
- A *random graph* has exponential distribution
- Best seen when n is on a log scale



Zipf Distribution

- $n(d) \cong 1/d^x$, for some value $x>0$
- Human-generated data has Zipf distribution: letters in alphabet, words in vocabulary, etc.
- Best seen in a log-log scale (why ?)



The Histogram of the Web

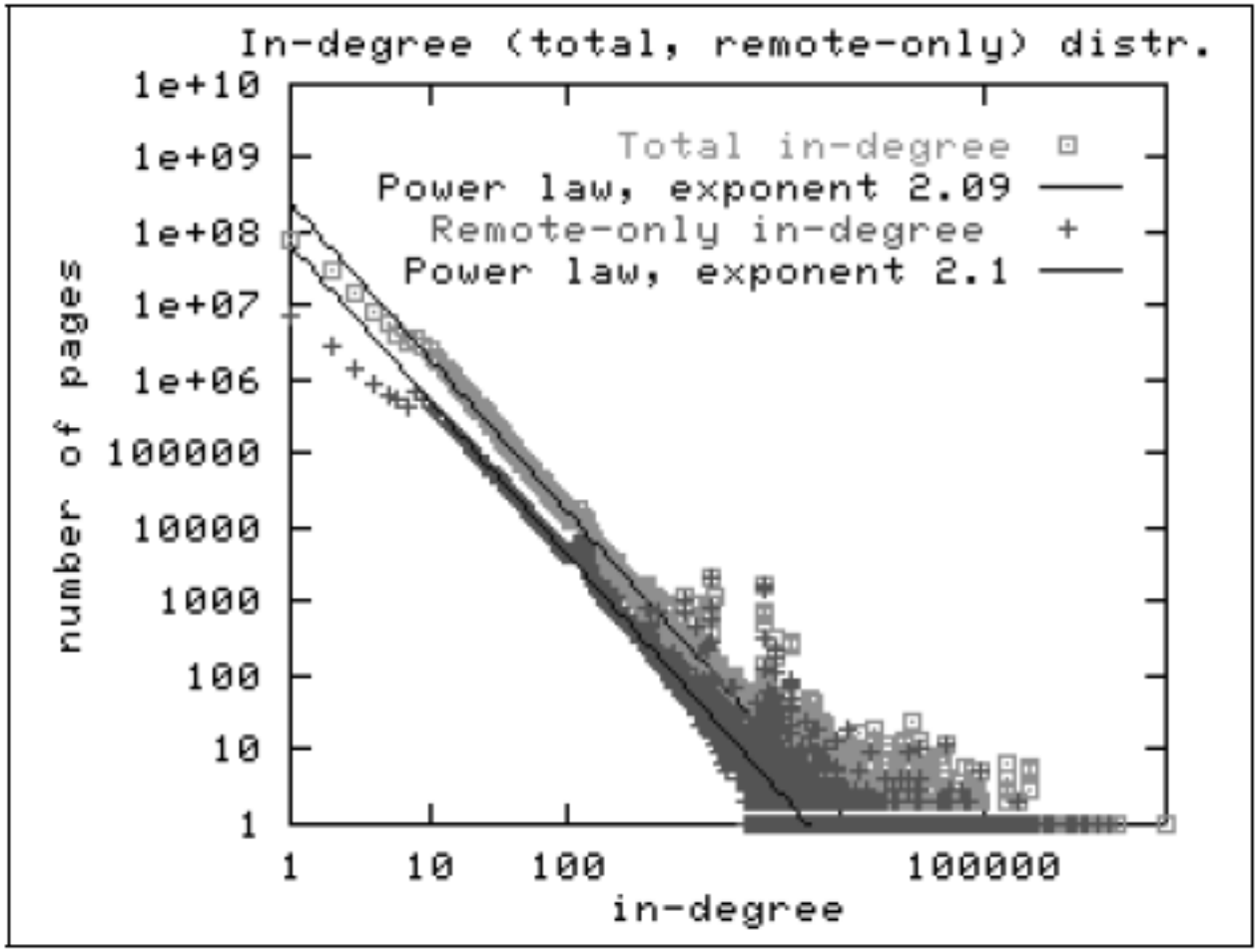


Figure 2: In-degree distribution.

Late 1990's
200M Webpages

Exponential ?

Zipf ?

The Bowtie Structure of the Web

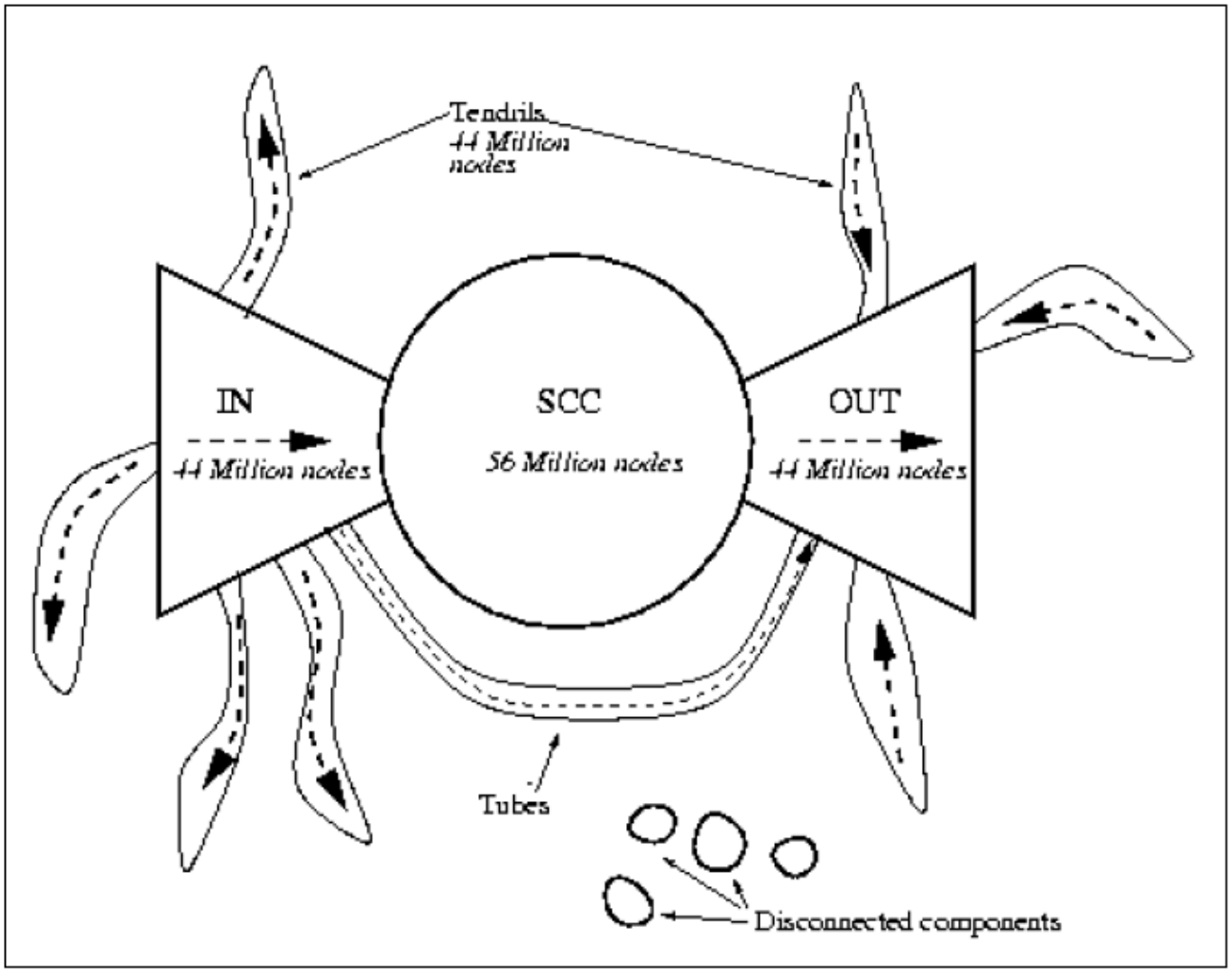


Figure 4: The web as a bowtie. SCC is a giant strongly connected component. IN consists of pages with paths to SCC, but no path from SCC. OUT consists of pages with paths from SCC, but no path to SCC. TENDRILS consists of pages that cannot surf to SCC, and which cannot be reached by surfing from SCC.

Big Data: Summary

- Today, such analysis are done daily, by all large corporations
- Increasingly, using Cluster Computing:
  - Distributed File System (for storing the data)
  - Map/reduce
  - Declarative languages *over* Map/Reduce: Pig-Latin, SQL, Hive, Scope, Dryad-Linq, …

Cluster Computing

Cluster Computing

- Large number of commodity servers, connected by high speed, commodity network
- Rack: holds a small number of servers
- Data center: holds many racks

Cluster Computing

- Massive parallelism:
  - 100s, or 1000s, or 10000s servers
  - Many hours
- Failure:
  - If medium-time-between-failure is 1 year
  - Then 10000 servers have one failure / hour

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

Map Reduce

- Google: paper published 2004
- Free variant: Hadoop

- Map-reduce = high-level programming model and implementation for large-scale parallel data processing

Data Model

Files !

A file = a bag of `(key, value)` pairs

A map-reduce program:

- Input: a bag of `(inputkey, value)` pairs
- Output: a bag of `(outputkey, value)` pairs

Step 1: the MAP Phase

User provides the MAP-function:

- Input: `(input key, value)`
- Ouput:
  bag of `(intermediate key, value`

System applies the map function in parallel to all `(input key, value)` pairs in the input file

Step 2: the REDUCE Phase

User provides the REDUCE function:

- Input: `(intermediate key, bag of values)`
- Output: bag of output `values`

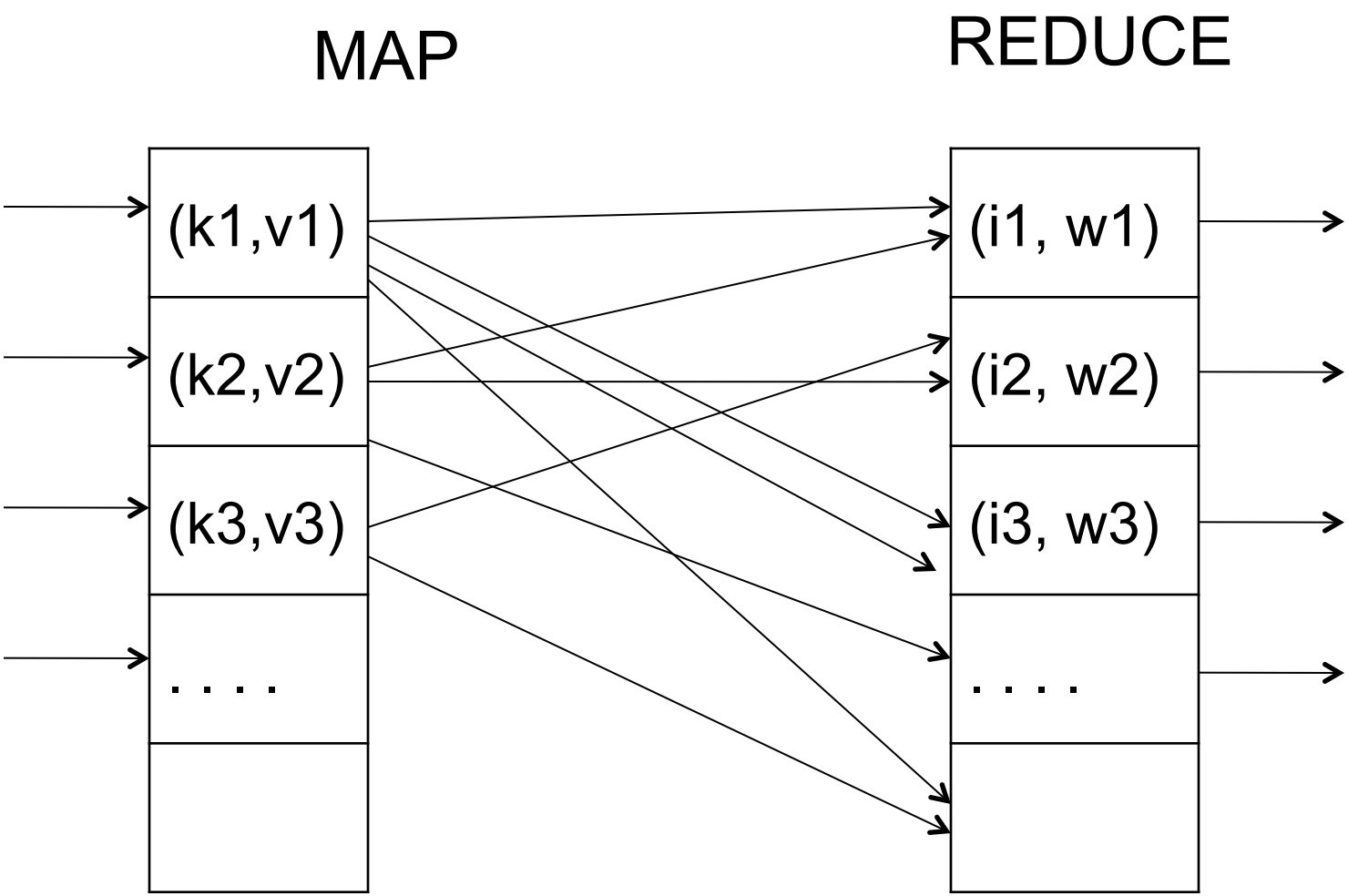
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

Example

- Counting the number of occurrences of each word in a large collection of

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, “1”):
```

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```


MAP
REDUCE
(k1,v1)
(k2,v2)
(k3,v3)
. . . .
(i1, w1)
(i2, w2)
(i3, w3)
. . . .

Map = GROUP BY, Reduce = Aggregate

R(documentID, word)

```
SELECT word, sum(1)
FROM R
GROUP BY word
```

Example 2: MR word length count

Abridged Declaration of Independence

A Declaration By the Representatives of the United States of America, in General Congress Assembled.
When in the course of human events it becomes necessary for a people to advance from that subordination in which they have hitherto remained, and to assume among powers of the earth the equal and independent station to which the laws of nature and of nature's god entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the change.
We hold these truths to be self-evident; that all men are created equal and independent; that from that equal creation they derive rights inherent and inalienable, among which are the preservation of life, and liberty, and the pursuit of happiness; that to secure these ends, governments are instituted among men, deriving their just power from the consent of the governed; that whenever any form of government shall become destructive of these ends, it is the right of the people to alter or to abolish it, and to institute new government, laying it's foundation on such principles and organizing it's power in such form, as to them shall seem most likely to effect their safety and happiness. Prudence indeed will dictate that governments long established should not be changed for light and transient causes: and accordingly all experience hath shewn that mankind are more disposed to suffer while evils are sufferable, than to right themselves by abolishing the forms to which they are accustomed. But when a long train of abuses and usurpations, begun at a distinguished period, and pursuing invariably the same object, evinces a design to reduce them to arbitrary power, it is their right, it is their duty, to throw off such government and to provide new guards for future security. Such has been the patient sufferings of the colonies; and such is now the necessity which constrains them to expunge their former systems of government. the history of his present majesty is a history of unremitting injuries and usurpations, among which no one fact stands single or solitary to contradict the uniform tenor of the rest, all of which have in direct object the establishment of an absolute tyranny over these states. To prove this, let facts be submitted to a candid world, for the truth of which we pledge a faith yet unsullied by falsehood.

Example 2: MR word length count

Map Task 1
(204 words)

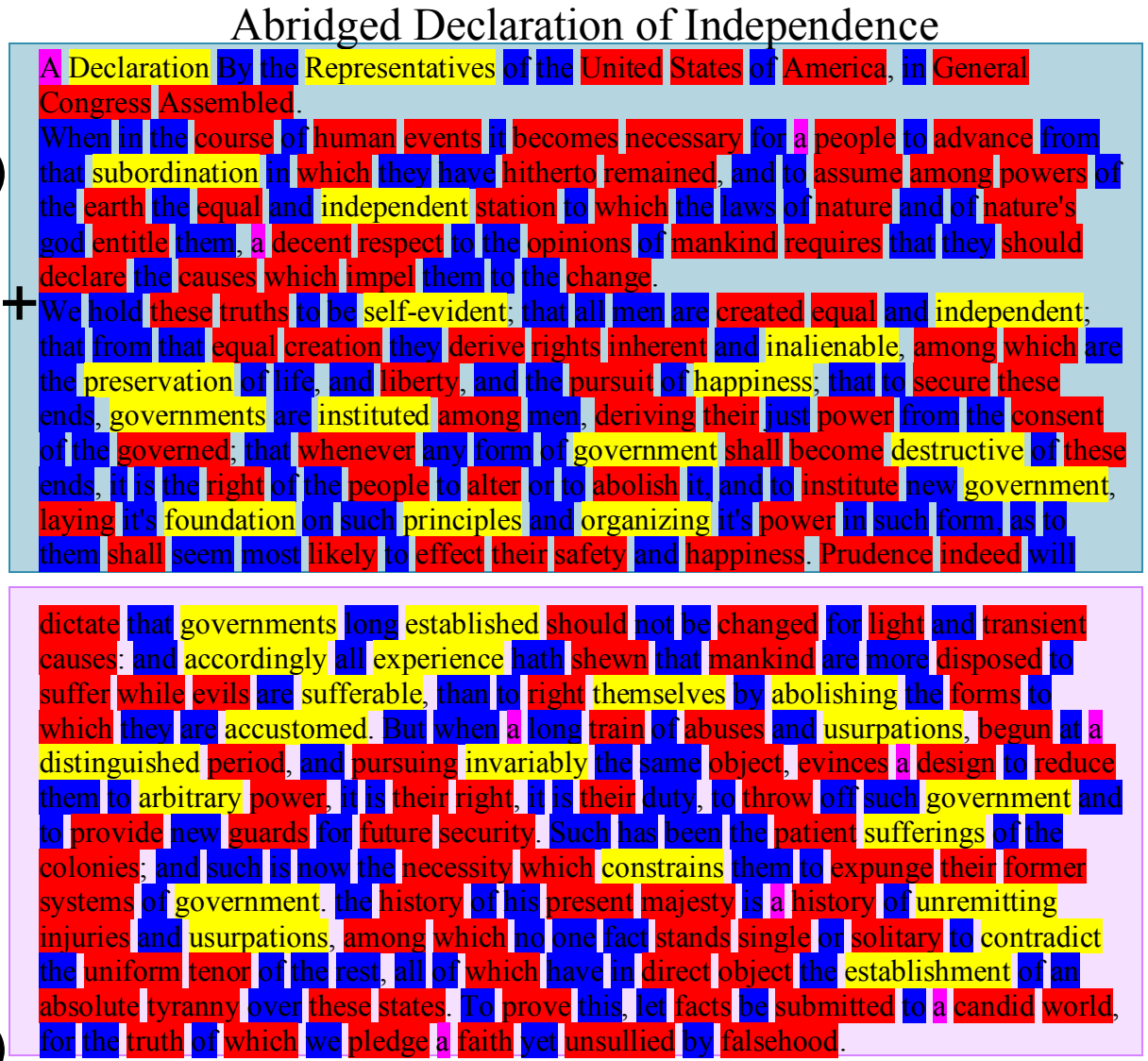
Yellow: 10+

Red: 5..9

Blue: 2..4

Pink: = 1

Map Task 2
(190 words)



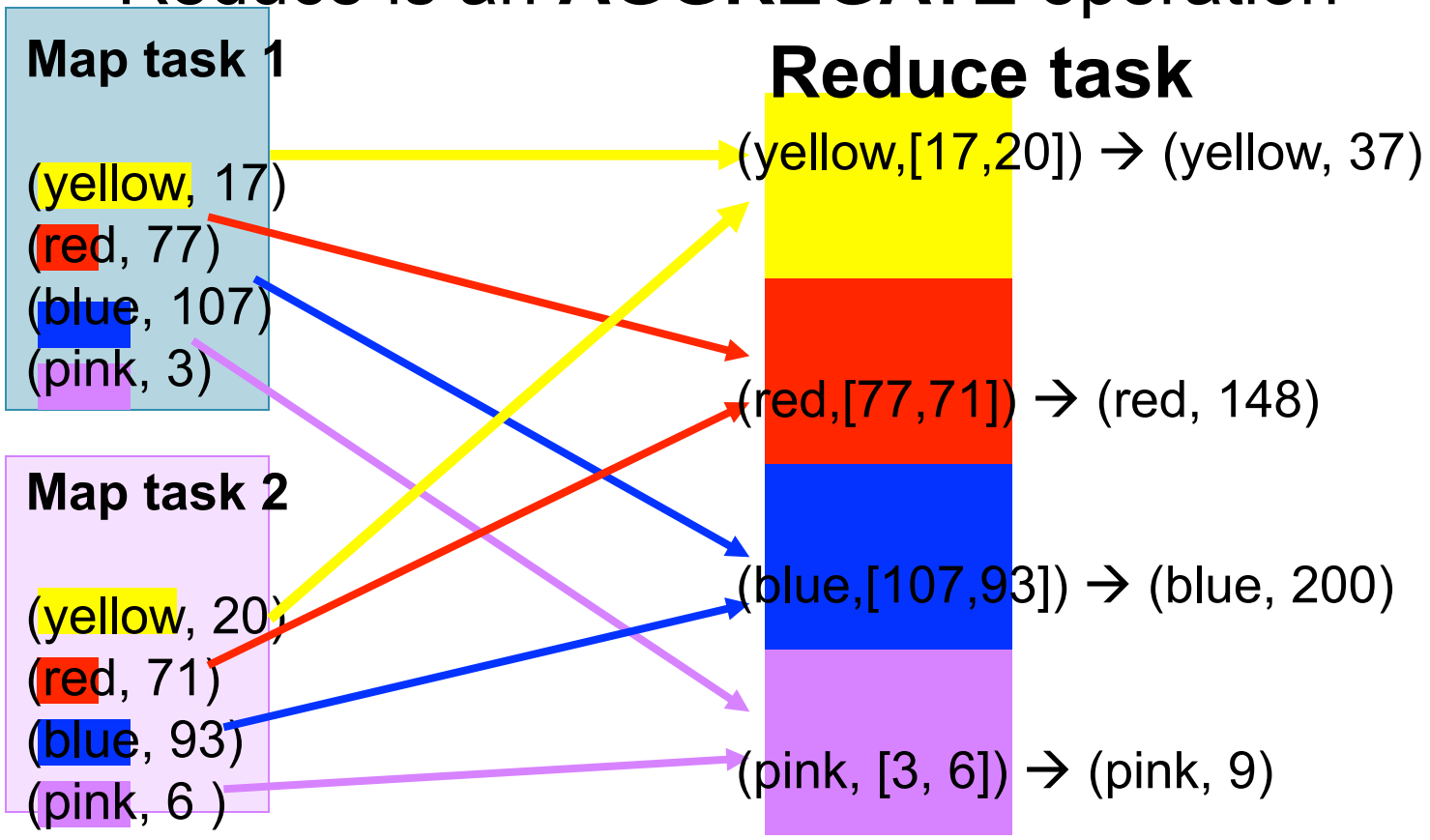
(key, value)

(yellow, 17)
(red, 77)
(blue, 107)
(pink, 3)

(yellow, 20)
(red, 71)
(blue, 93)
(pink, 6)

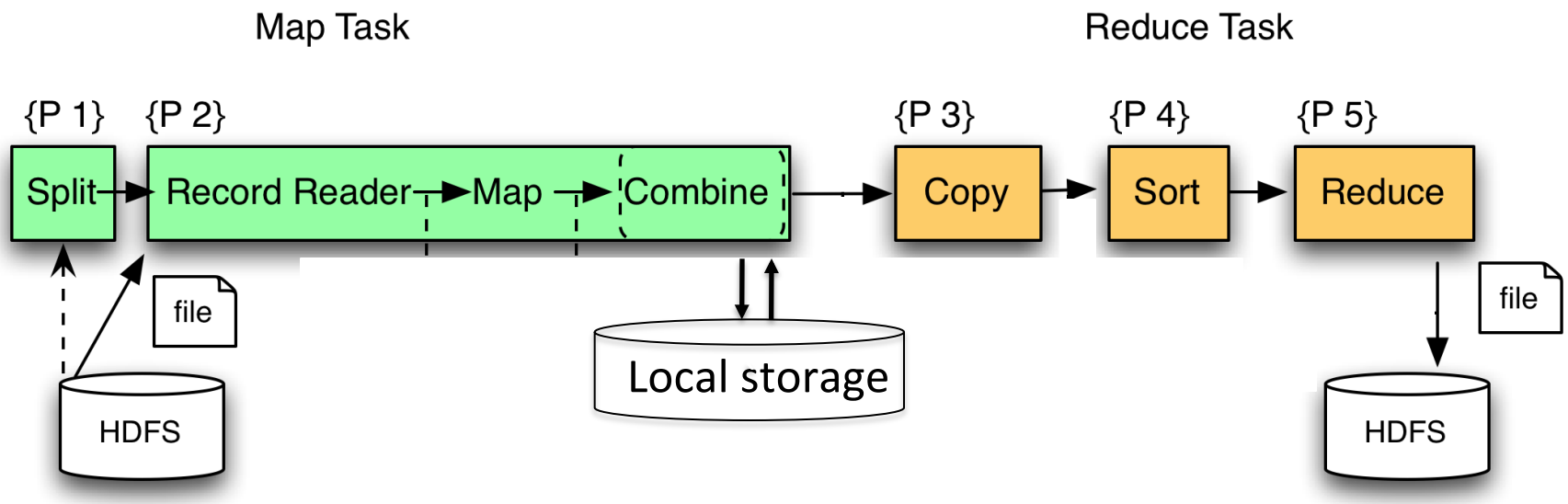
Example 2: MR word length count

Map is a **GROUP BY** operation
Reduce is a **AGGREGATE** operation



MR Phases

- Each Map and Reduce task has multiple phases:



Implementation

- There is one master node
- Master partitions input file into *M splits*, by key
- Master assigns *workers* (=servers) to the *M map tasks*, keeps track of their progress
- Workers write their output to local disk, partition into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce workers read regions from the map workers' local disks

Interesting Implementation Details

Worker failure:

- Master pings workers periodically,
- If down then reassigns the task to another worker

Interesting Implementation Details

Backup tasks:

- *Straggler* = a machine that takes unusually long time to complete one of the last tasks. Eg:
  - Bad disk forces frequent correctable errors (30MB/s → 1MB/s)
  - The cluster scheduler has scheduled other tasks on that machine
- Stragglers are a main reason for slowdown
- Solution*: pre-emptive backup execution of the last few remaining in-progress tasks*

Tuning Map-Reduce

- It is very hard!
- Choice of M and R:
  - Larger is better for load balancing
  - Limitation: master needs O(M×R) memory
  - Typical choice:
    - M=number of chunks (“number of blocks”)
    - R=much smaller (why??); rule of thumb: R=1.5 * number of servers (does AWS follow it?)

Tuning Map-Reduce

- Lots of other parameters: partition function, sort factor, etc, etc
- The combiner! (Talk in class…)
- Over 100 parameters to tune; about 50 affect running time significantly
- Active research on automatic tuning

[Shivnath Babu, SOCC'2010

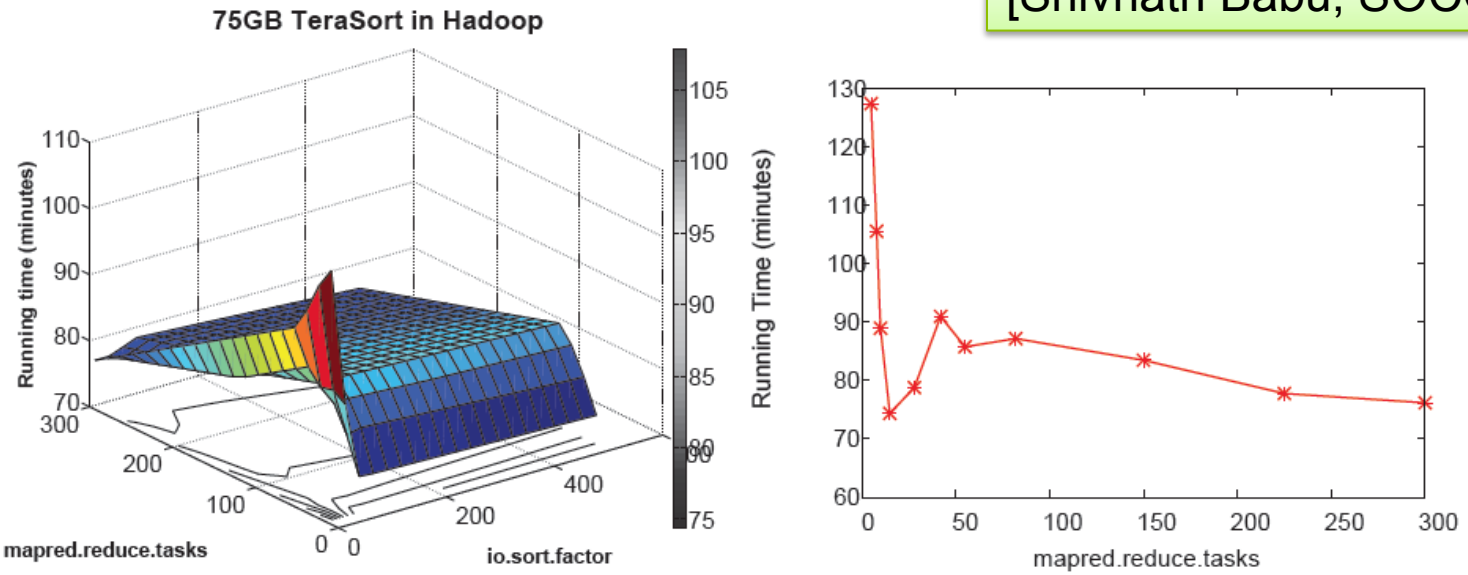


Figure 2: (a) 2D response surface of the TeraSort MapReduce program in Hadoop over a 75GB dataset, with mapred.reduce.tasks $\in [15, 300]$ and io.sort.factor $\in [10, 500]$; (b) a 1D projection of the surface for io.sort.factor=500 for mapred.reduce.tasks $\in [5, 300]$

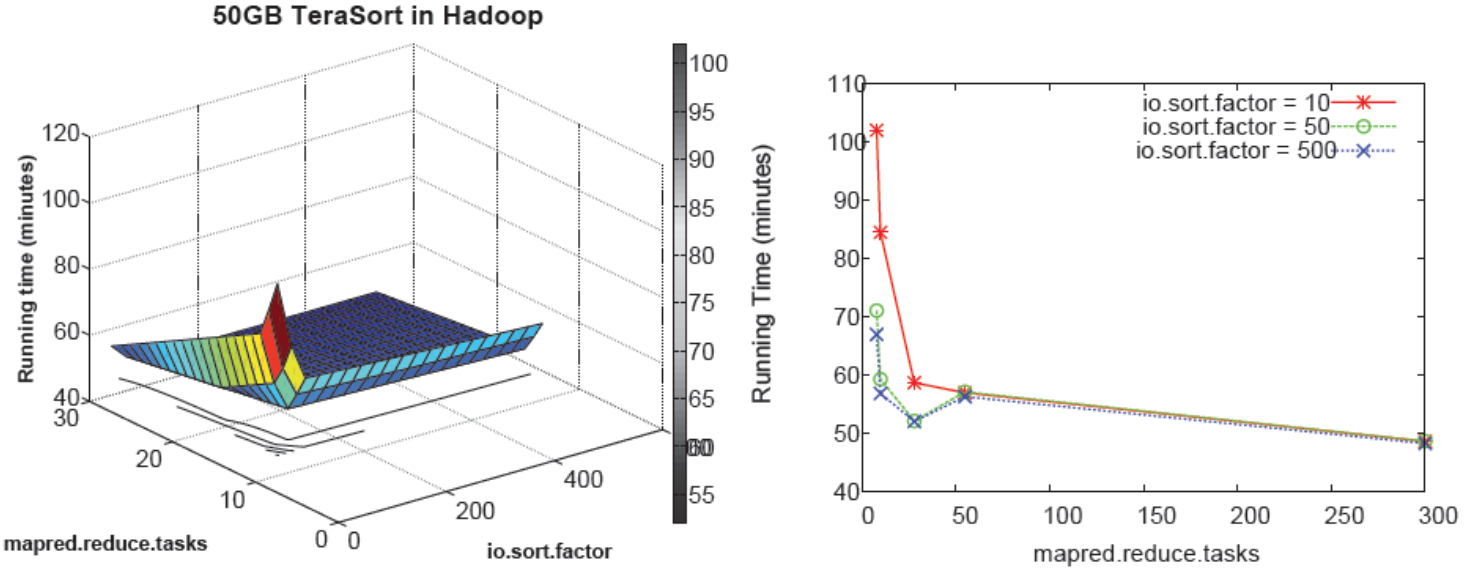


Figure 3: (a) 2D response surface of the TeraSort MapReduce program in Hadoop over a 50GB dataset, with mapred.reduce.tasks $\in [8, 28]$ and io.sort.factor $\in [10, 500]$; (b) 1D projections of the surface for different io.sort.factor for mapred.reduce.tasks $\in [8, 300]$

Map-Reduce Summary

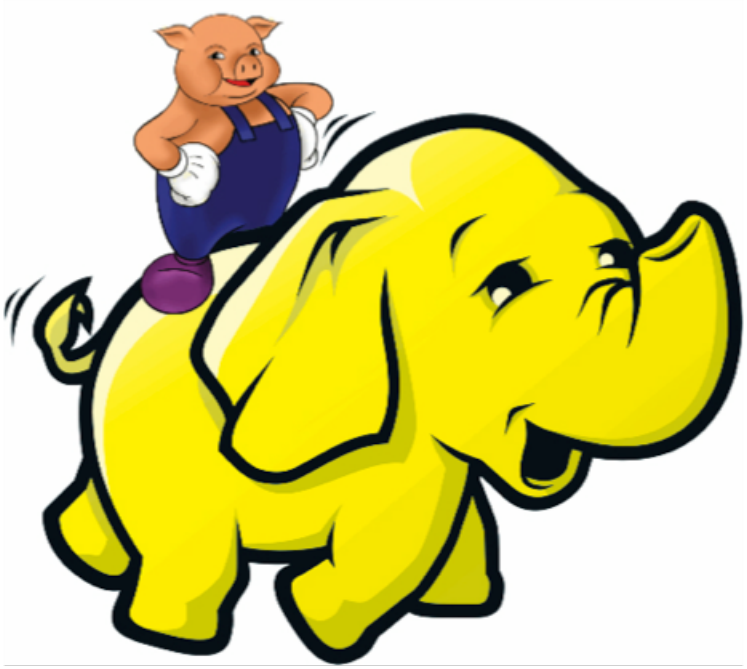
- Hides scheduling and parallelization details
- However, very limited queries
  - Difficult to write more complex tasks
  - Need multiple map-reduce operations
- Solution: declarative query language

Declarative Languages on MR

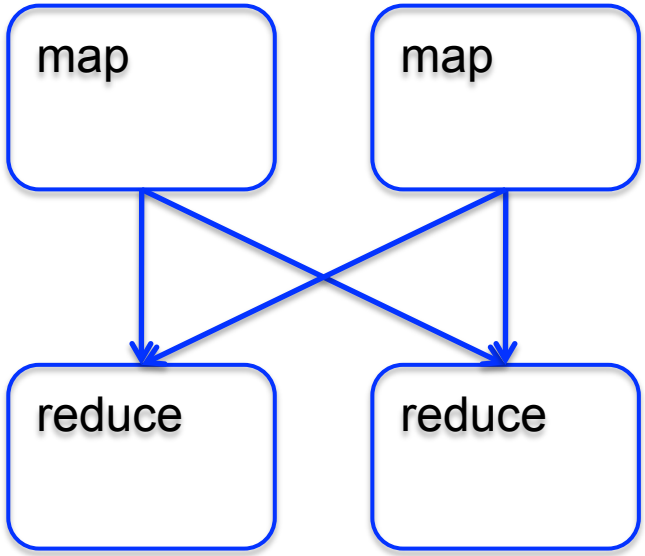
- PIG Latin (Yahoo!)
  - New language, like Relational Algebra
  - Open source
- SQL / Tenzing (google)
  - SQL on MR
  - Proprietary
- Others (won't discuss):
  - Scope (MS): SQL; proprietary
  - DryadLINQ (MS): LINQ; proprietary
  - Clustera (other UW) : SQL; Not publicly available

What is Pig?

- An engine for executing programs on top of Hadoop
- It provides a language, Pig Latin, to specify these programs
- An Apache open source project
  http://hadoop.apache.org/pig/



Map Reduce Illustrated

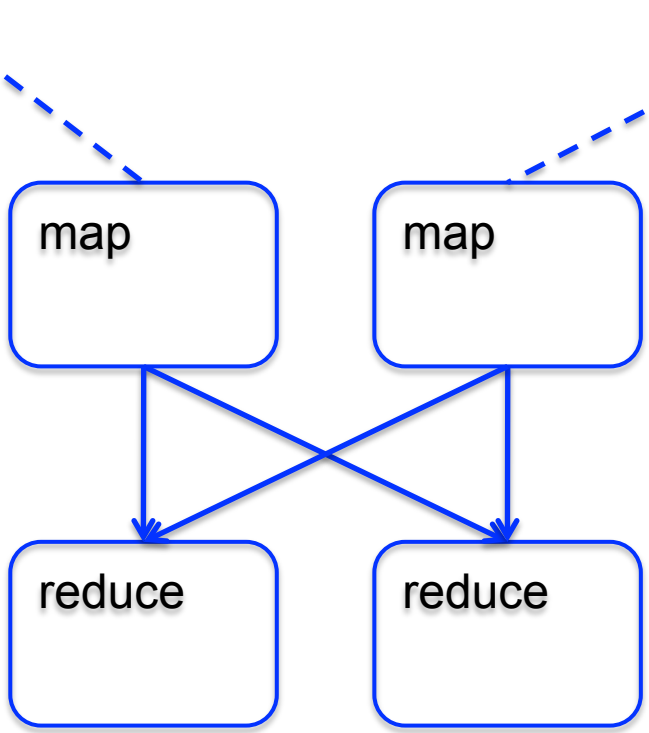


Credit: Alan Gates, Yahoo!

Map Reduce Illustrated

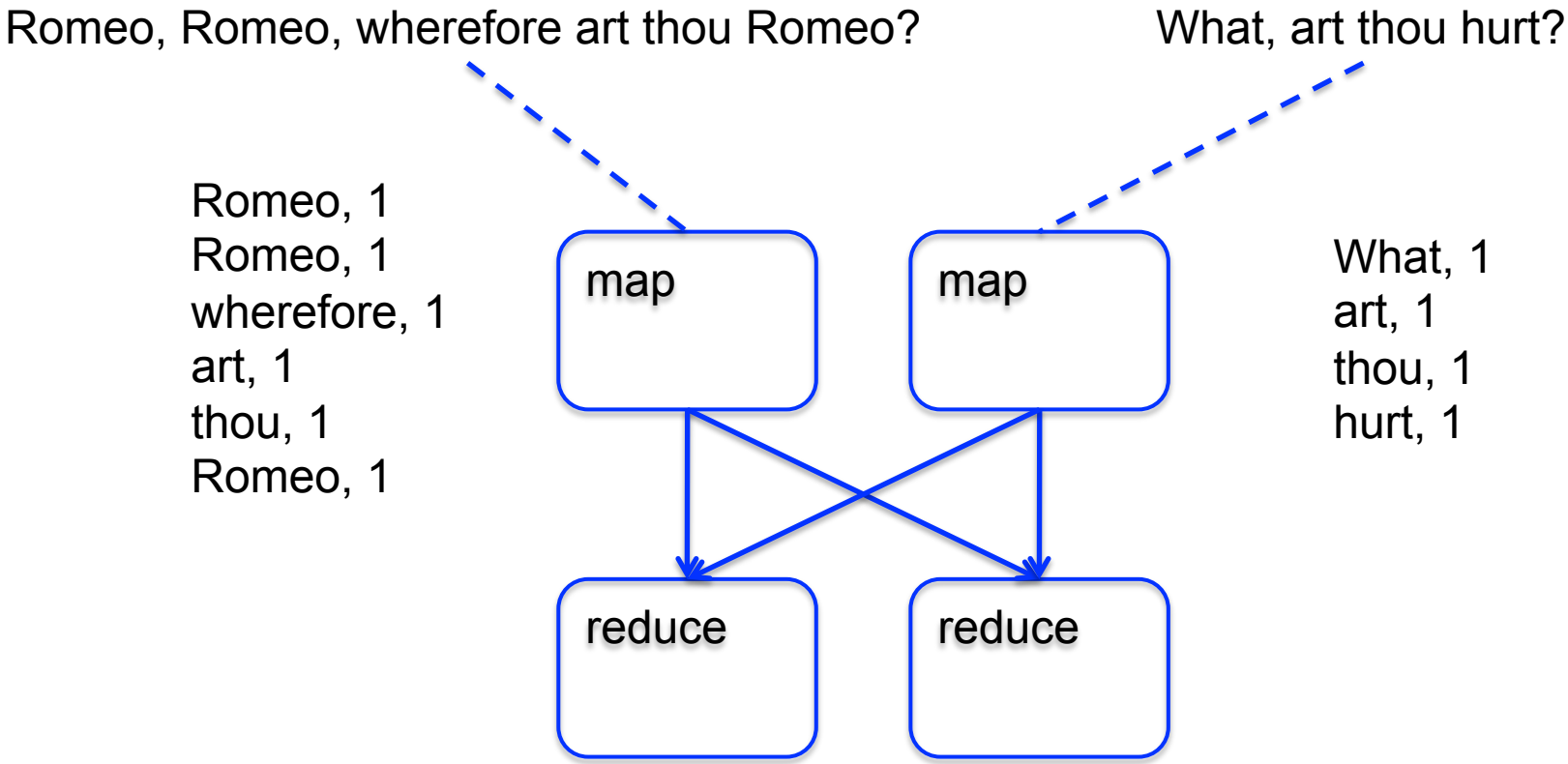
Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?



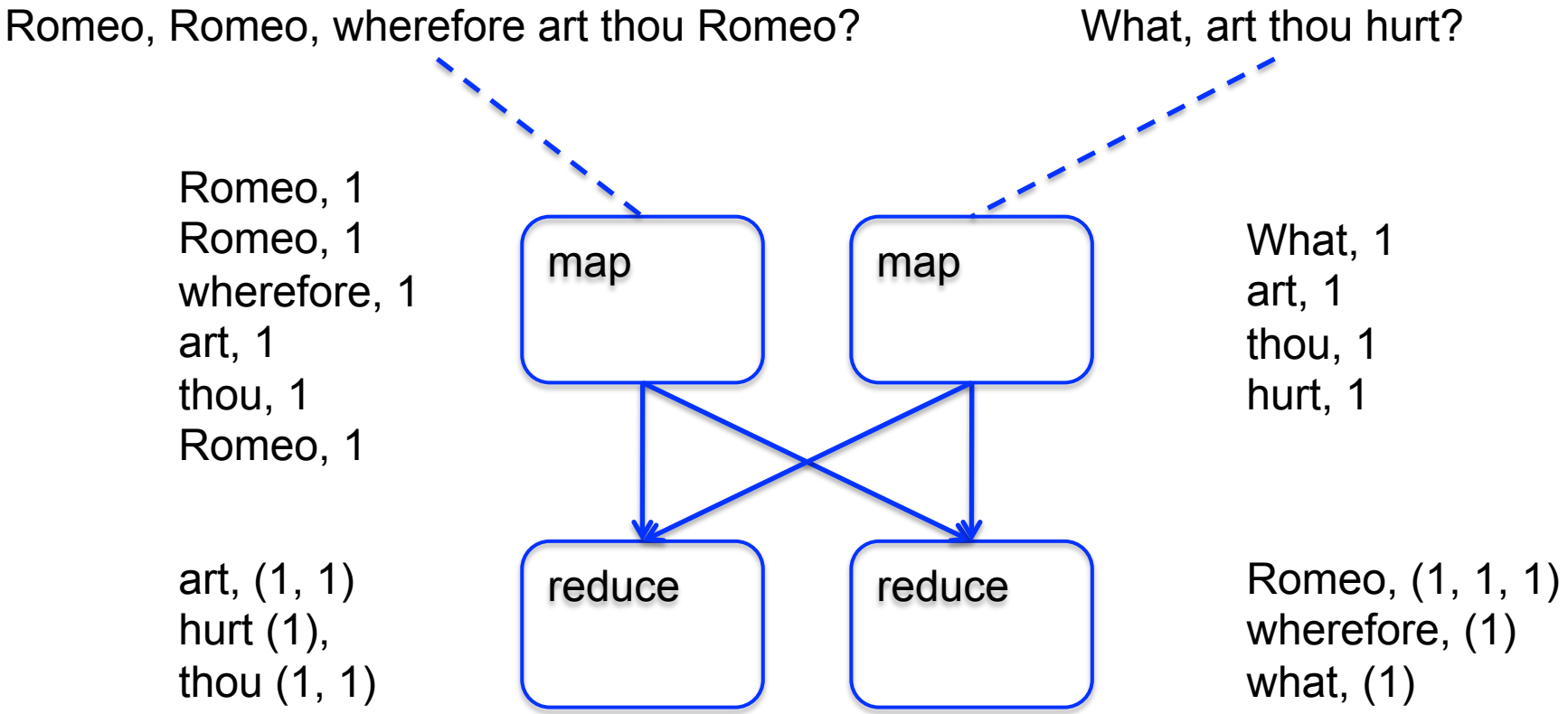
Credit: Alan Gates, Yahoo!

Map Reduce Illustrated

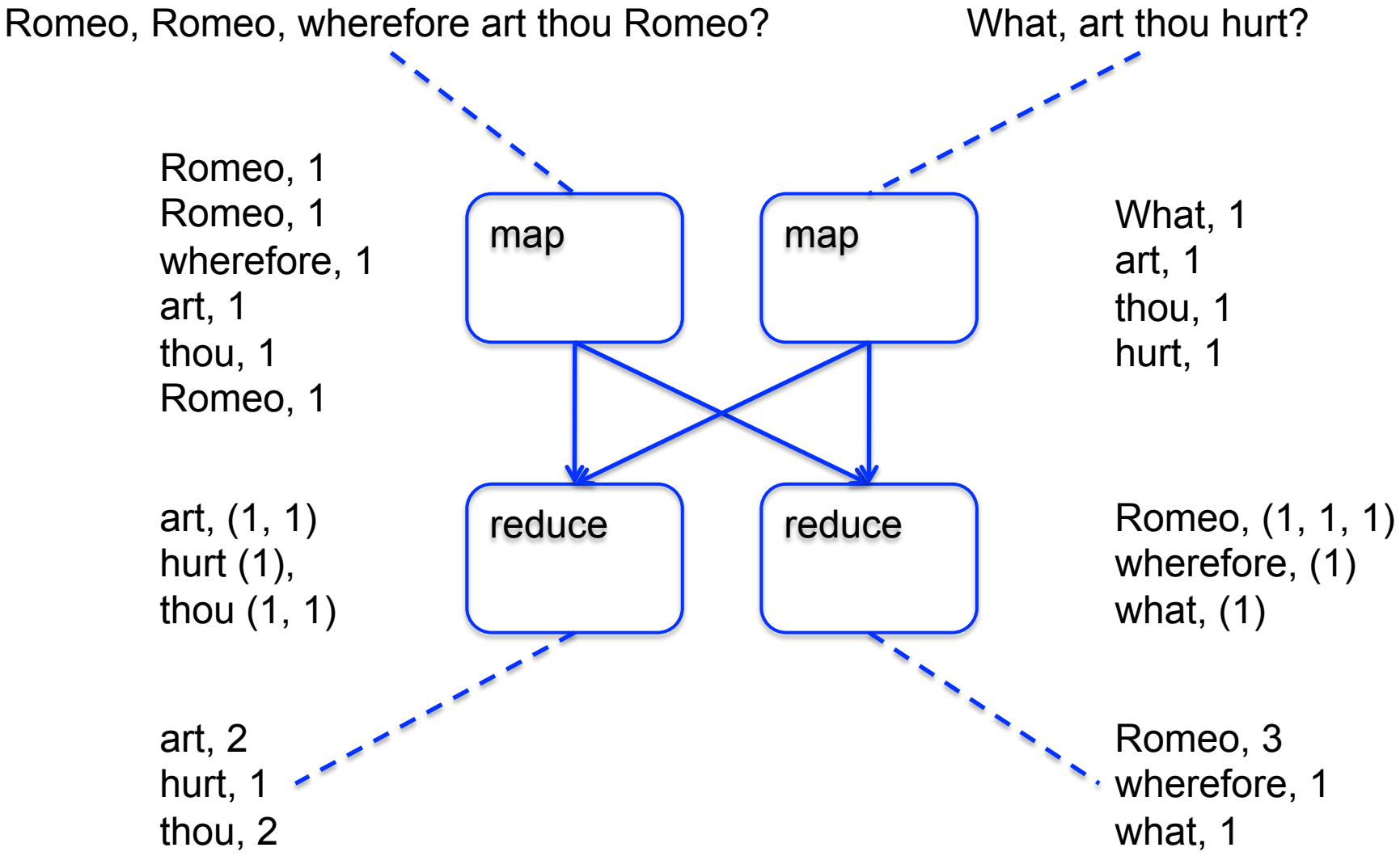


Credit: Alan Gates, Yahoo!

Map Reduce Illustrated



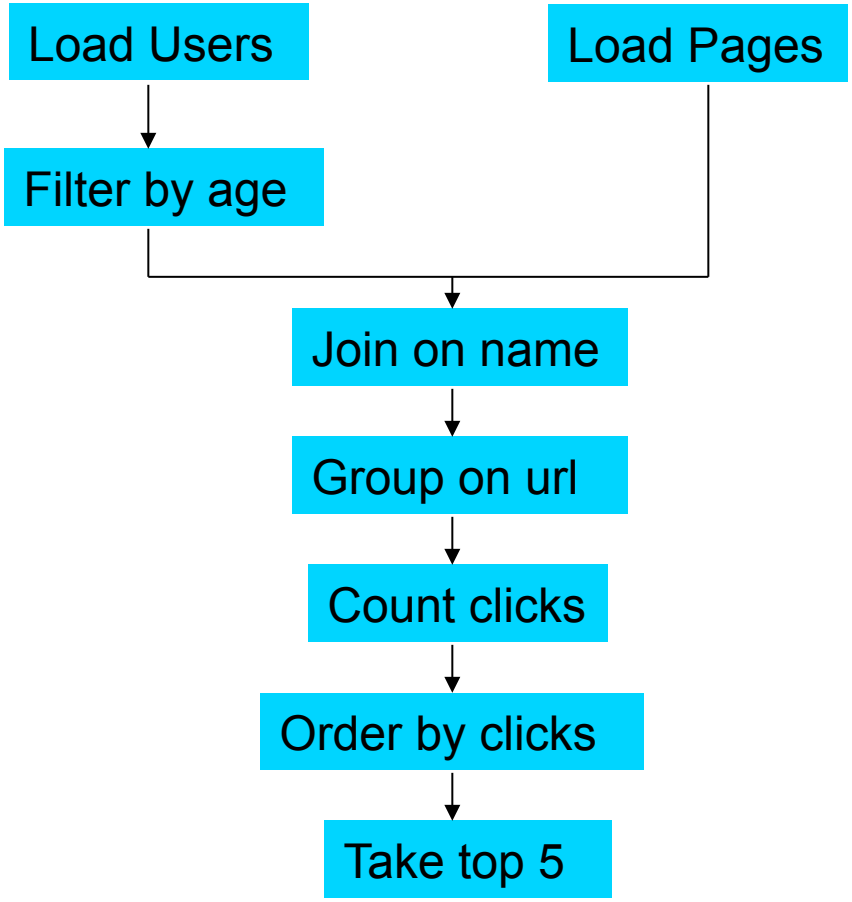
Map Reduce Illustrated



Credit: Alan Gates, Yahoo!

Why use Pig?

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



In Map-Reduce

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
                new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
                new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

       JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

170 lines of code, 4 hours to write

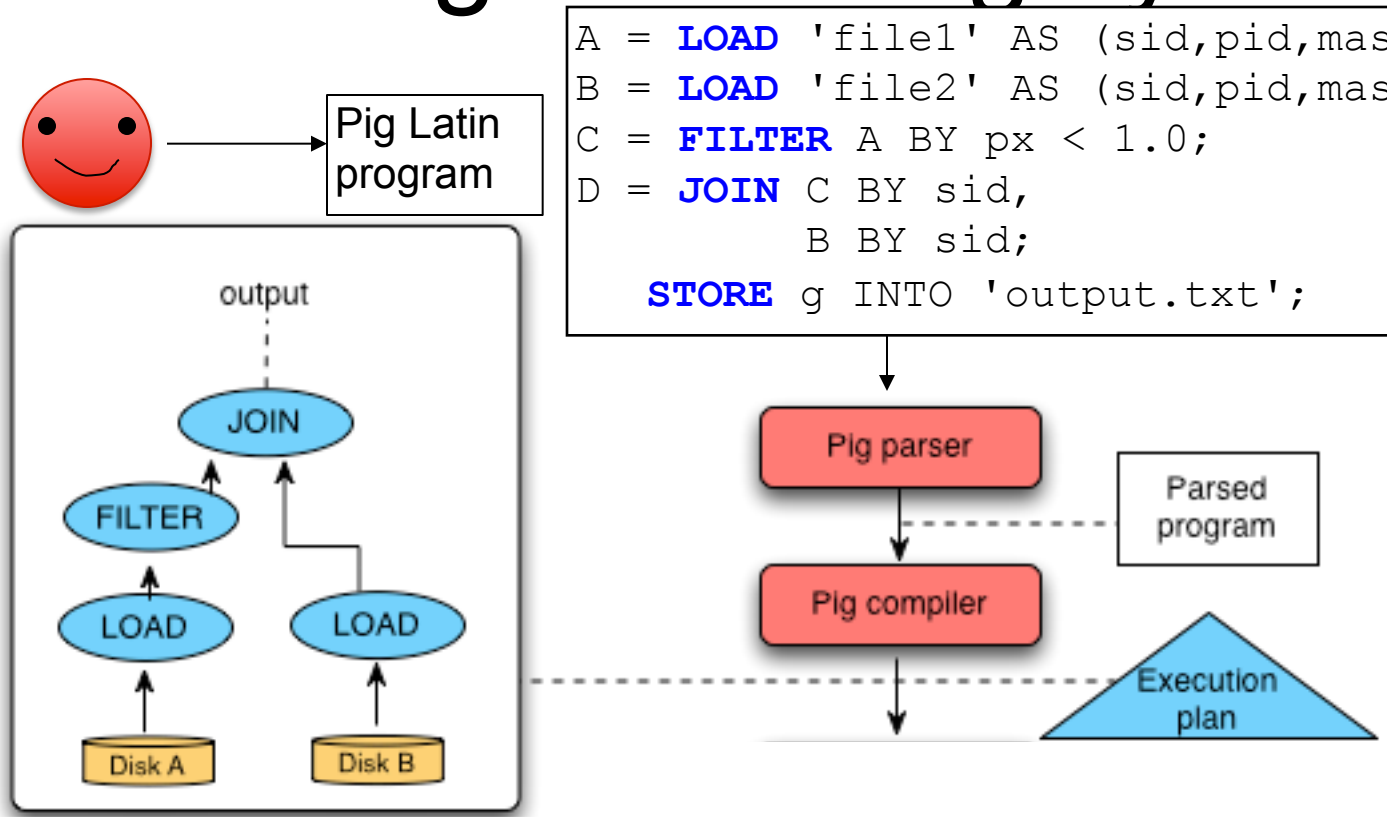
Credit: Alan Gates, Yahoo!

In Pig Latin

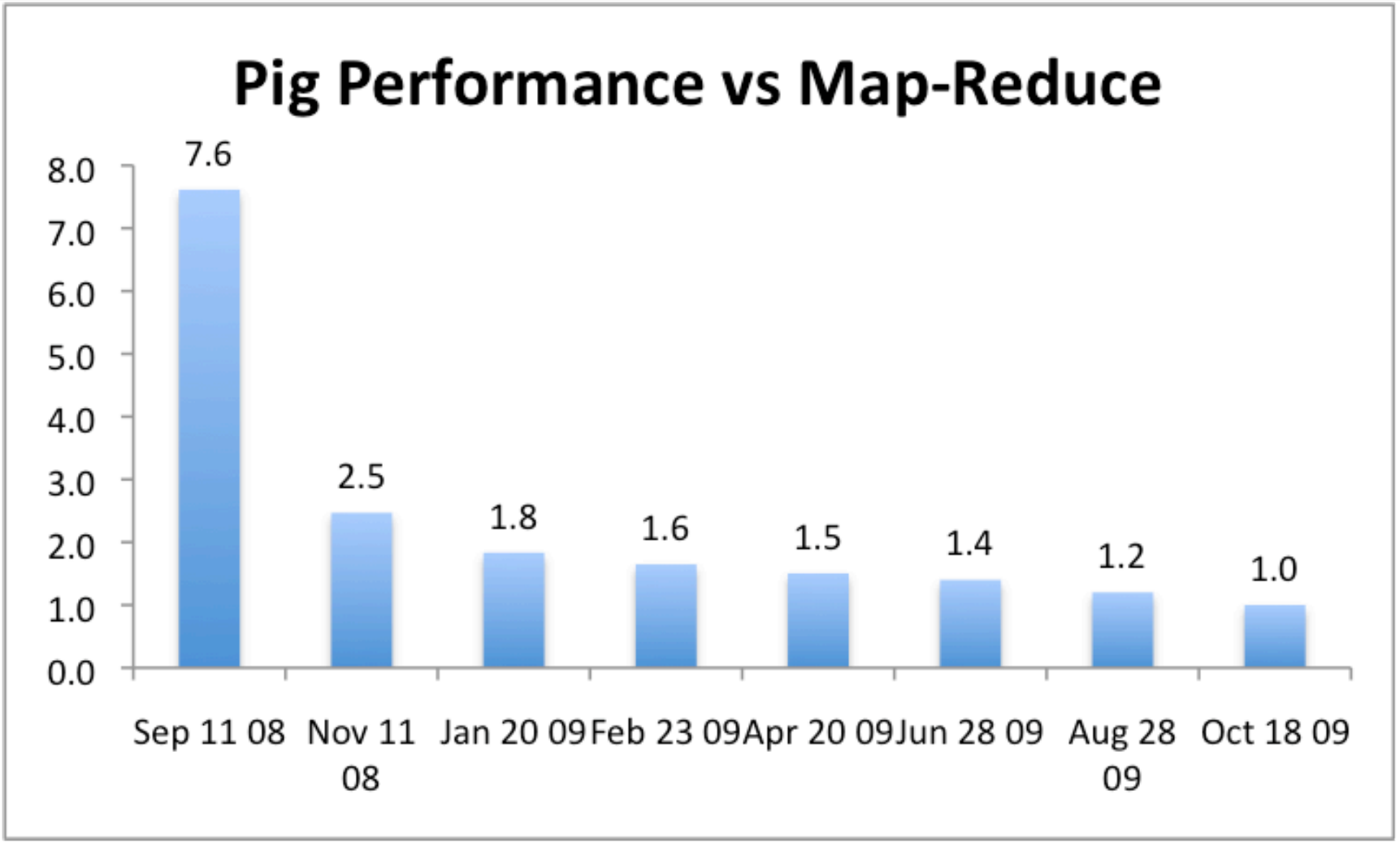
```
Users = load 'users' as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
       COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```

9 lines of code, 15 minutes to write

Background: Pig system



But can it fly?



Credit: Alan Gates, Yahoo!

Essence of Pig

- Map-Reduce is too low a level to program, SQL too high
- Pig Latin, a language intended to sit between the two:
  - Imperative
  - Provides standard relational transforms (join, sort, etc.)
  - Schemas are optional, used when available, can be defined at runtime
  - User Defined Functions are first class citizens
  - Opportunities for advanced optimizer but optimizations by programmer also possible

Credit: Alan Gates, Yahoo!

How It Works

Script

```
A = load
B = filter
C = group
D = foreach
```

Parser

Logical Plan

Semantic Checks

Logical Plan

Logical Optimizer

Logical Plan ≈ relational algebra

Plan standard optimizations

Logical Plan

Logical to Physical Translator

Physical Plan

Physical To MR Translator

Map-Reduce Plan

MapReduce Launcher

Jar to hadoop

Map-Reduce Plan = physical operators broken into Map, Combine, and Reduce stages

Physical Plan = physical operators to be executed

Credit: Alan Gates, Yahoo!

Tenzing

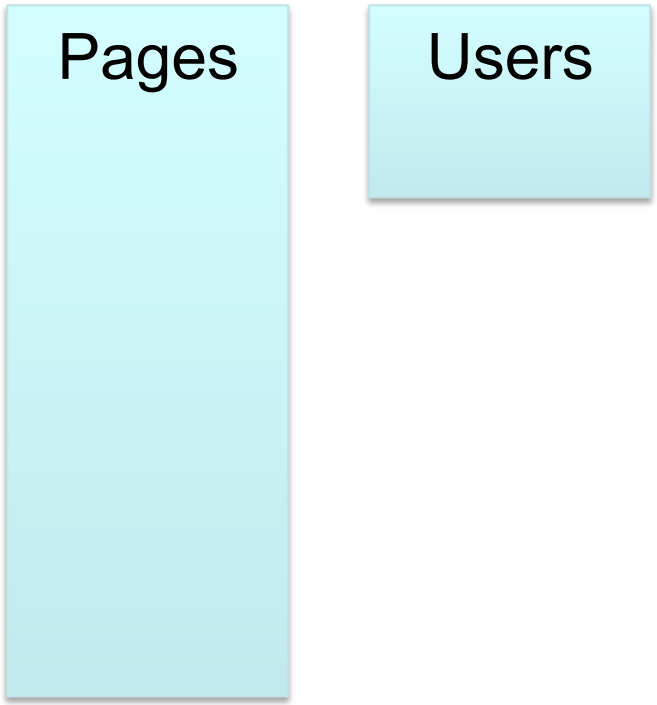
- Google's implementation of SQL
- Supports full SQL92
- On top of google's Map/Reduce
- Uses traditional query optimizer, plus optimizations to MR
- Widely adopted inside Google, especially by the non-engineering community

Join Algorithms on Map/Reduce

- Broadcast join
- Hash-join
- Skew join
- Merge join

Fragment Replicate Join

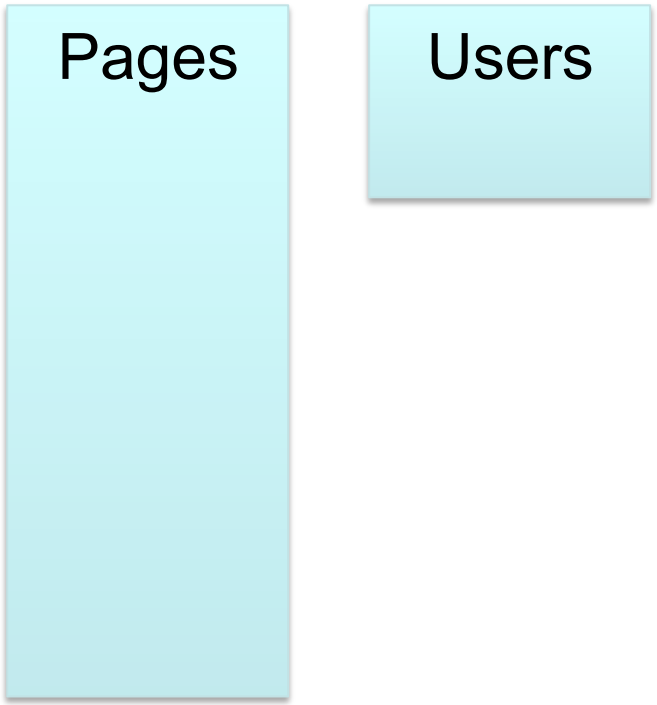
Aka
“Broakdcast Join”



Fragment Replicate Join

Aka
“Broakdcast Join”

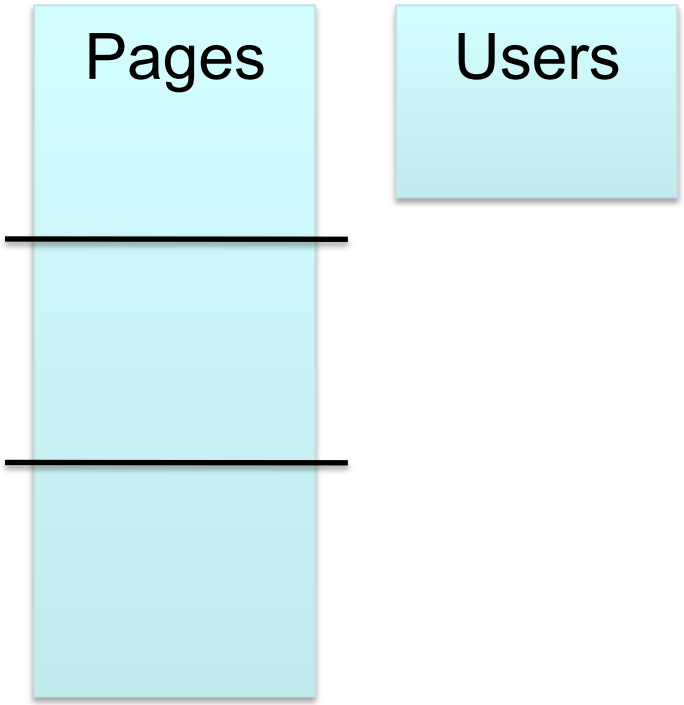
```
Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “replicated”;
```



Fragment Replicate Join

Aka
“Broakdcast Join”

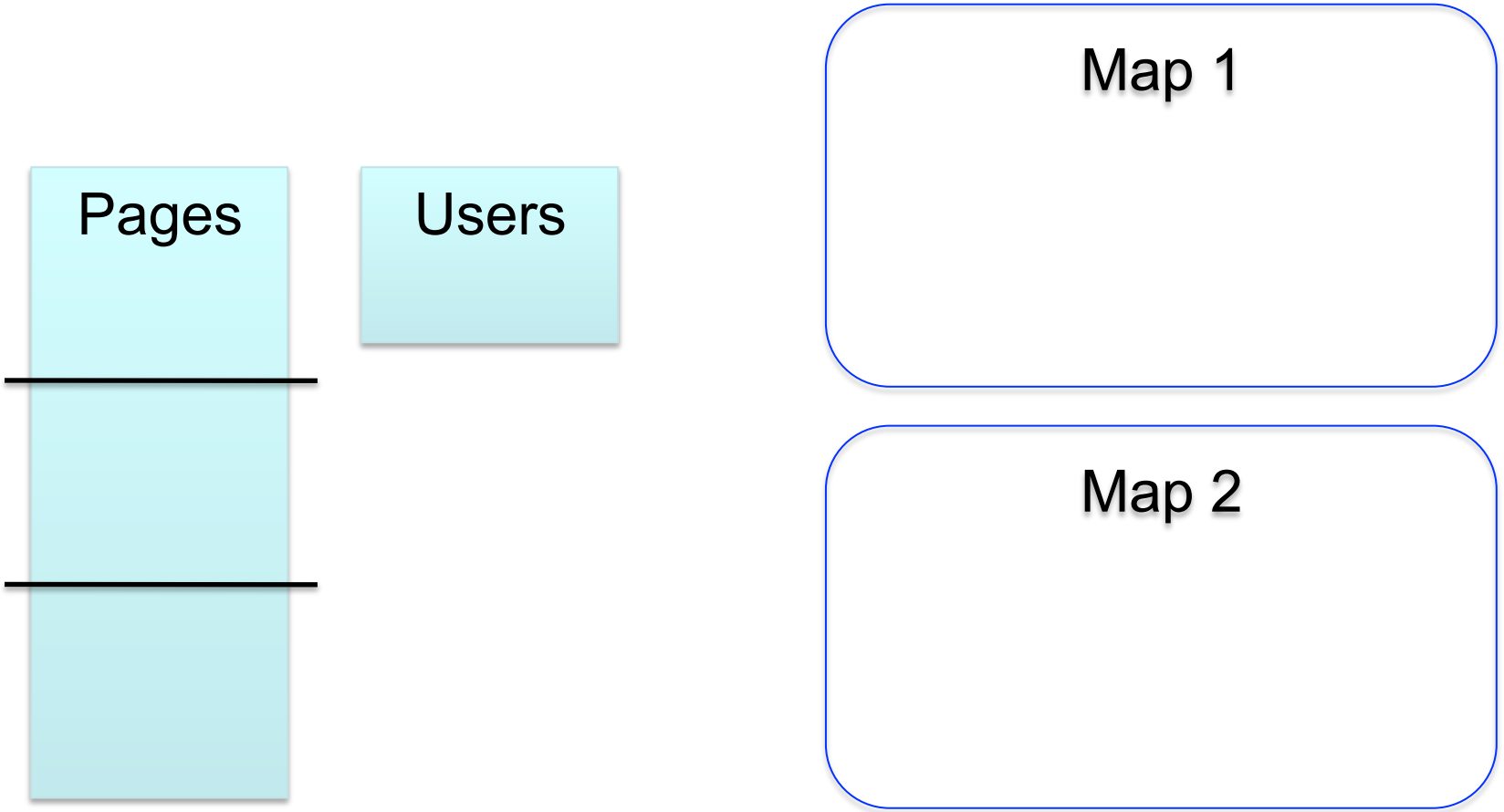
```
Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “replicated”;
```



Fragment Replicate Join

Aka
“Broakdcast Join”

```
Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “replicated”;
```

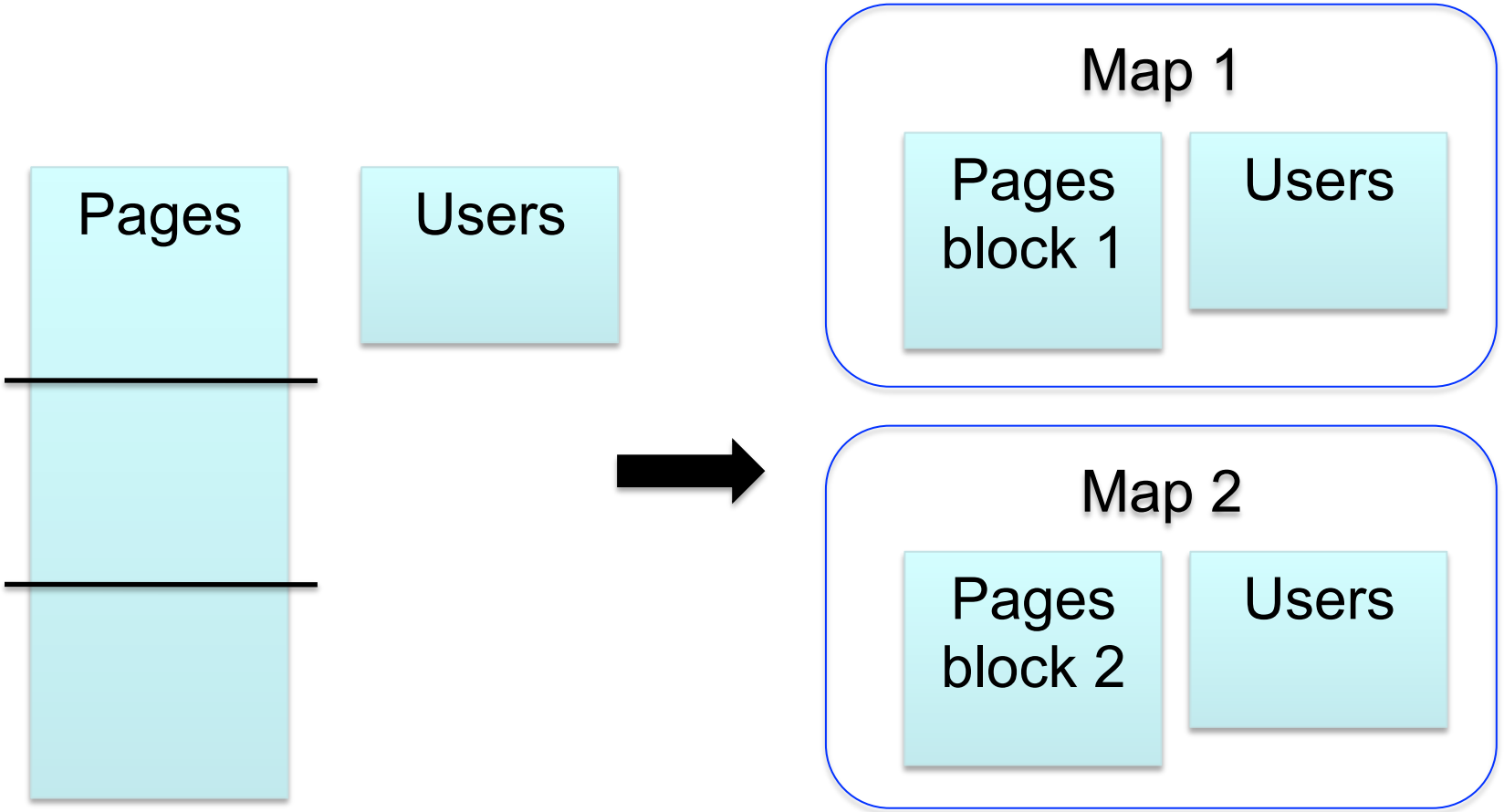


Credit: Alan Gates, Yahoo!

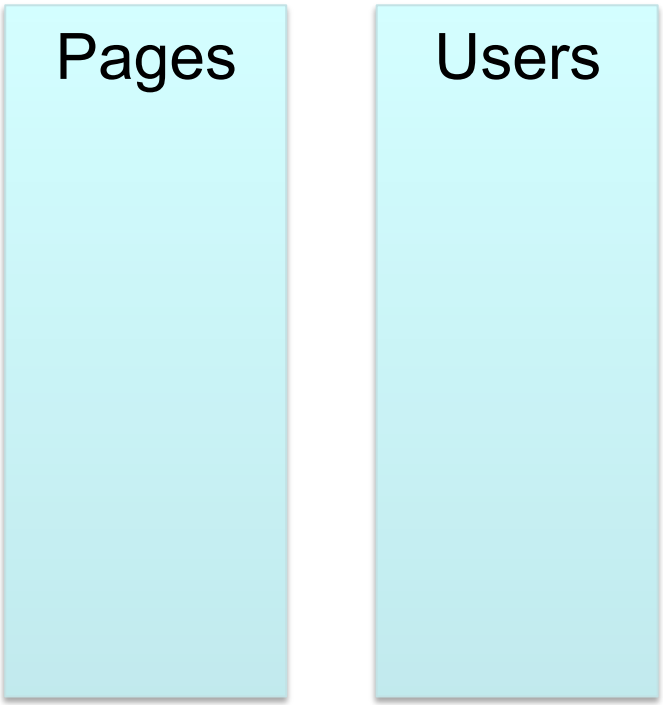
Fragment Replicate Join

Aka
"Broakdcast Join"

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "replicated";
```

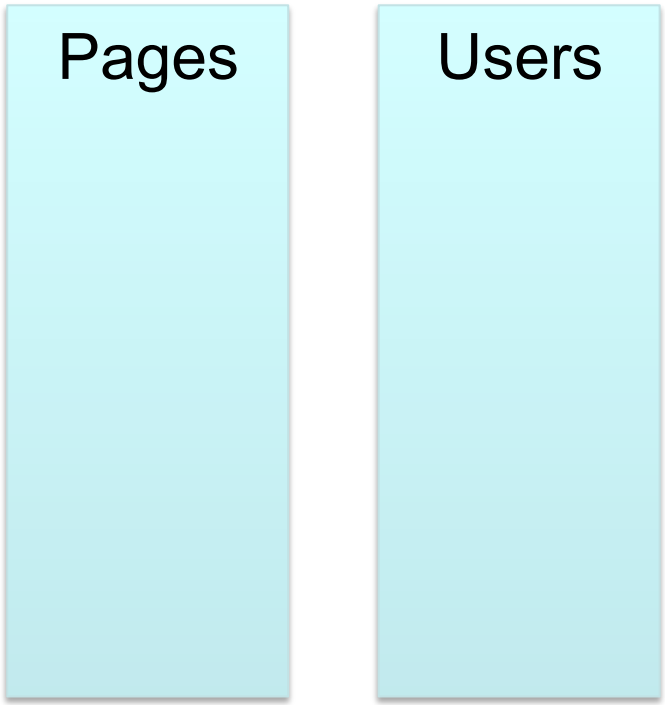


Hash Join



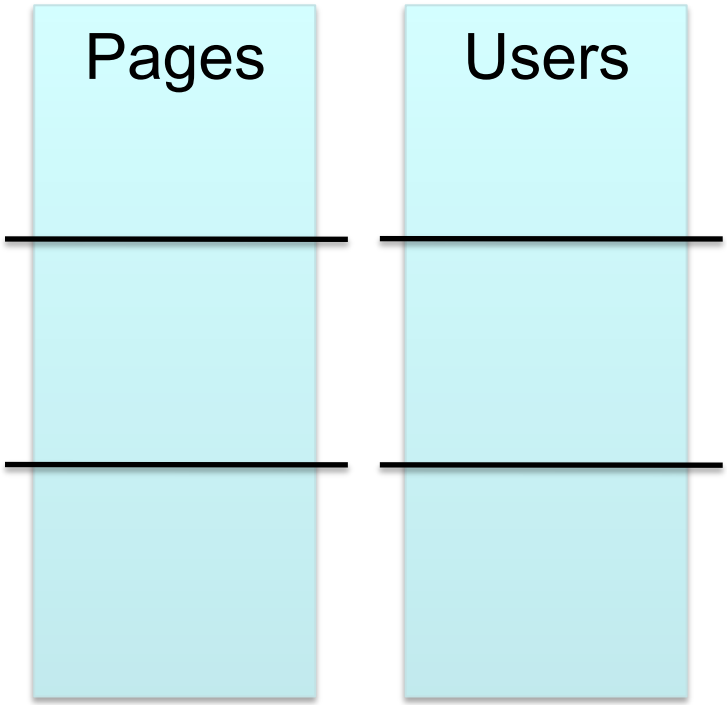
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



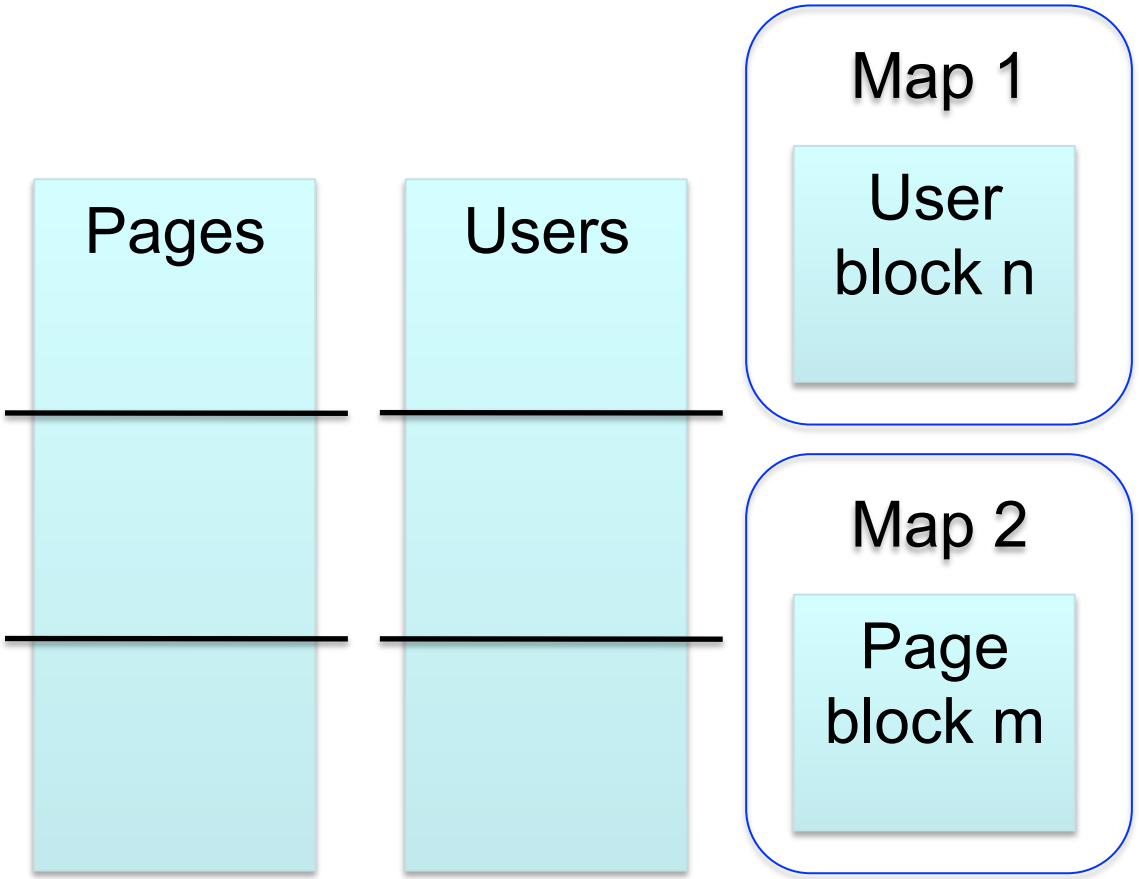
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



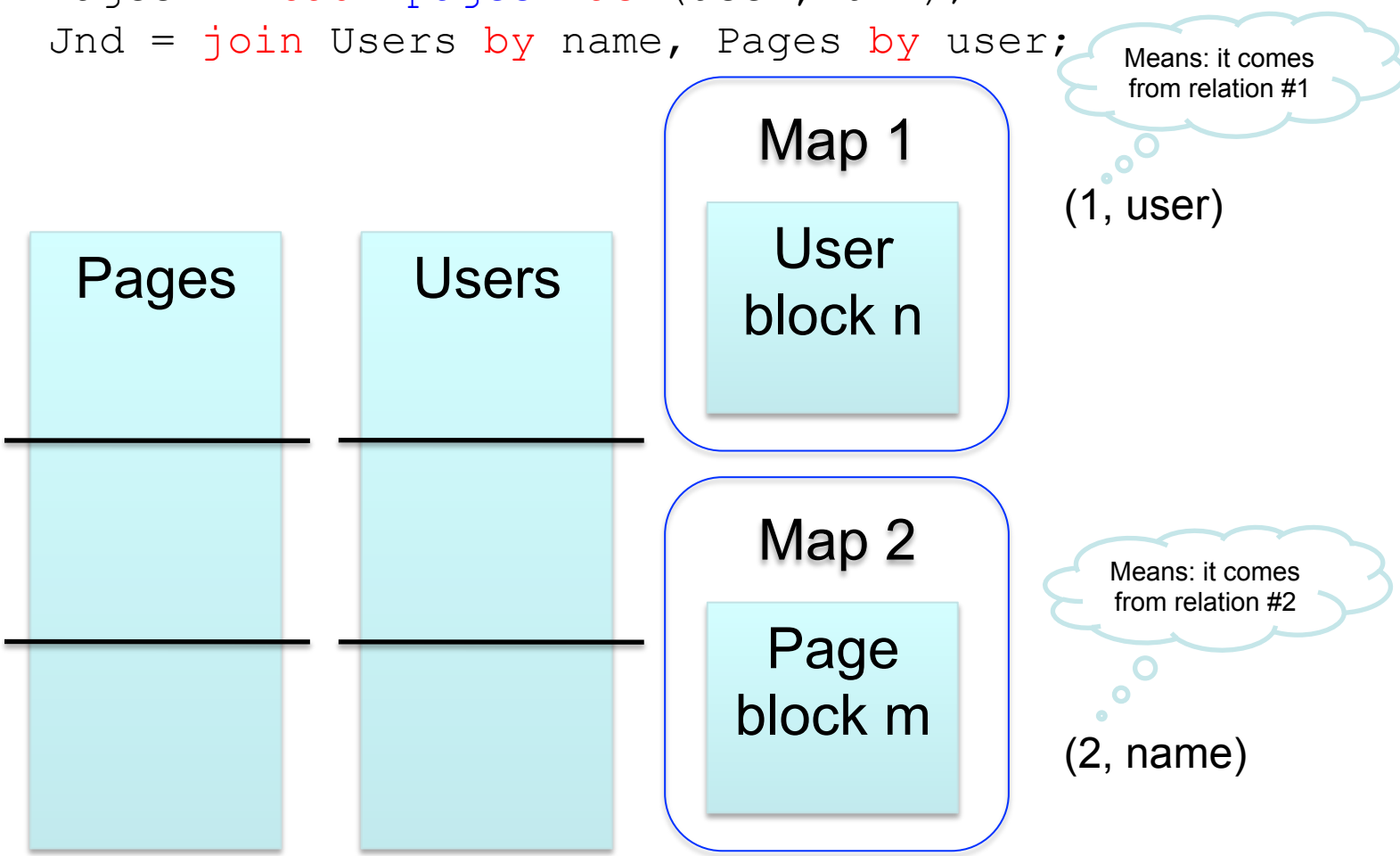
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



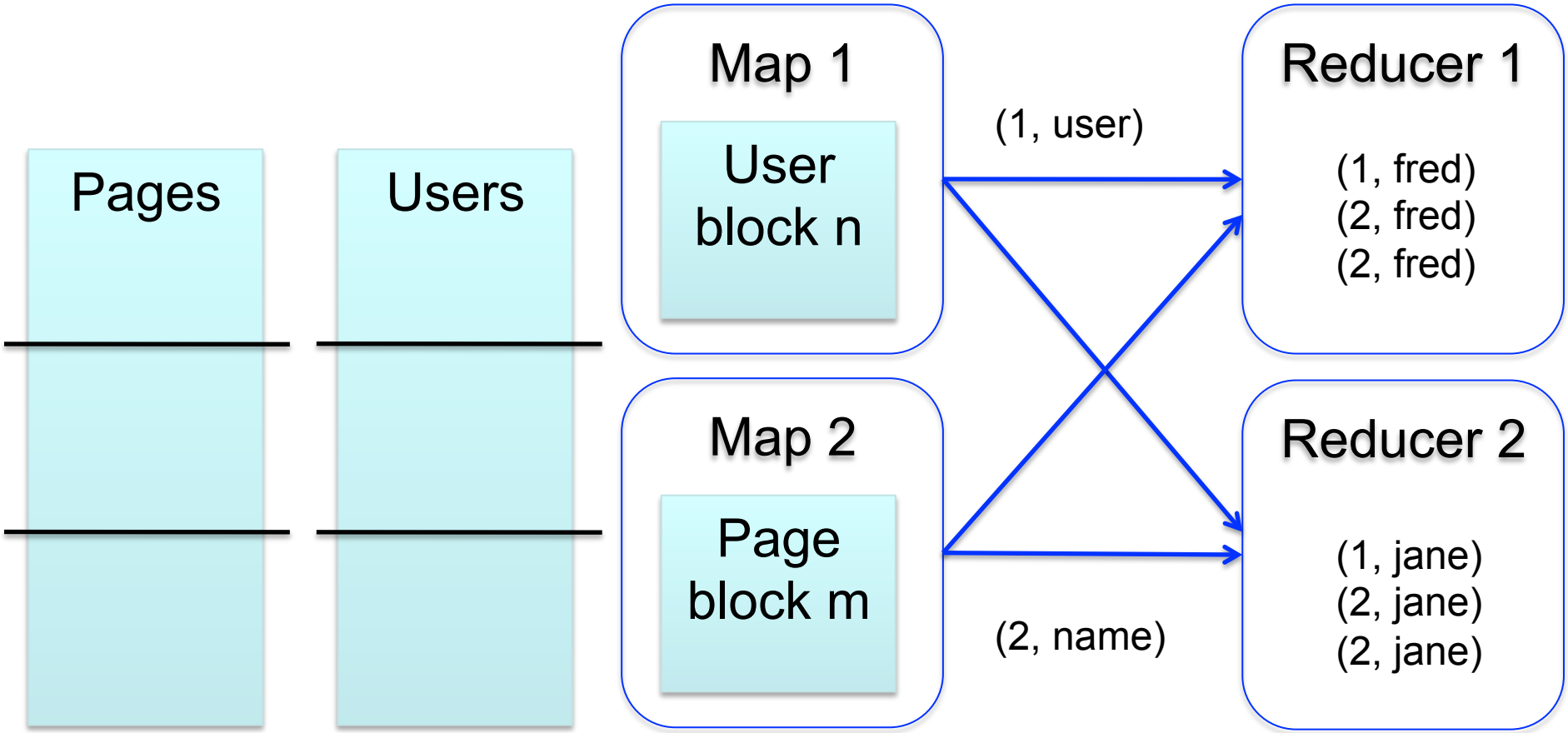
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



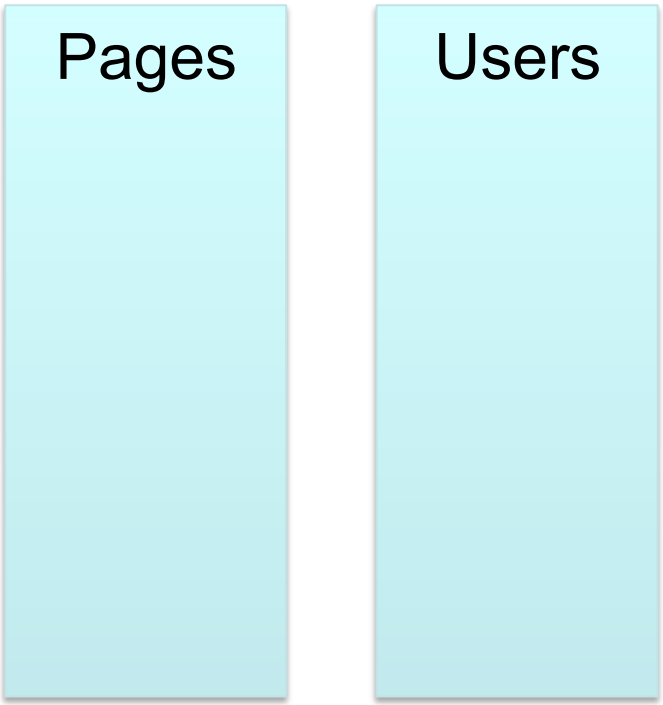
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



Credit: Alan Gates, Yahoo!

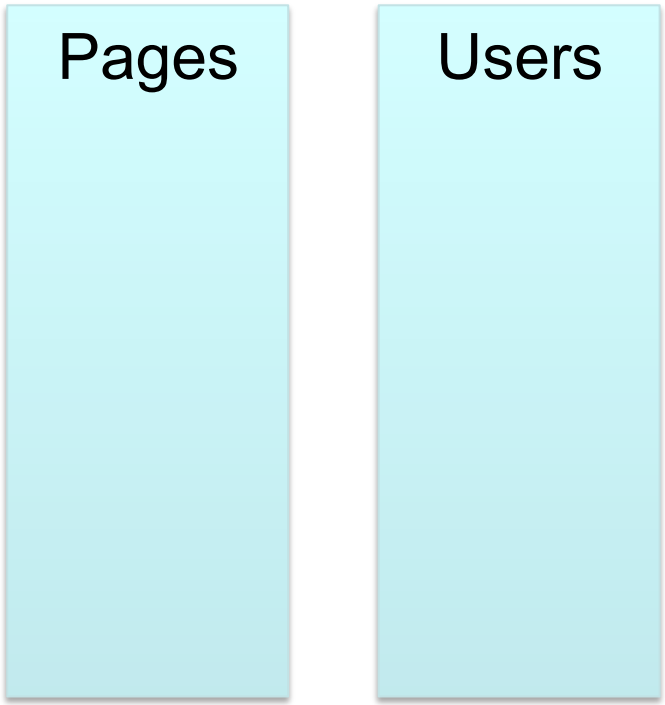
Skew Join



Credit: Alan Gates, Yahoo!

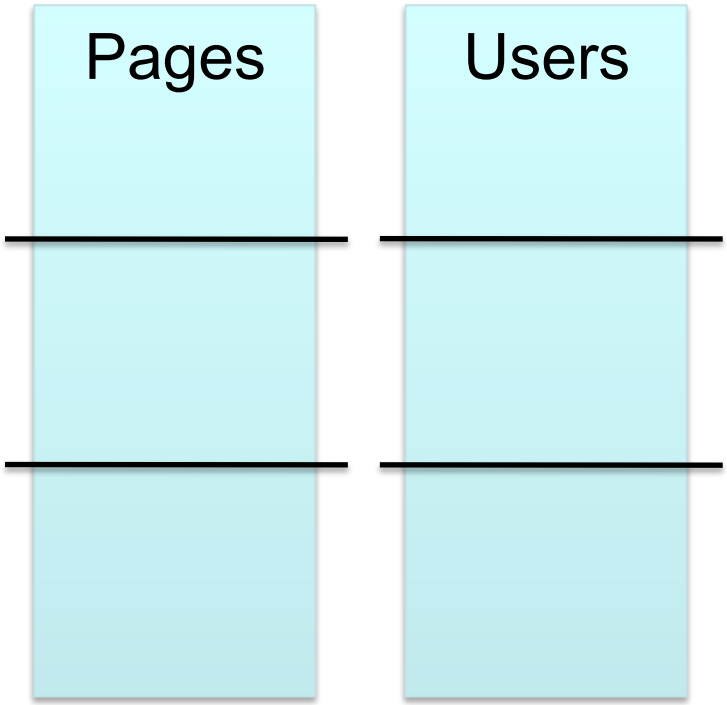
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



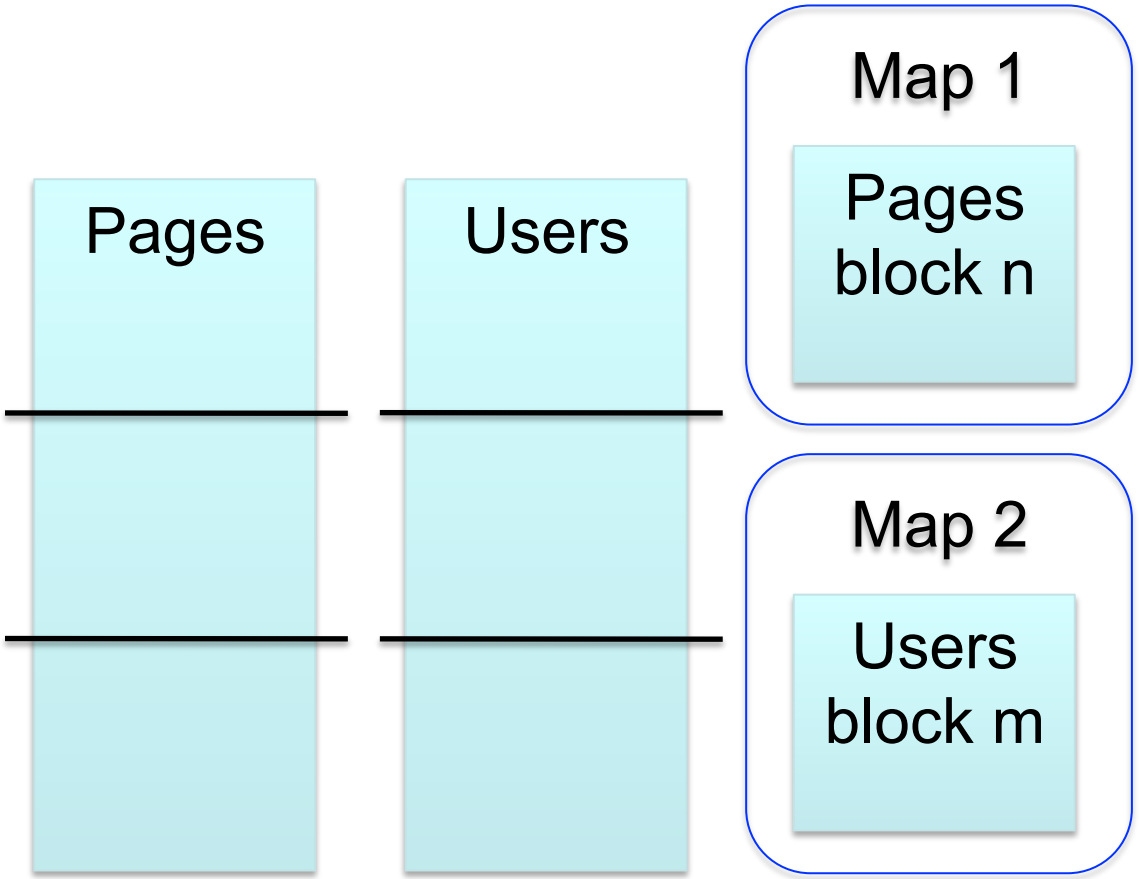
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



Skew Join

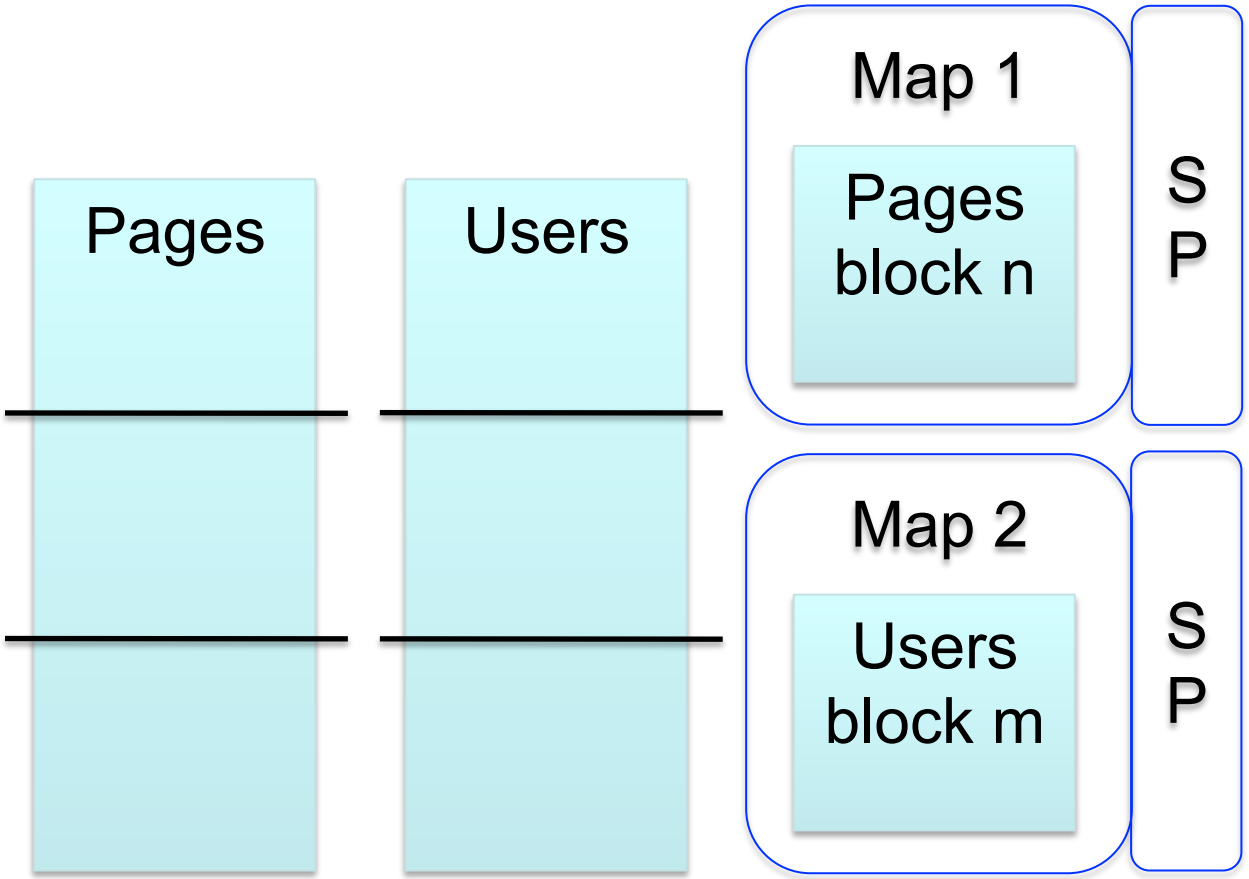
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



Credit: Alan Gates, Yahoo!

Skew Join

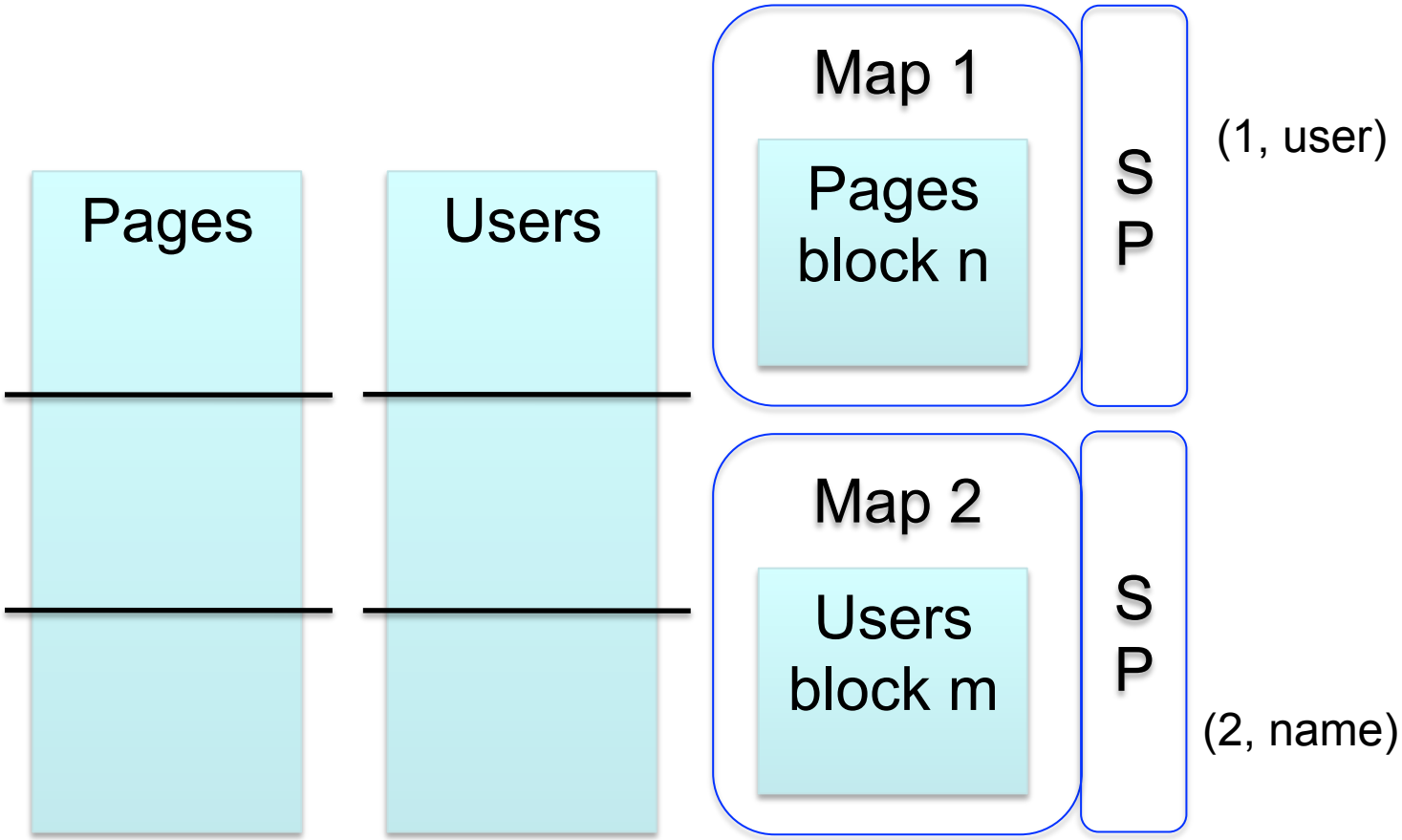
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



Credit: Alan Gates, Yahoo!

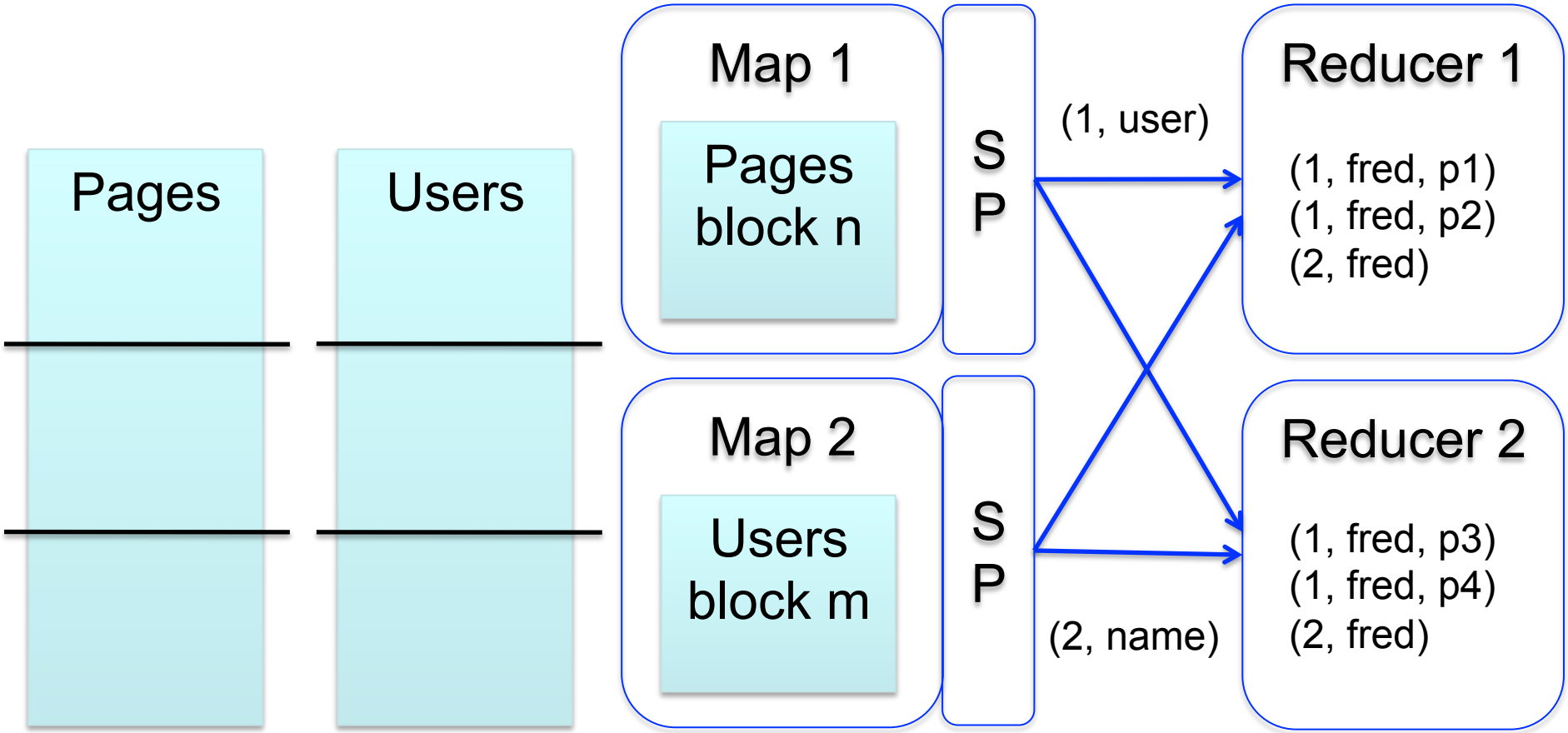
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



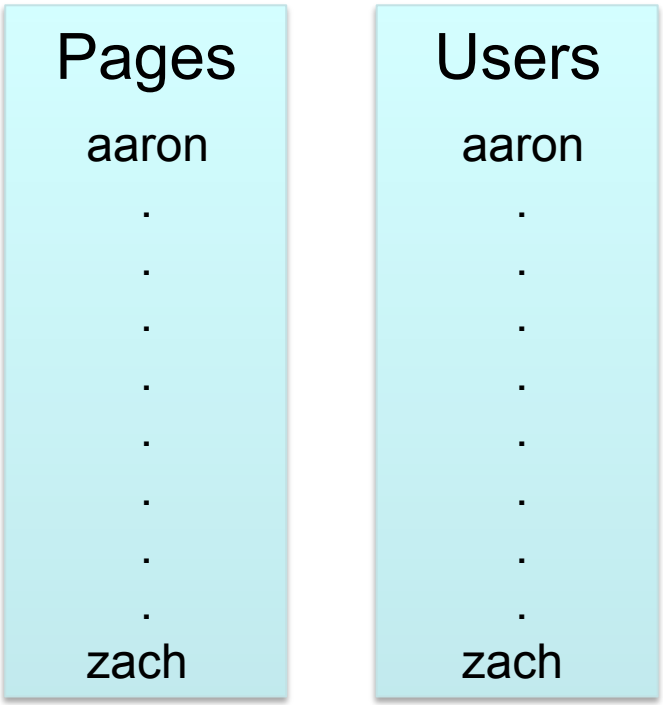
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



Credit: Alan Gates, Yahoo!

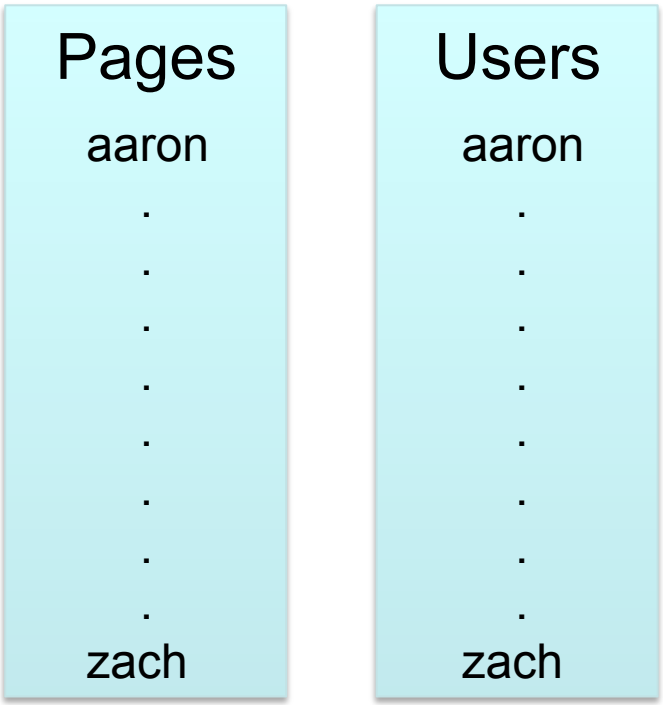
Merge Join



Credit: Alan Gates, Yahoo!

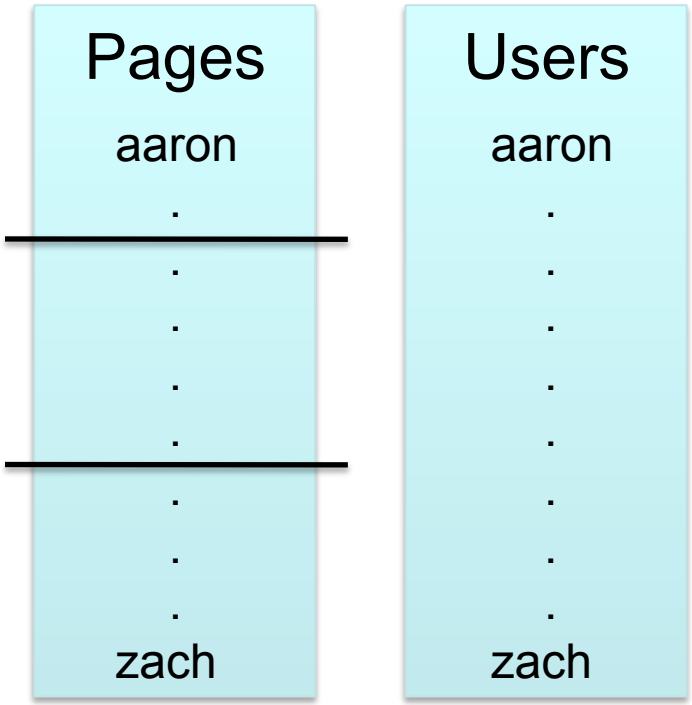
Merge Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "merge";
```



Merge Join

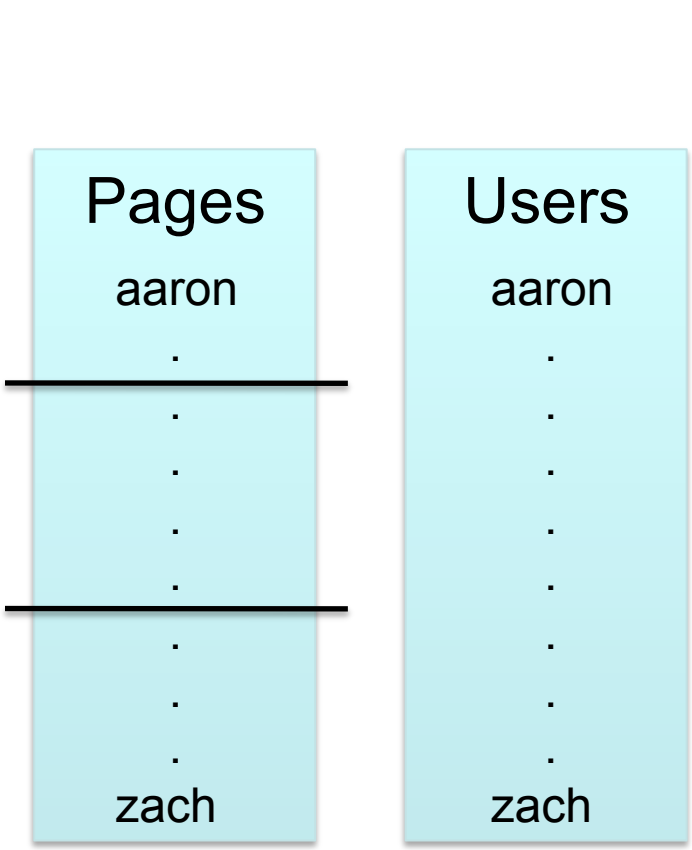
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "merge";
```

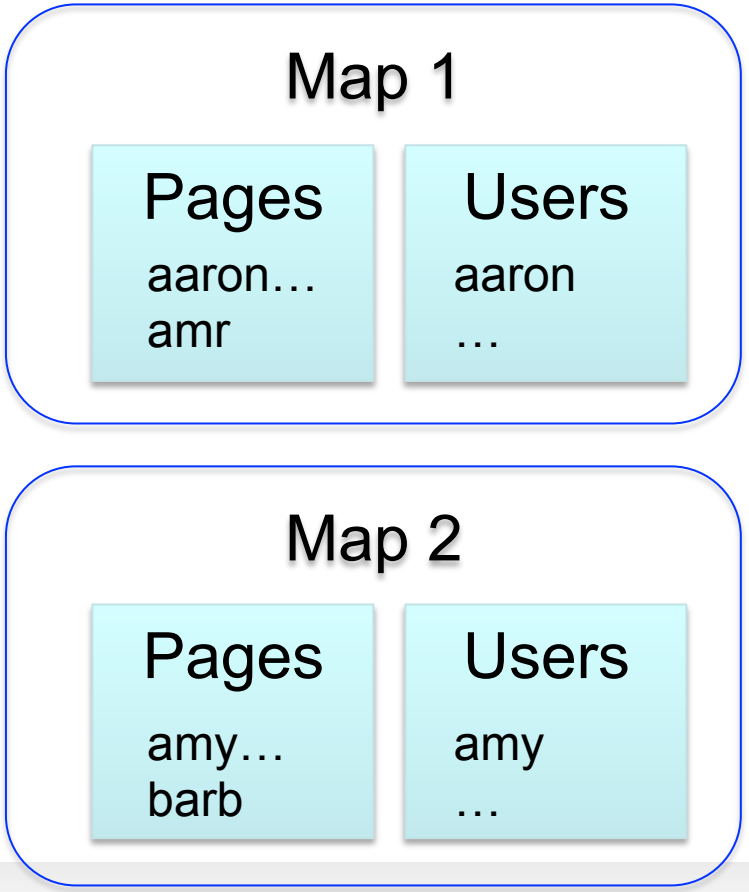


Credit: Alan Gates, Yahoo!

Merge Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "merge";
```

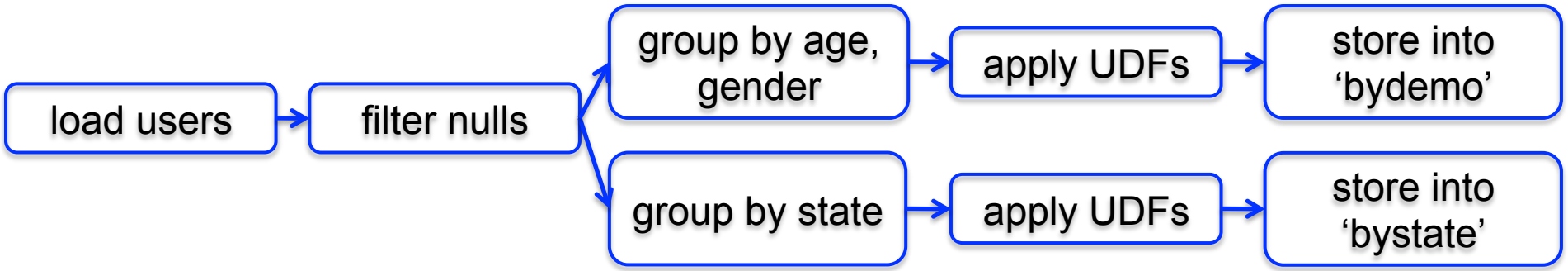




Credit: Alan Gates, Yahoo!

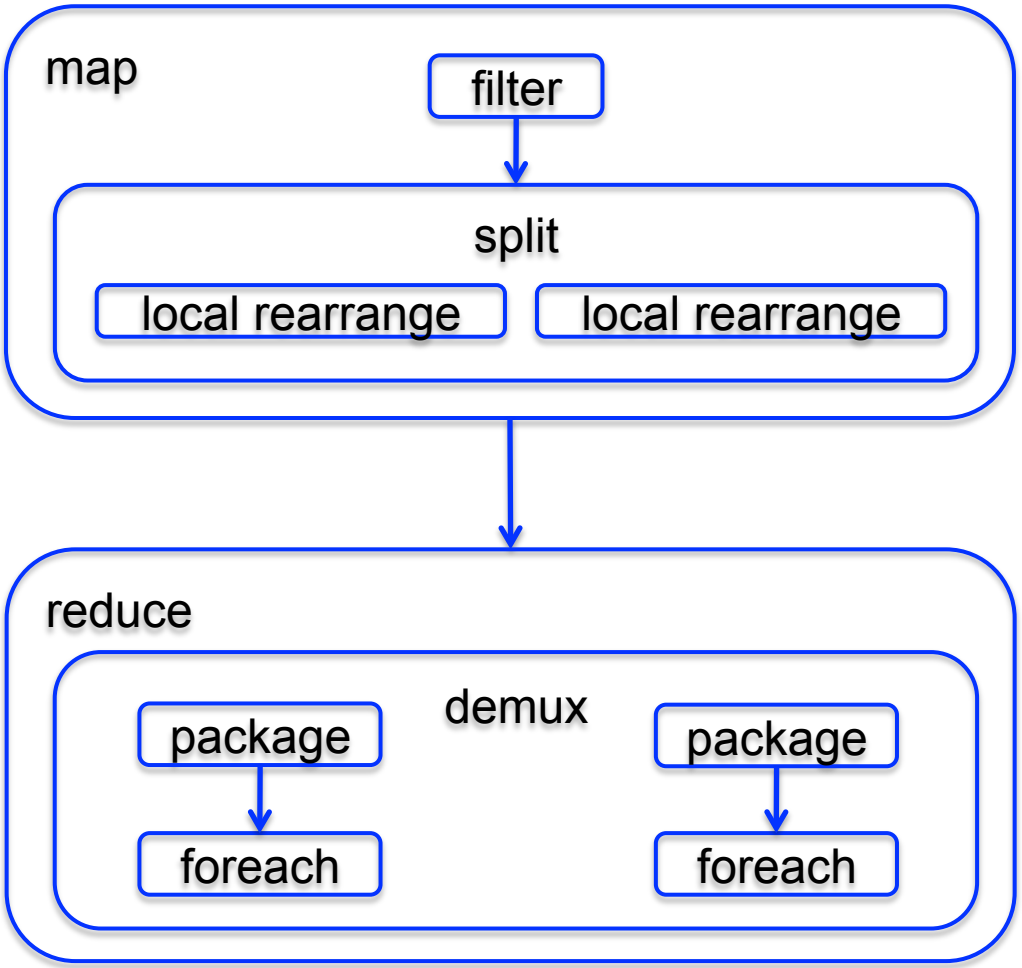
Multi-store script

```
A = load 'users' as (name, age, gender,
      city, state);
B = filter A by name is not null;
C1 = group B by age, gender;
D1 = foreach C1 generate group, COUNT(B);
store D into 'bydemo';
C2= group B by state;
D2 = foreach C2 generate group, COUNT(B);
store D2 into 'bystate';
```



Credit: Alan Gates, Yahoo!

Multi-Store Map-Reduce Plan



Credit: Alan Gates, Yahoo!

Other Optimizations in Tenzing

- Keep processes running: process pool
- Remove reducer-side sort for hash-based algorithms
  - Note: the data must fit in main memory, otherwise the task fails
- Pipelining
- Indexes

Final Thoughts

Challenging problems in MR jobs:

- Skew

- Fault tolerance

Balazinska, *A study of Skew*

Skew

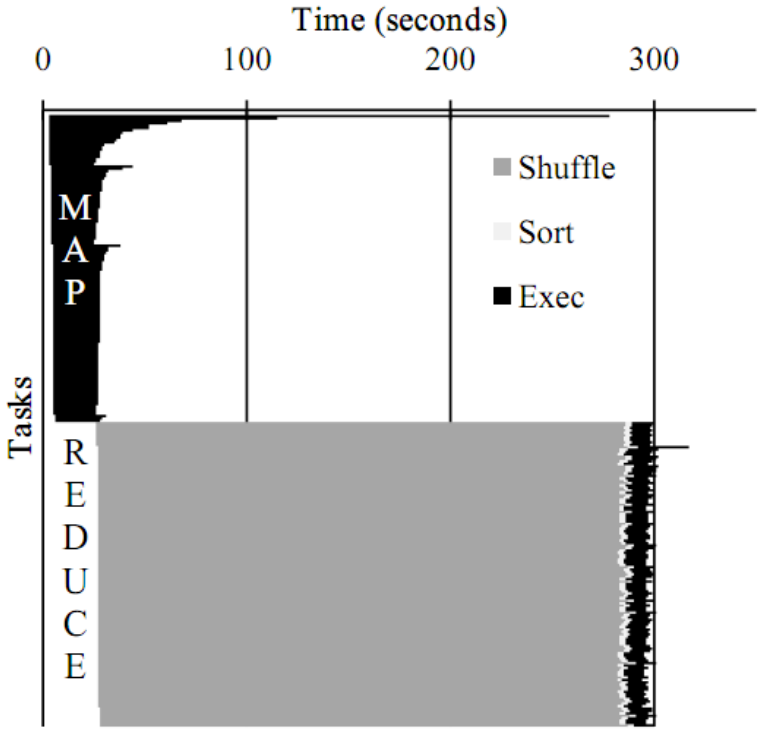
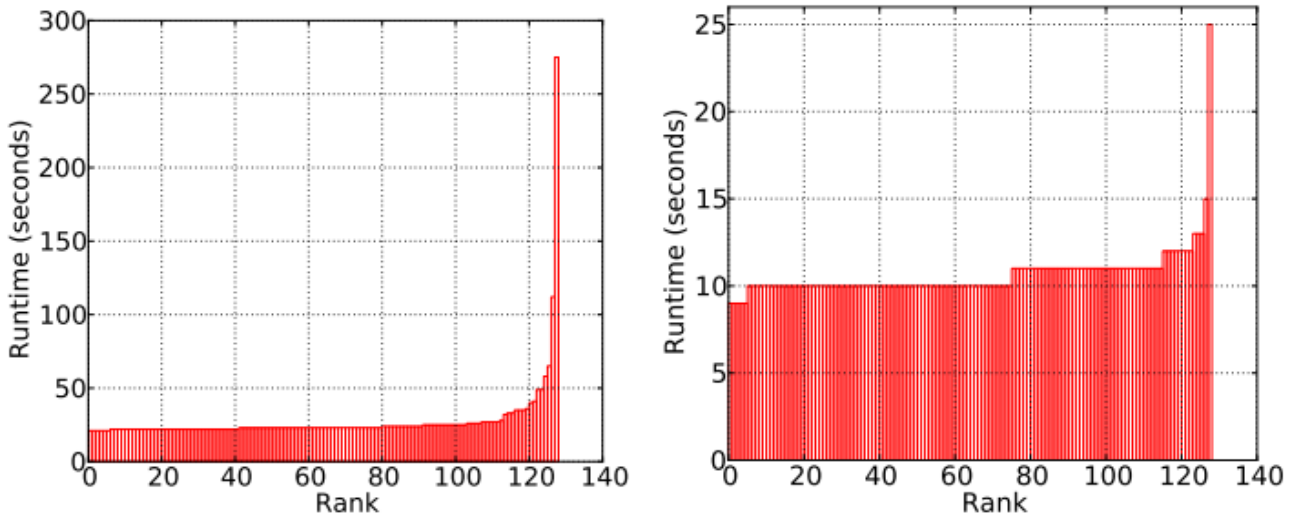


Fig. 1. A timing chart of a MapReduce job running the PageRank algorithm from Cloud 9 [5]. Exec represents the actual map and reduce operations. The slowest map task (first one from the top) takes more than twice as long to complete as the second slowest map task, which is still five times slower than the average. If all tasks took approximately the same amount of time, the job would have completed in less than half the time.

Balazinska, *A study of Skew*

Skew



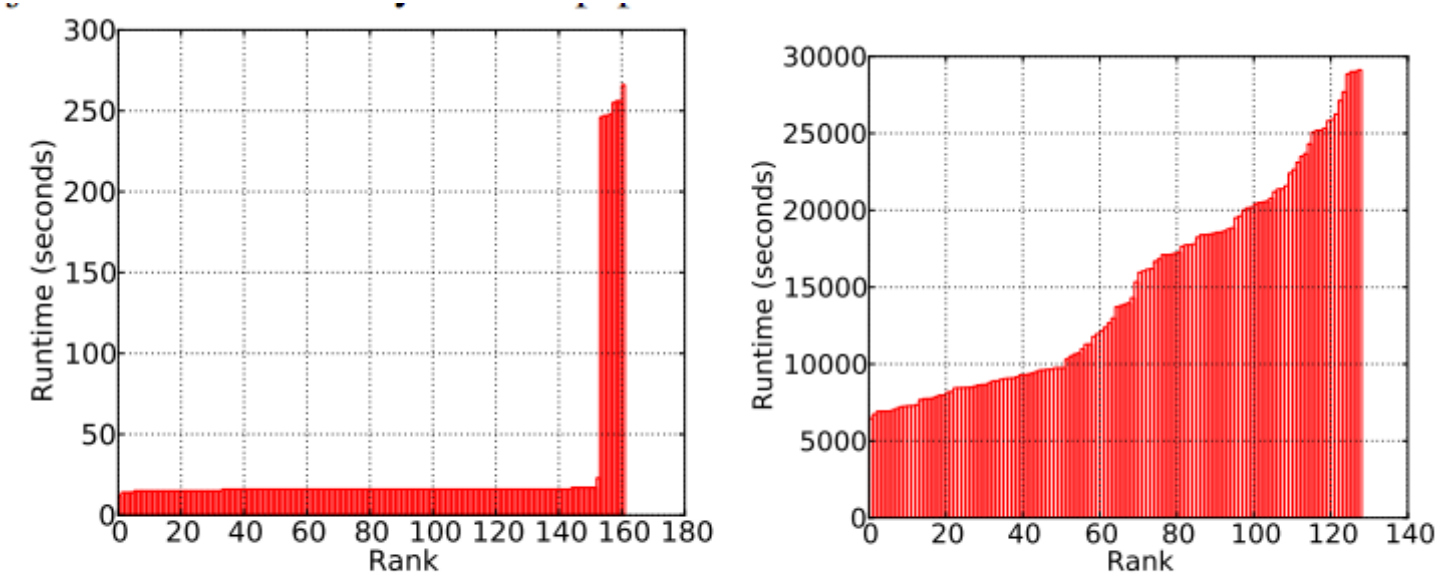
(a) Page Rank - Map

(b) Page Rank - Reduce

Fig. 2. The distribution of task runtimes for PageRank with 128 map and 128 reduce tasks. A graph node with a large number of edges is much more expensive to process than many graph nodes with few edges. Skew arises in both the map and reduce phases, but the overall job is dominated by the map phase.

Balazinska, *A study of Skew*

Skew



(a) CloudBurst - Map (b) CloudBurst - Reduce

Fig. 3. Distribution of task runtime for CloudBurst. Total 162 map tasks, and 128 reduce tasks. The map phase exhibits a bimodal distribution. Each mode corresponds to map tasks processing a different input dataset. The reduce is computationally expensive and has a smooth runtime distribution, but there is a factor of five difference in runtime between the fastest and the slowest reduce tasks.

Fault Tolerance

- Fundamental tension:
- Materialize after each Map and each Reduce
  - This is what MR does
  - Ideal for fault tolerance
  - Very poor performance
- Pipeline between steps
  - This is what Parallel DBs usually do
  - Ideal for performance
  - Very poor fault tolerance

Pig Latin Mini-Tutorial

(will skip in class; please read in order to do homework 6)

Outline

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

Quiz section tomorrow: in CSE 403 (this is CSE, don't go to EE1)

Pig-Latin 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

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)
```

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

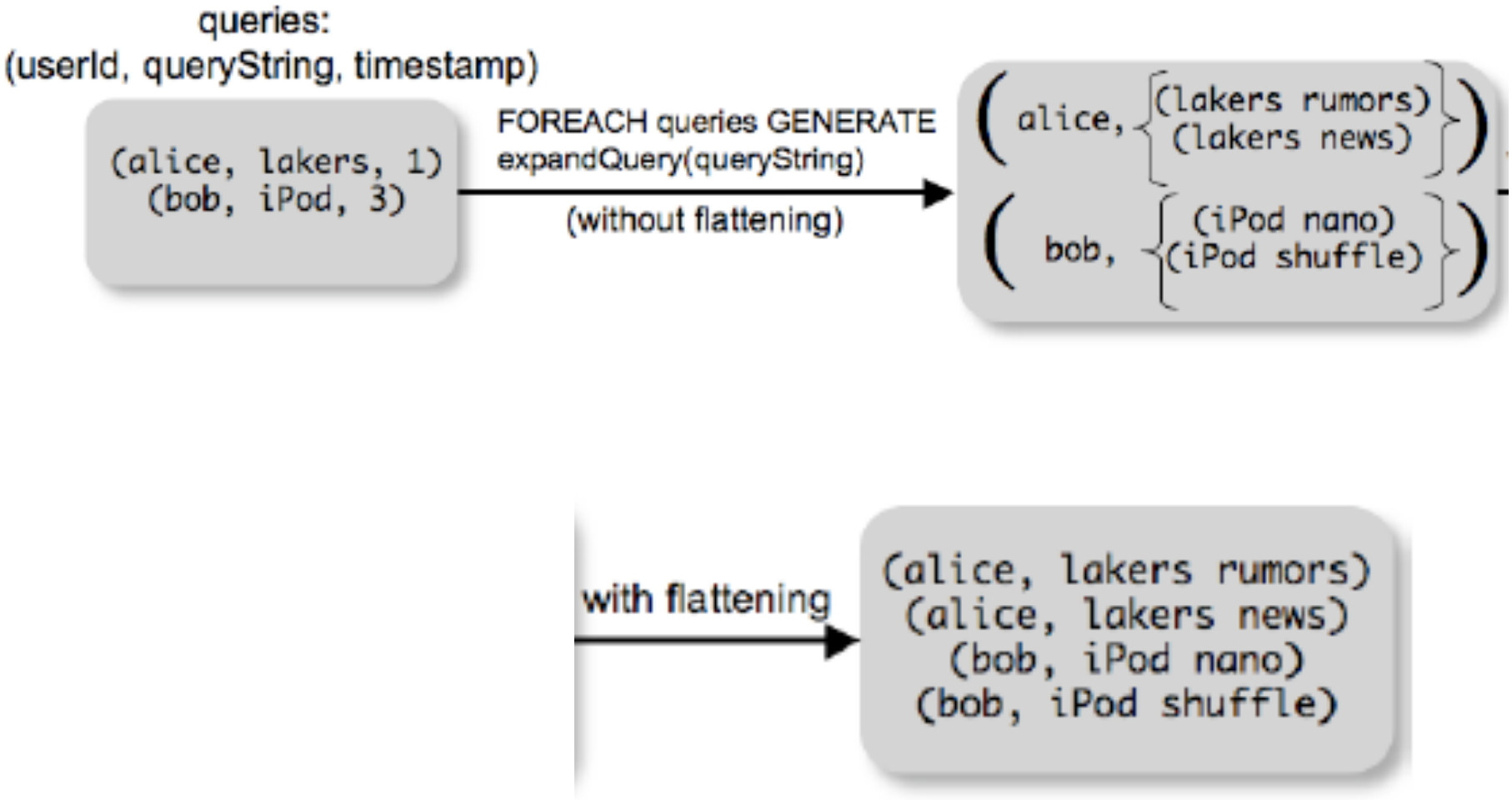
```
expanded_queries =
   FOREACH queries
   GENERATE userId, expandQuery(queryString)
```

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 !

(that's one thing I don't like about Pig-latin)

- 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:

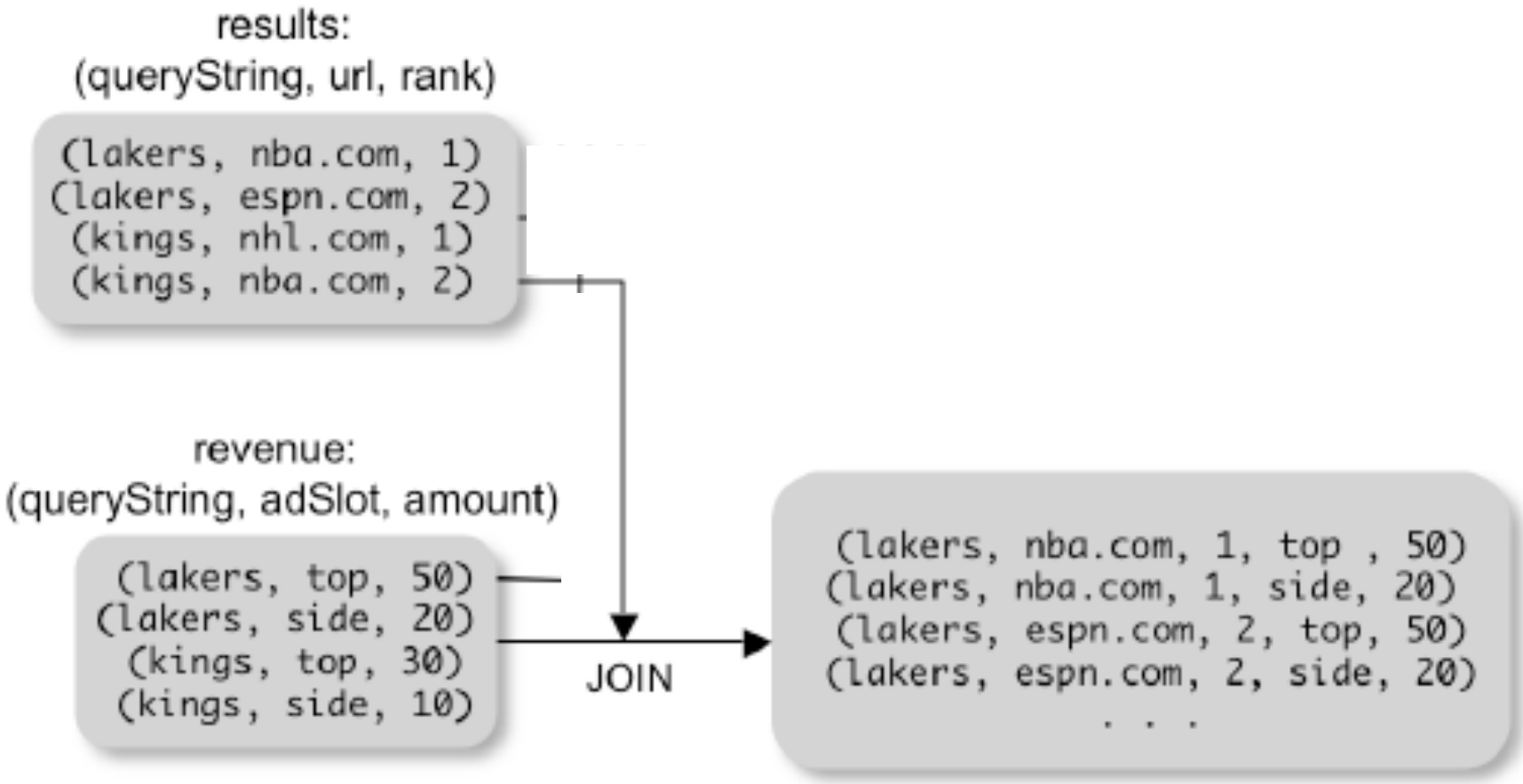
```
real_queries =  FILTER queries
                BY NOT isBot(userId)
```

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)}

Simple Map-Reduce

input : {(field1, field2, field3, ...)}

```
map_result = FOREACH input
                GENERATE FLATTEN(map(*))
key_groups = GROUP map_result BY $0
output = FOREACH key_groups
          GENERATE reduce($1)
```

map_result : {(a1, a2, a3, . . .)}
key_groups : {(a1, {(a2, a3, . . .)})}

Co-Group

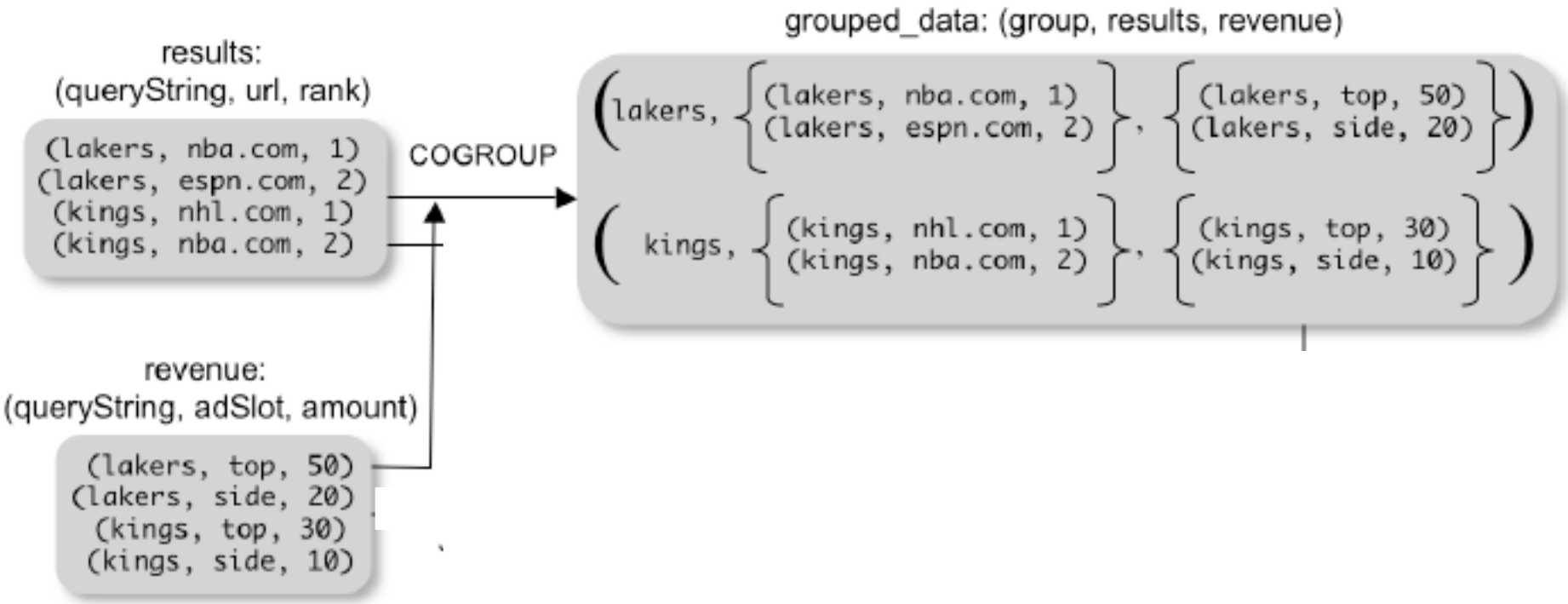
results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

```
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 ?

Co-Group



Is this an inner join, or an outer join ?

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’

Implementation

- Over Hadoop !
- Parse query:
  - Everything between LOAD and STORE → one logical plan
- Logical plan → sequence of Map/ Reduce ops
- All statements between two (CO) GROUPs → one Map/Reduce op

Implementation