CSEP 544: Lecture 06

Parallel DB and MR,

Transactions Part 1 (Recovery)

Outline

- Finish parallel databases and MapReduce
- Begin transactions

Big Data

- Gartner report*
  - High Volume
  - High Variety
  - High Velocity
- Stonebraker:
  - Big volumes, small analytics
  - Big analytics, on big volumes
  - Big velocity
  - Big variety

* http://www.gartner.com/newsroom/id/1731916

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?

Announcement

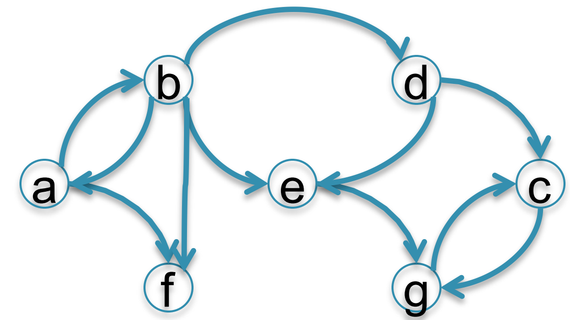
- Homework 3 (AWS) due this Friday!

- Remember to turn your instances off!

Graph Databases

Many large databases are graphs

- Give examples in class

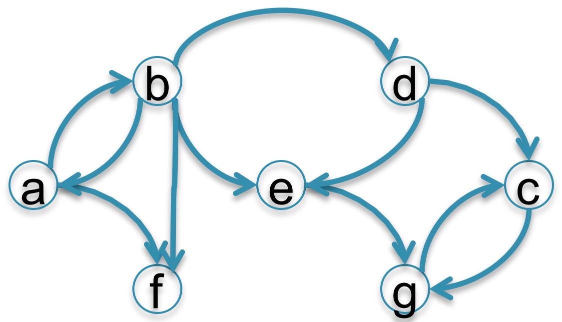


Source	Target
a	b
b	a
a	f
b	f
b	e
b	d
d	e
d	c
e	g
g	c
c	g

Graph Databases

Many large databases are graphs

- Give examples in class
- The Web
- The Internet
- Social Networks
- Flights between airports
- Etc.

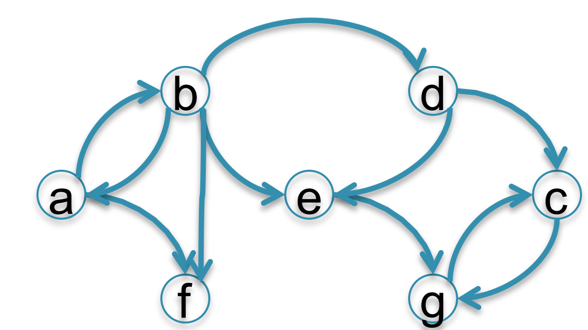


Source	Target
a	b
b	a
a	f
b	f
b	e
b	d
d	e
d	c
e	g
g	c
c	g

Data Analytics on Big Graphs

Queries expressible in SQL:

- How many nodes (edges)?
- How many nodes have > 4 neighbors?
- Which are "most connected nodes"?



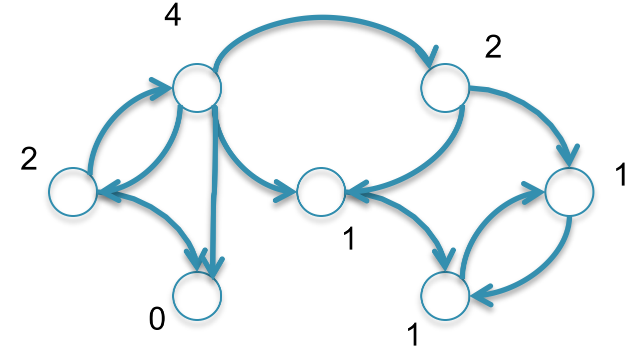
Source	Target
a	b
b	a
a	f
b	f
b	e
b	d
d	e
d	c
e	g
g	c
c	g

Queries requiring recursion:

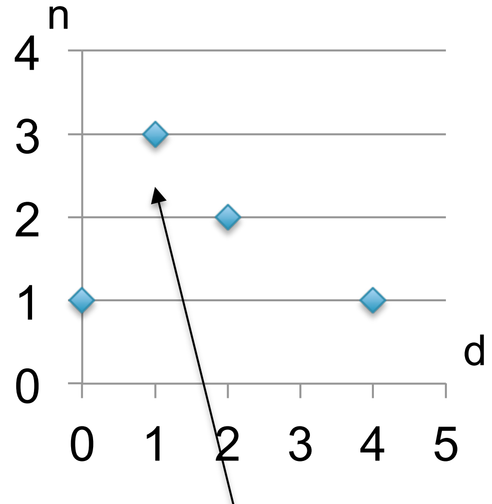
- Is the graph connected?
- What is the diameter of the graph?
- Compute *PageRank*
- Compute the *Centrality* of each node

Example: the 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))

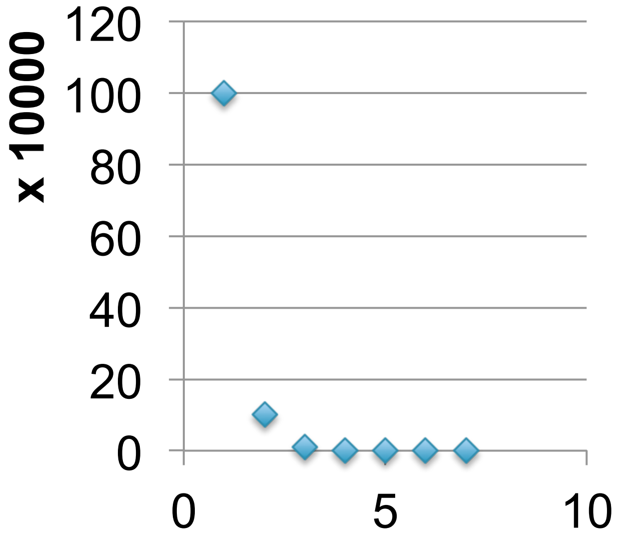


d	n(d)
0	1
1	3
2	2
3	0
4	1



Outdegree 1 is seen at 3 nodes

Histograms Tell Us Something About the Graph



x 10000
120
100
80
60
40
20
0
0
5
10

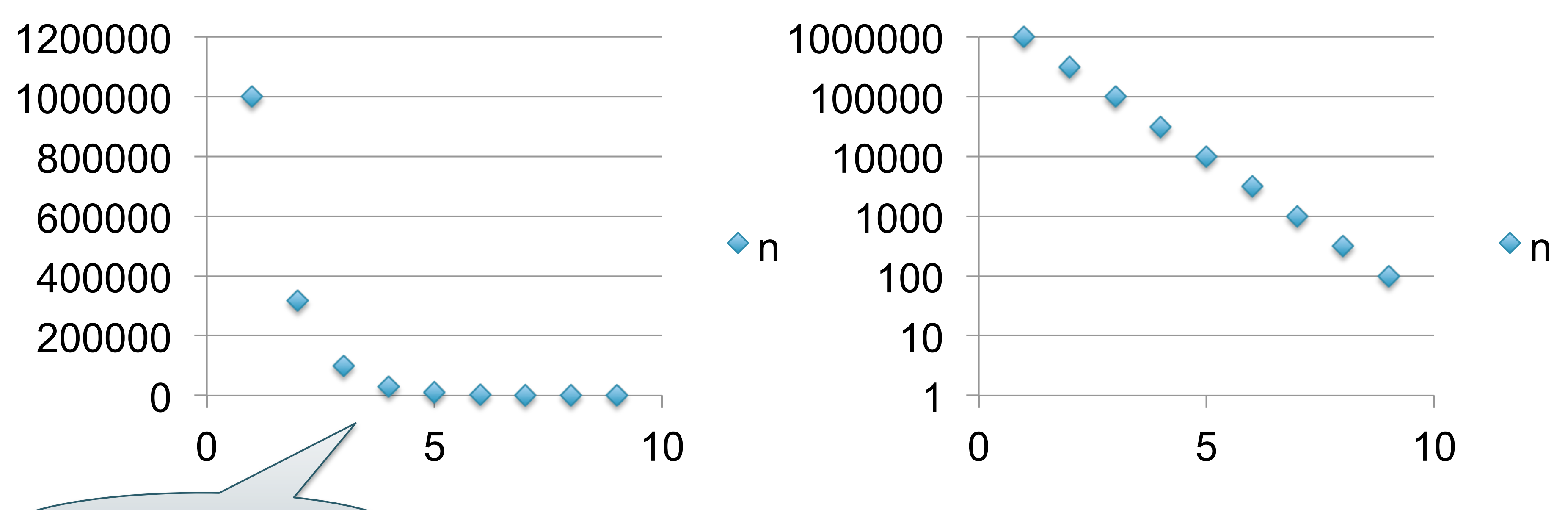
x 10000
120
100
80
60
40
20
0
0
5
10

What can you say about these graphs?

Exponential Distribution

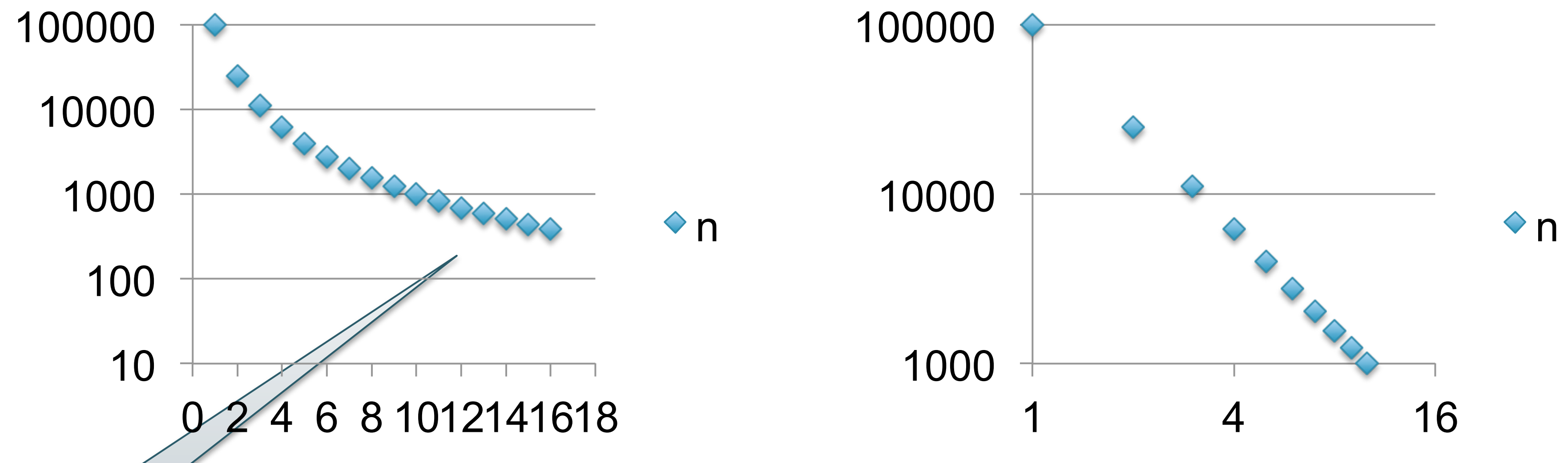
nodes with degree d

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



Power Law Distribution (Zipf)

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



The Histogram of the Web

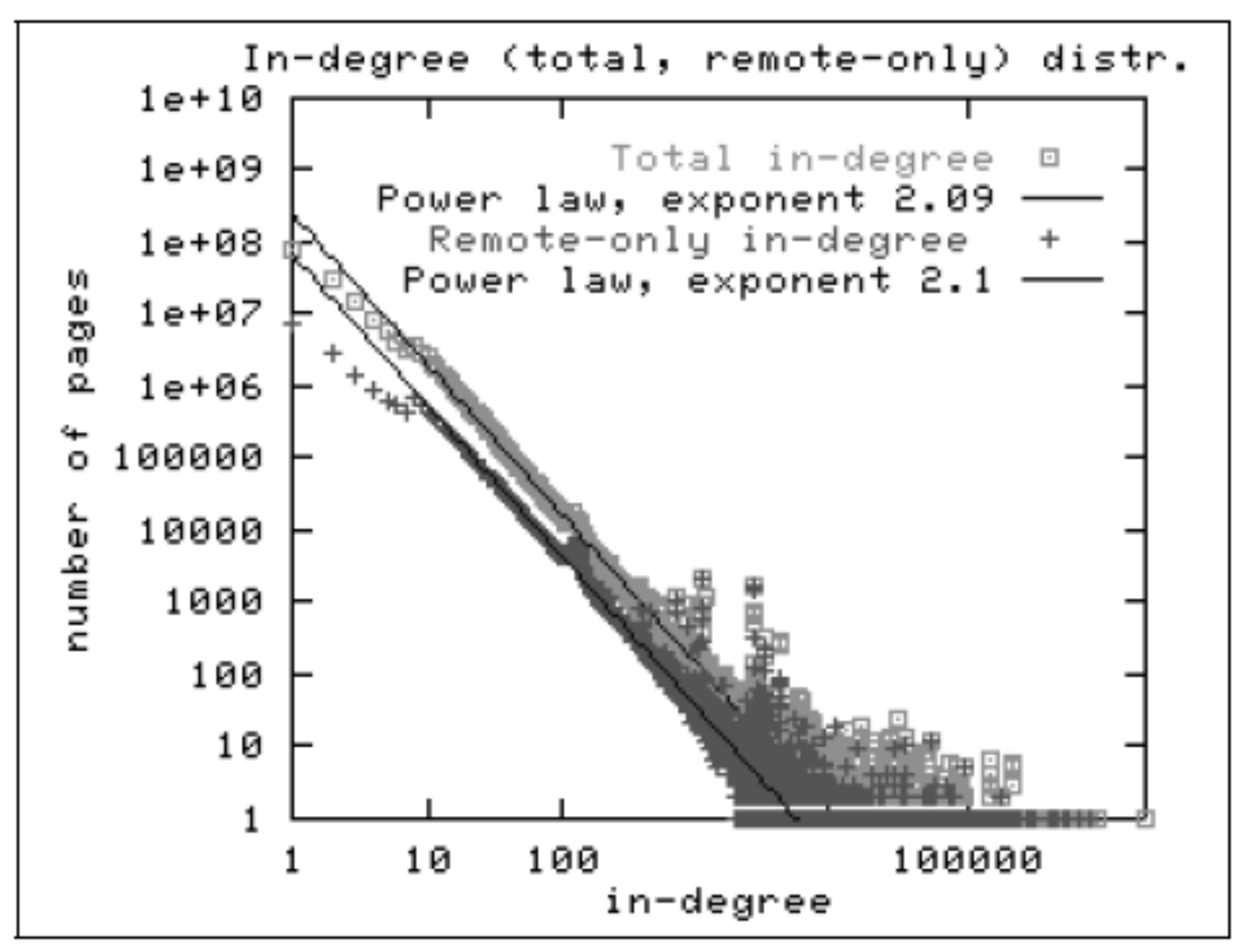


Figure 2: In-degree distribution.

Late 1990's
200M Webpages

Exponential ?

Power Law?

The Bowtie Structure of the Web

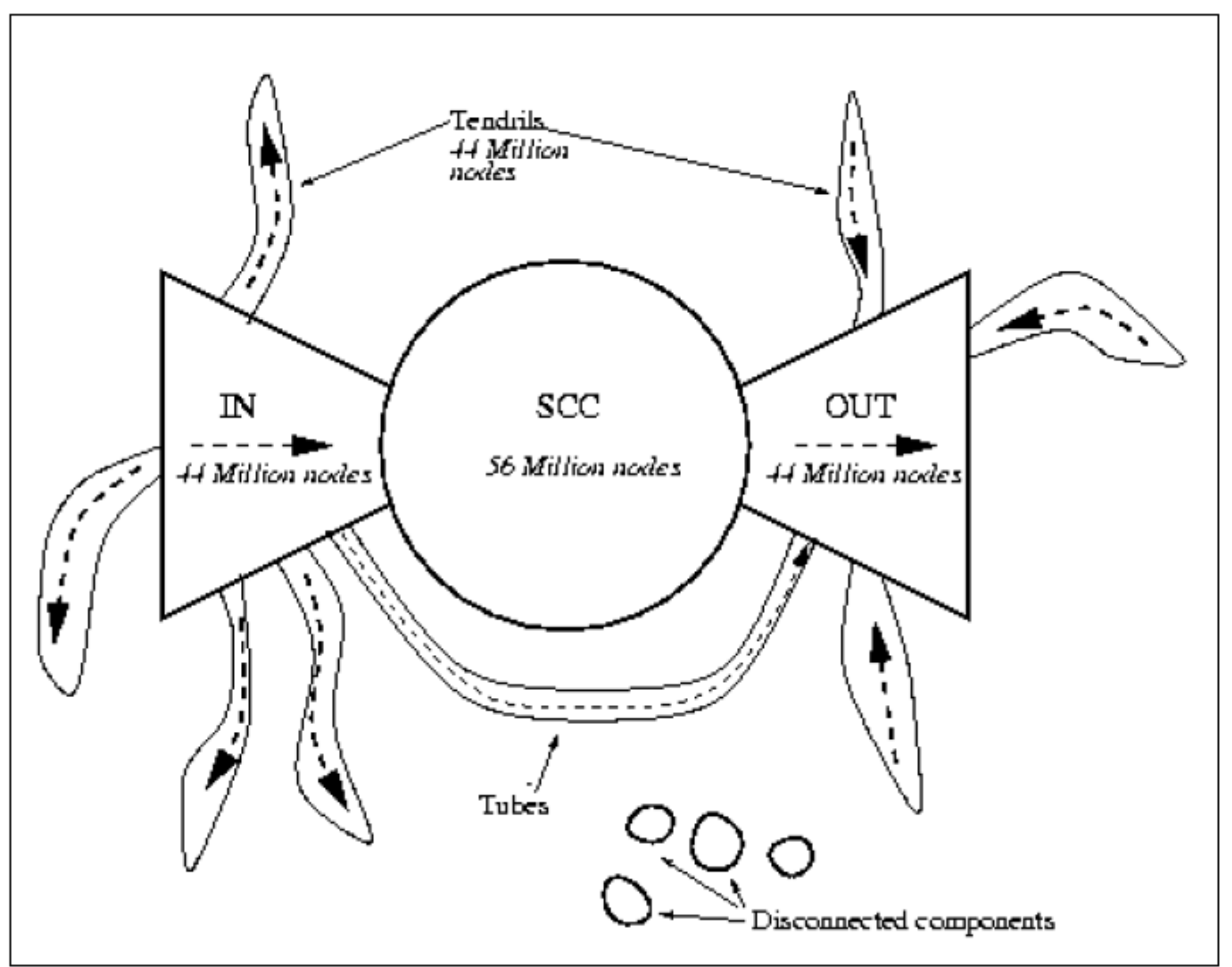


Figure 4: The web as a bowtie. SCC is a giant strongly connected component. IN consists of pages with paths to SCC, but no path from SCC. OUT consists of pages with paths from SCC, but no path to SCC. TENDRILS consists of pages that cannot surf to SCC, and which cannot be reached by surfing from SCC.

Executing a Large MapReduce Job

Anatomy of a Query Execution

- Running problem #4
- 20 nodes = 1 master + 19 workers
- Using PARALLEL 50

March 2013

Hadoop job_201303091944_0001 on domU-12-31-39-06-75-A1

User: hadoop
Job Name: PigLatin:DefaultJobName
Job File: hdfs://10.208.122.79:9000/mnt/var/lib/hadoop/tmp/mapred/staging/hadoop/.staging/job_201303091944_0001/job.xml
Submit Host: domU-12-31-39-06-75-A1.compute-1.internal
Submit Host Address: 10.208.122.79
Job-ACLs: All users are allowed
Job Setup: Successful
Status: Succeeded
Started at: Sat Mar 09 19:49:21 UTC 2013
Finished at: Sat Mar 09 23:33:14 UTC 2013
Finished in: 3hrs, 43mins, 52sec
Job Cleanup: Successful
Black-listed TaskTrackers: 1

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	7908	0	0	7908	0	14 / 16
reduce	100.00%	50	0	0	50	0	0 / 8

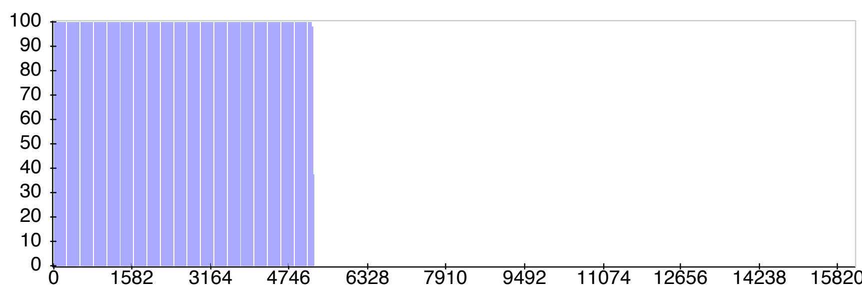
	Counter	Map	Reduce	Total
Job Counters	SLOTS_MILLIS_MAPS	0	0	454,162,761
	Launched reduce tasks	0	0	58
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Rack-local map tasks	0	0	7,938
	Total time spent by all maps waiting after reserving slots	0	0	0

Some other time (March 2012)

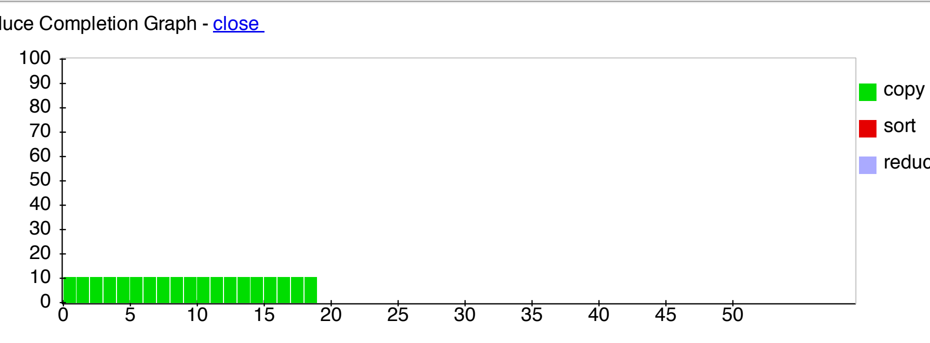
- Let’s see what happened…

1h 16min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0



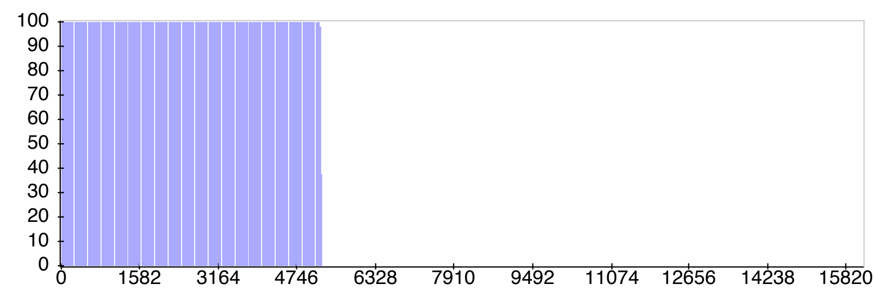
luce Completion Graph - close

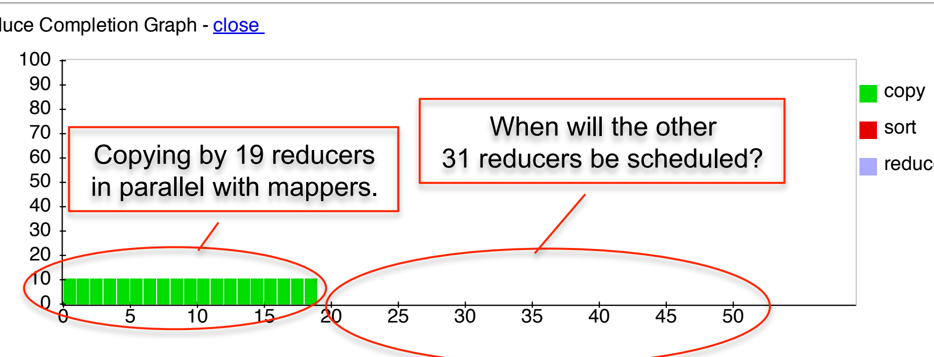


1h 16min

Only 19 reducers active, out of 50. Why?

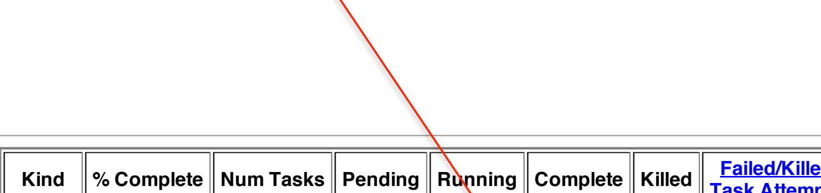
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0



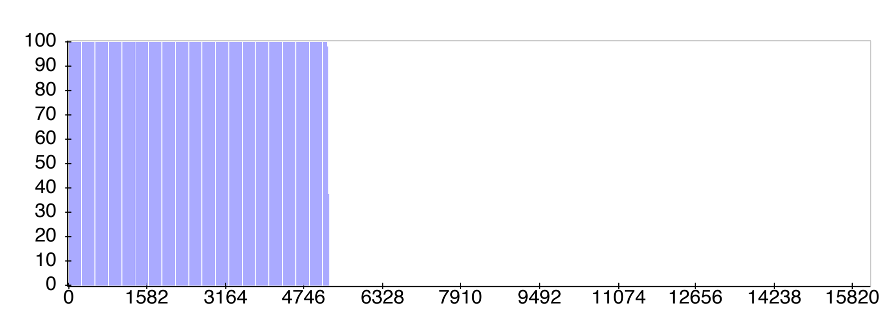


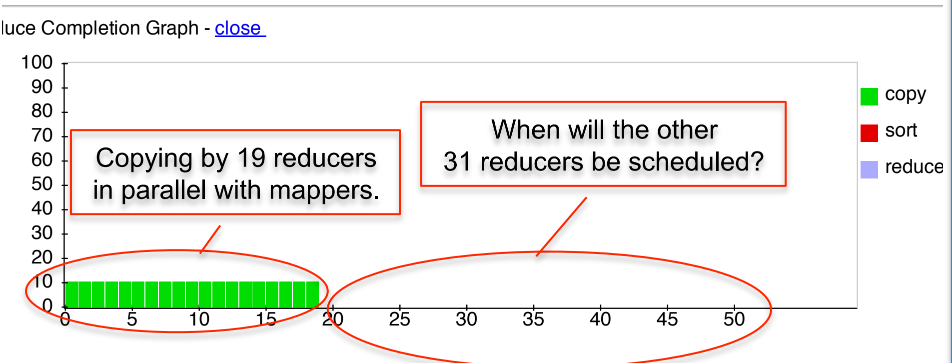
1h 16min

Only 19 reducers active, out of 50. Why?



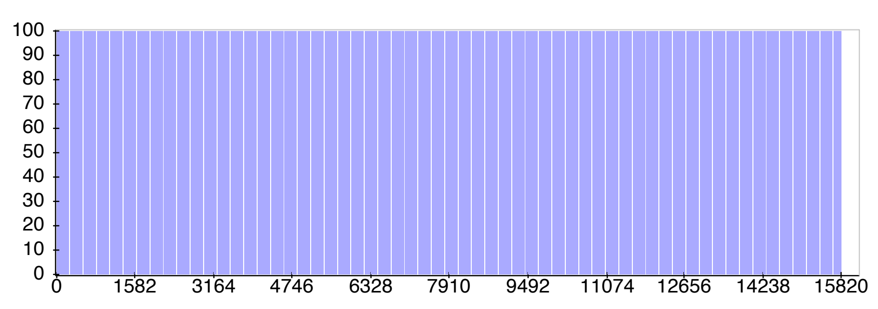
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0

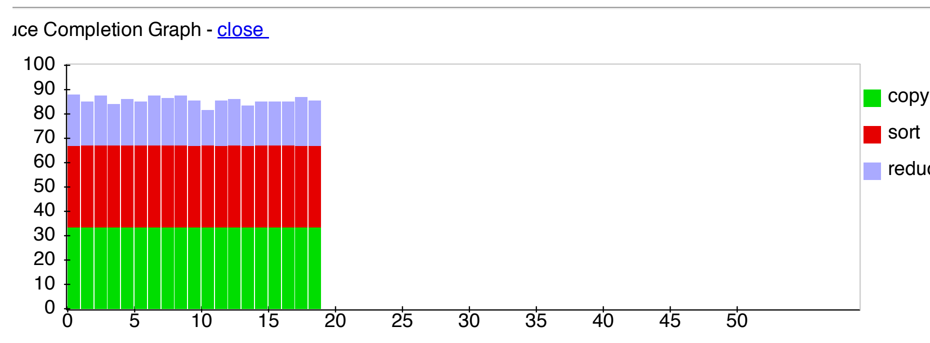




3h 50min

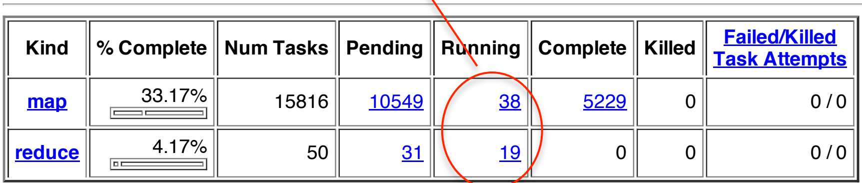
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	32.42%	50	31	19	0	0	0 / 0



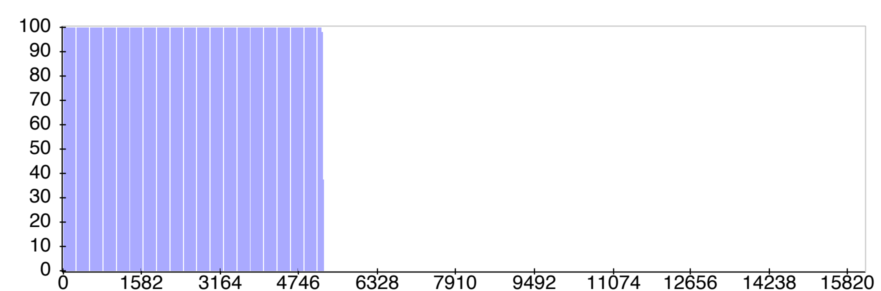


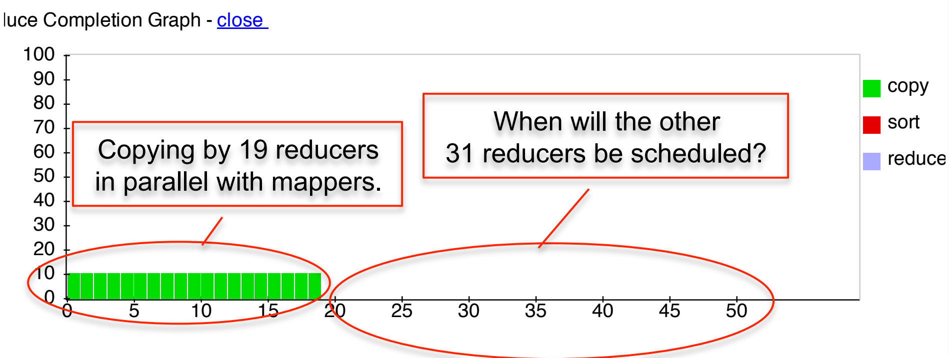
1h 16min

Only 19 reducers active, out of 50. Why?



Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0



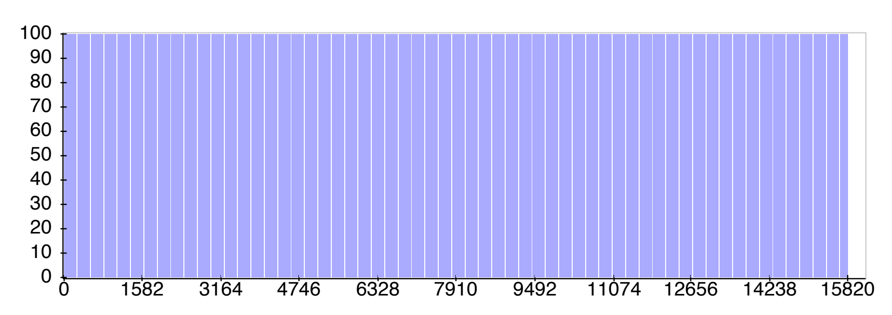


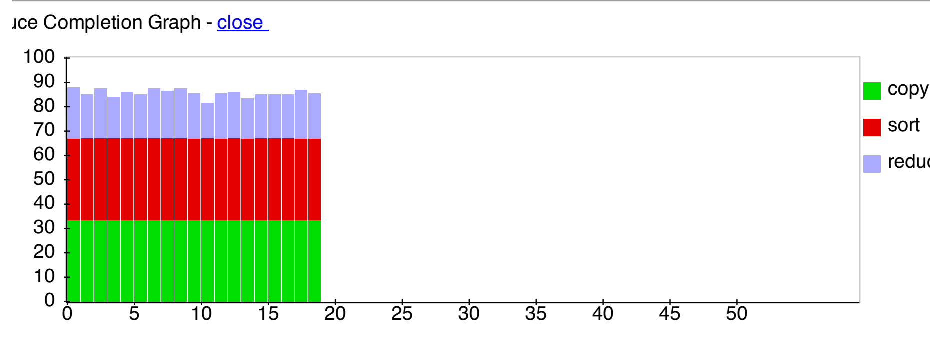
3h 50min

Speculative Execution

Completed. Sorting, and the rest of Reduce may proceed now

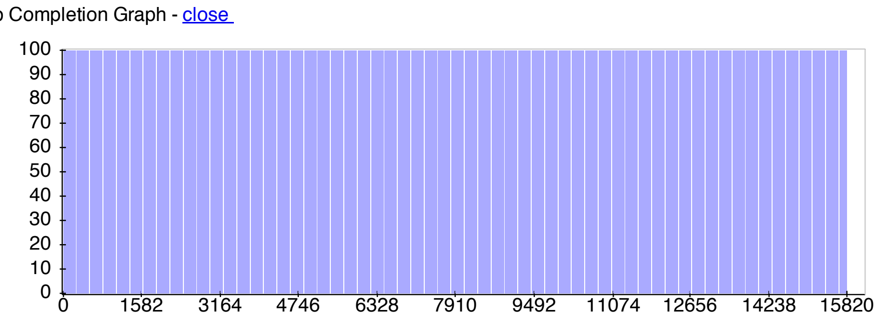
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	32.42%	50	31	19	0	0	0 / 0

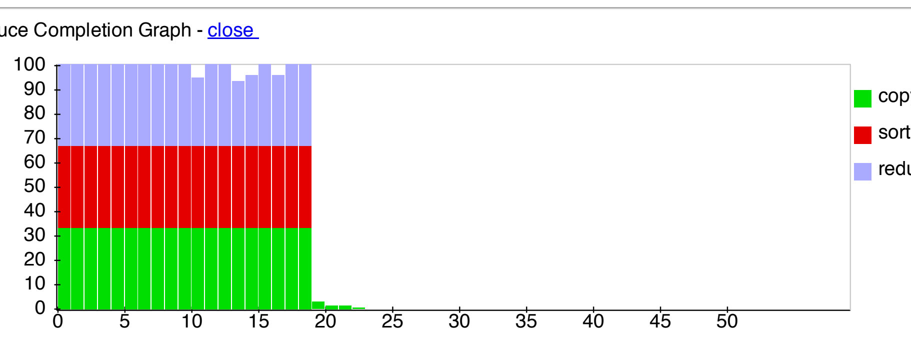




3h 51min

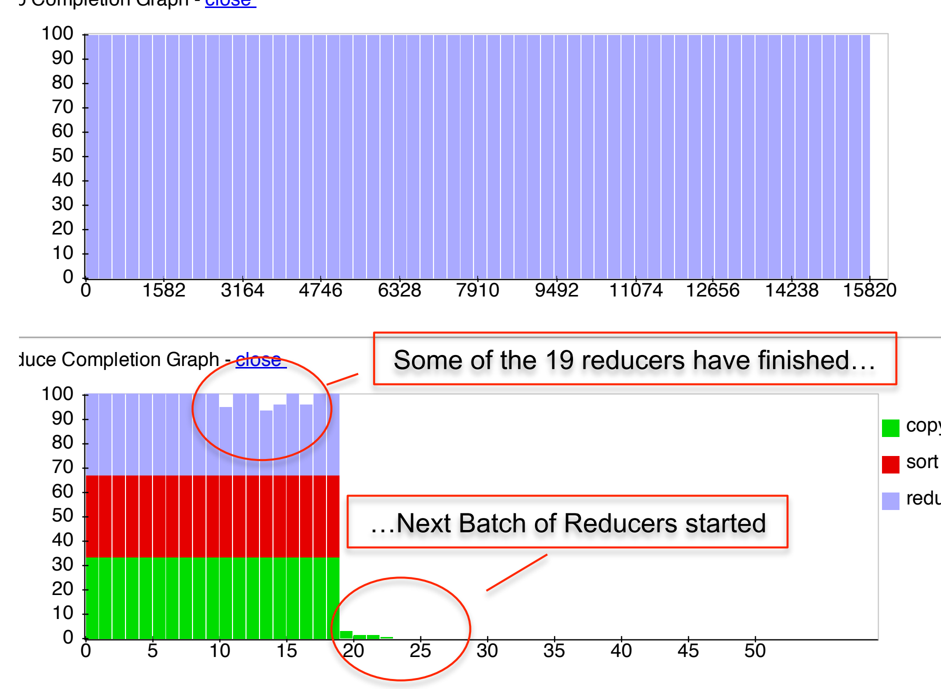
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	37.72%	50	19	22	9	0	0 / 0





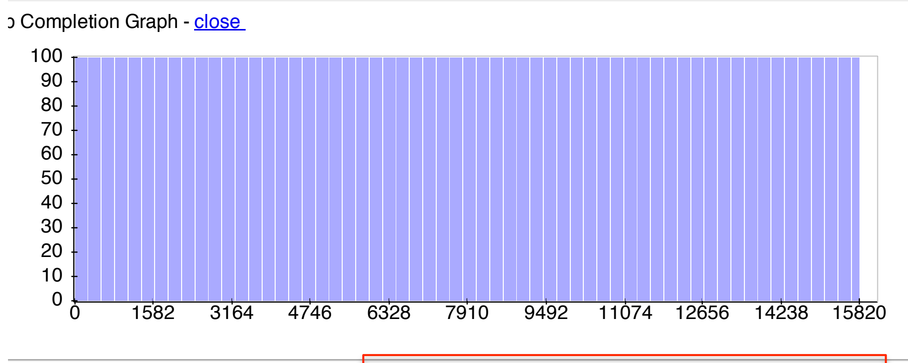
3h 51min

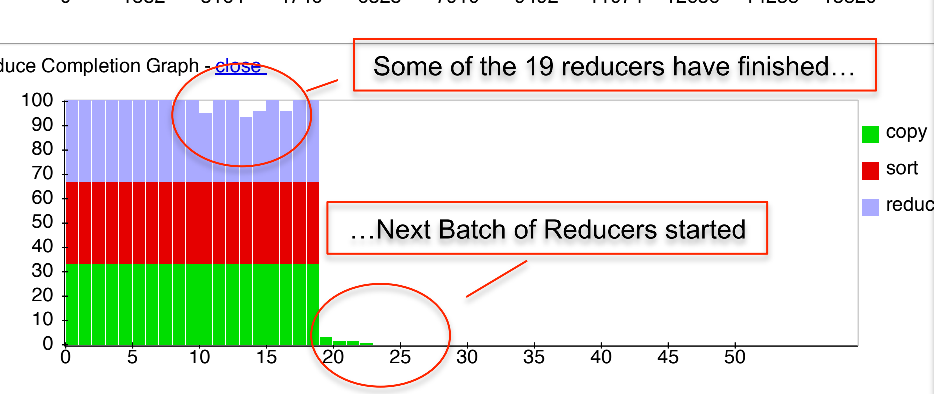
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	37.72%	50	19	22	9	0	0 / 0



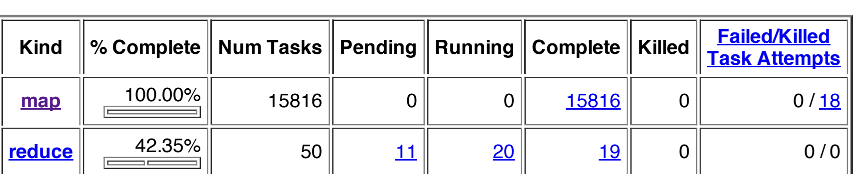
3h 51min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	37.72%	50	19	22	9	0	0 / 0

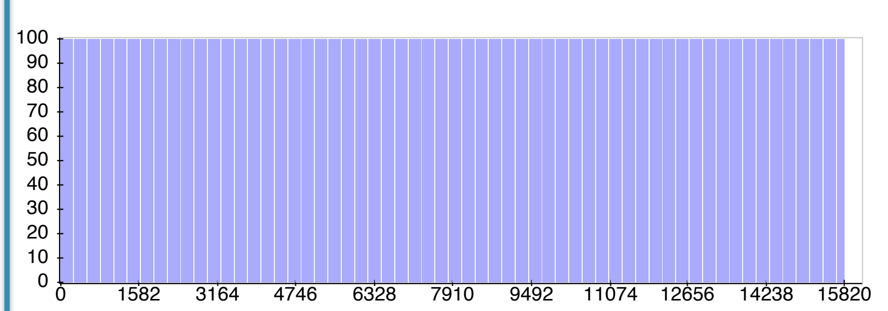


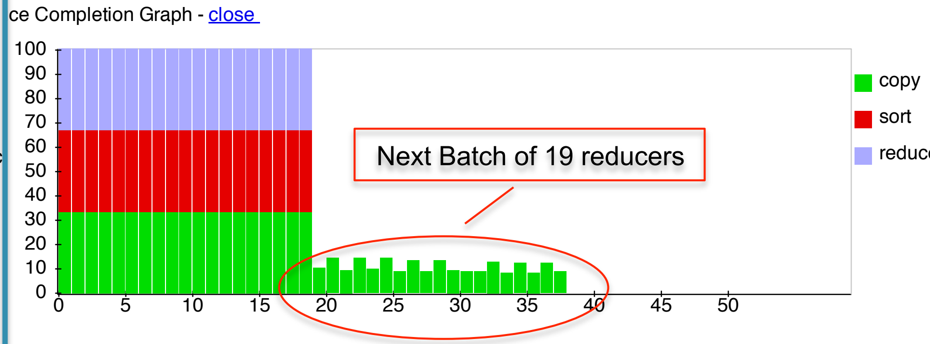


3h 52min



Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	42.35%	50	11	20	19	0	0 / 0

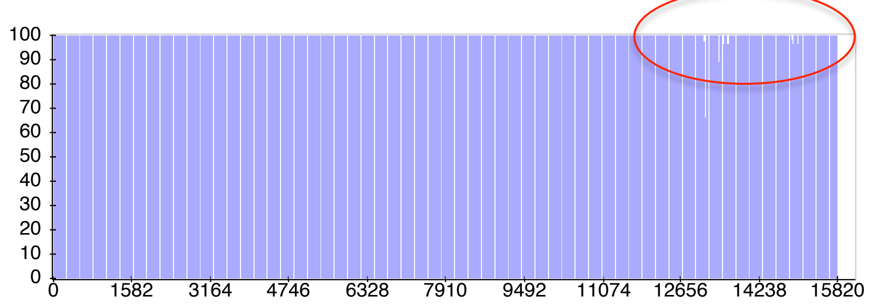


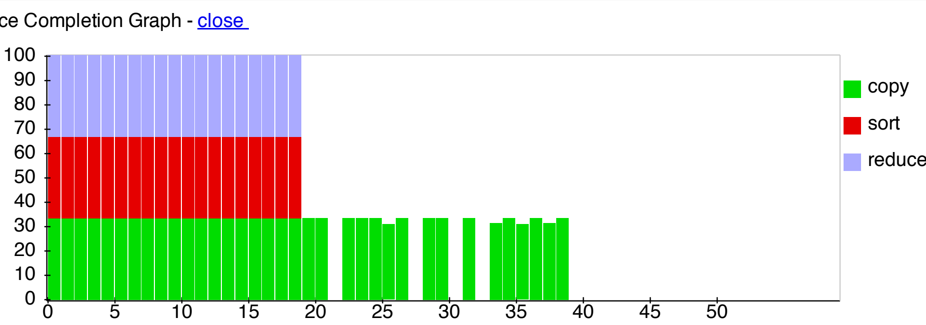


4h 18min

Several servers failed: “fetch error”. Their map tasks need to be rerun. All reducers are waiting….

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	99.88%	15816	2638	30	13148	0	15 / 3337
reduce	48.42%	50	15	16	19	0	0 / 0

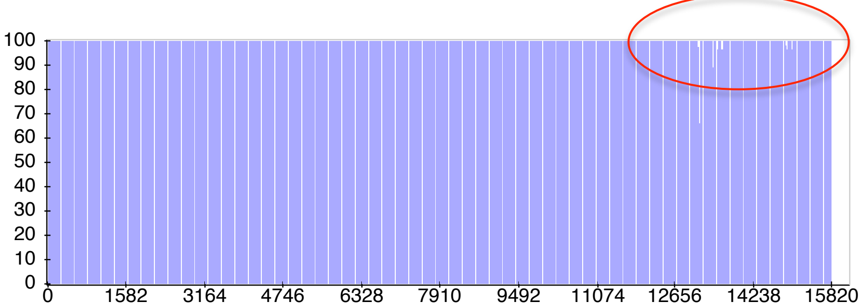




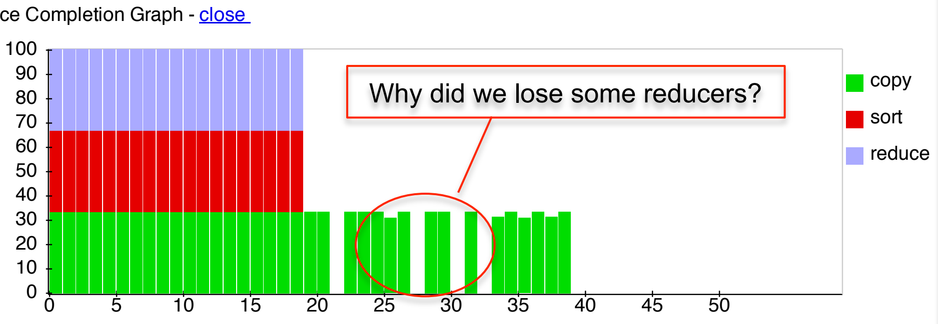
4h 18min

Several servers failed: “fetch error”. Their map tasks need to be rerun. All reducers are waiting….

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	99.88%	15816	2638	30	13148	0	15 / 3337
reduce	48.42%	50	15	16	19	0	0 / 0



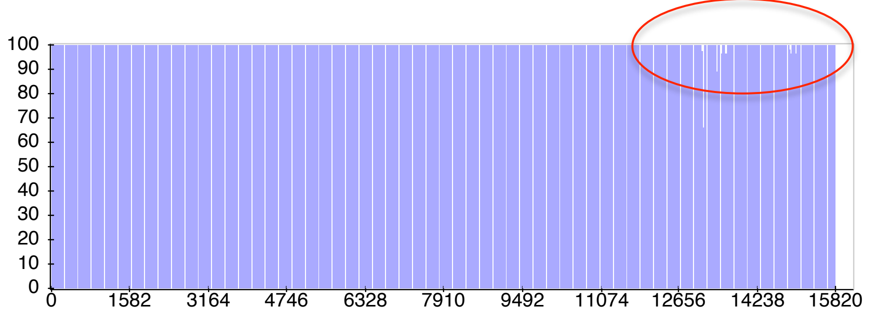
uce Completion Graph - close



4h 18min

Several servers failed: “fetch error”. Their map tasks need to be rerun. All reducers are waiting….

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	99.88%	15816	2638	30	13148	0	15 / 3337
reduce	48.42%	50	15	16	19	0	0 / 0

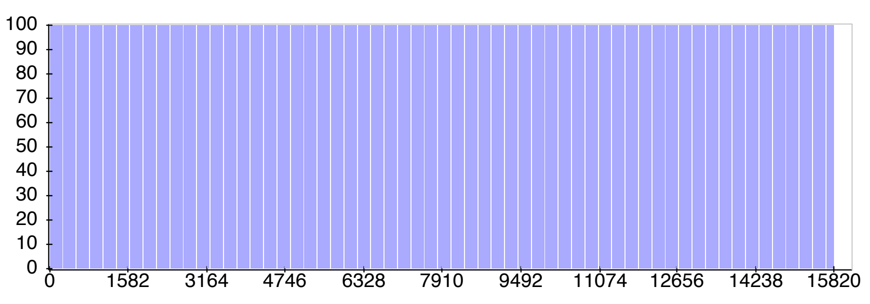


uce Completion Graph - close

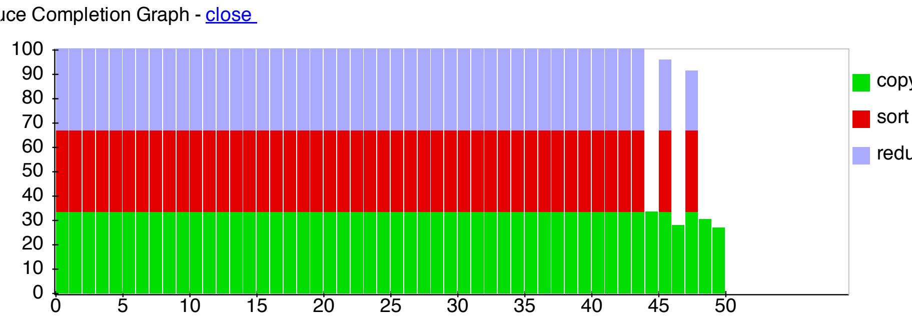
7h 10min

Mappers finished, reducers resumed.

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	26 / 5968
reduce	94.15%	50	0	6	44	0	0 / 8



uce Completion Graph - close



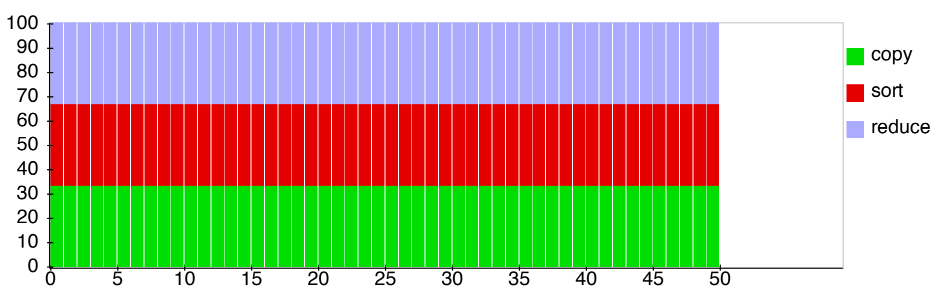
7h 20min

Success! 7hrs, 20mins.

Hadoop job_201203041905_0001 on ip-10-203-30-146

User: hadoop
Job Name: PigLatin:DefaultJobName
Job File: hdfs://10.203.30.146:9000/mnt/var/lib/hadoop/tmp/mapred/staging/hadoop/.staging/job_201203041905_0001/job.xml
Submit Host: ip-10-203-30-146.ec2.internal
Submit Host Address: 10.203.30.146
Job-ACLs: All users are allowed
Job Setup: Successful
Status: Succeeded
Started at: Sun Mar 04 19:08:29 UTC 2012
Finished at: Mon Mar 05 02:28:39 UTC 2012
Finished in: 7hrs, 20mins, 10sec
Job Cleanup: Successful
Black-listed TaskTrackers: 3

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	26 / 5968
reduce	100.00%	50	0	0	50	0	0 / 14



Parallel Query Processing

How do we compute these operations on a shared-nothing parallel db?

- Selection: $\sigma_{A=123}(R)$ (that's easy, won't discuss…)
- Group-by: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$

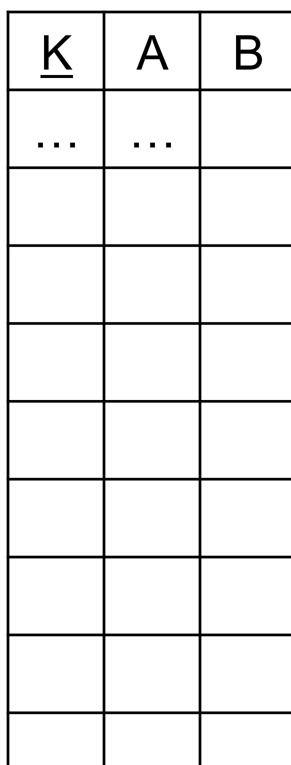
Before we answer that: how do we store R (and S) on a shared-nothing parallel db?

Review

- Shared memory / disk / nothing
- Speedup / Scaleup
- Interquery-, intraquery-, intraoperator parallelism
- Horizontal data partitioning

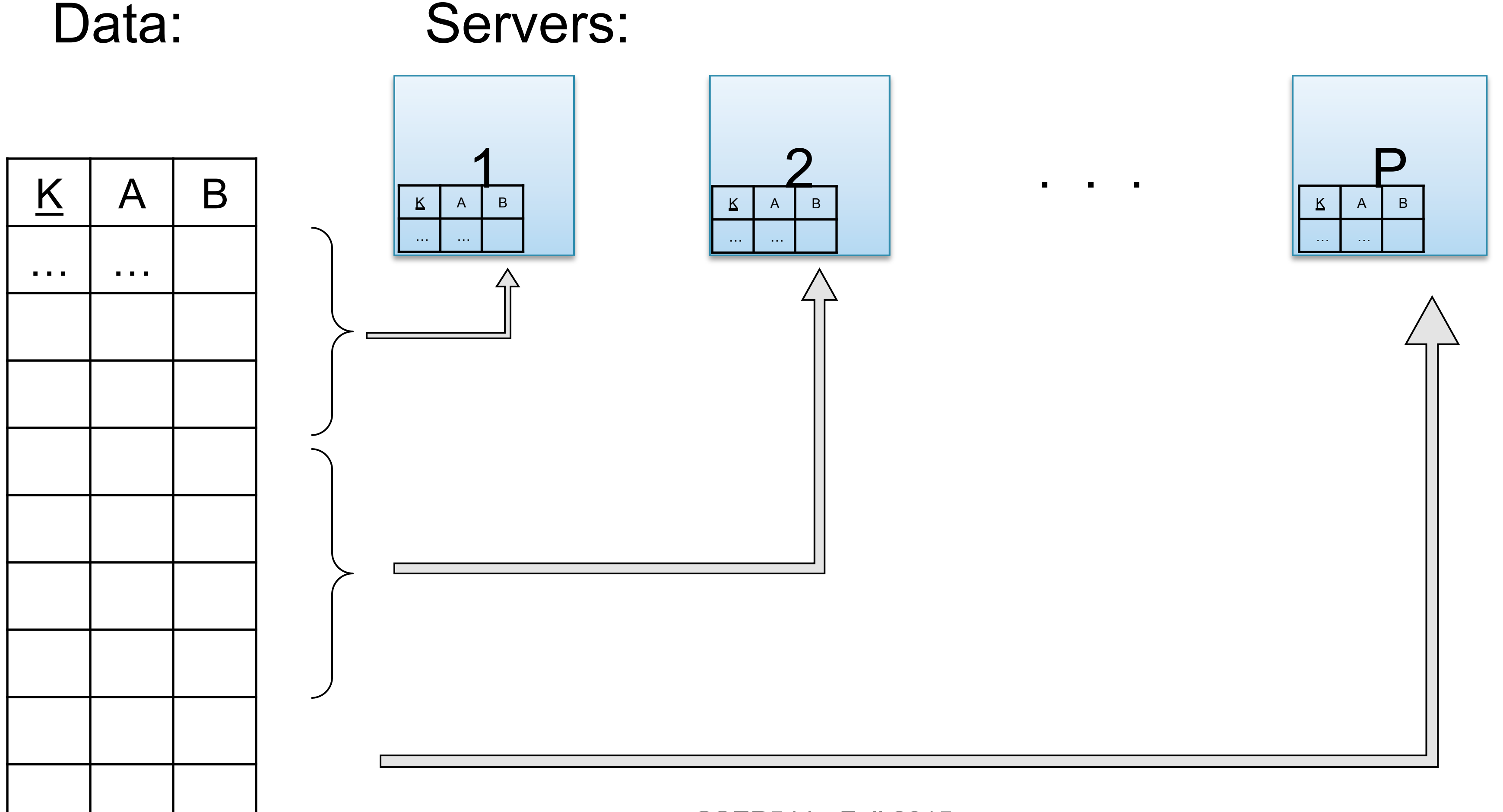
Horizontal Data Partitioning

Data:

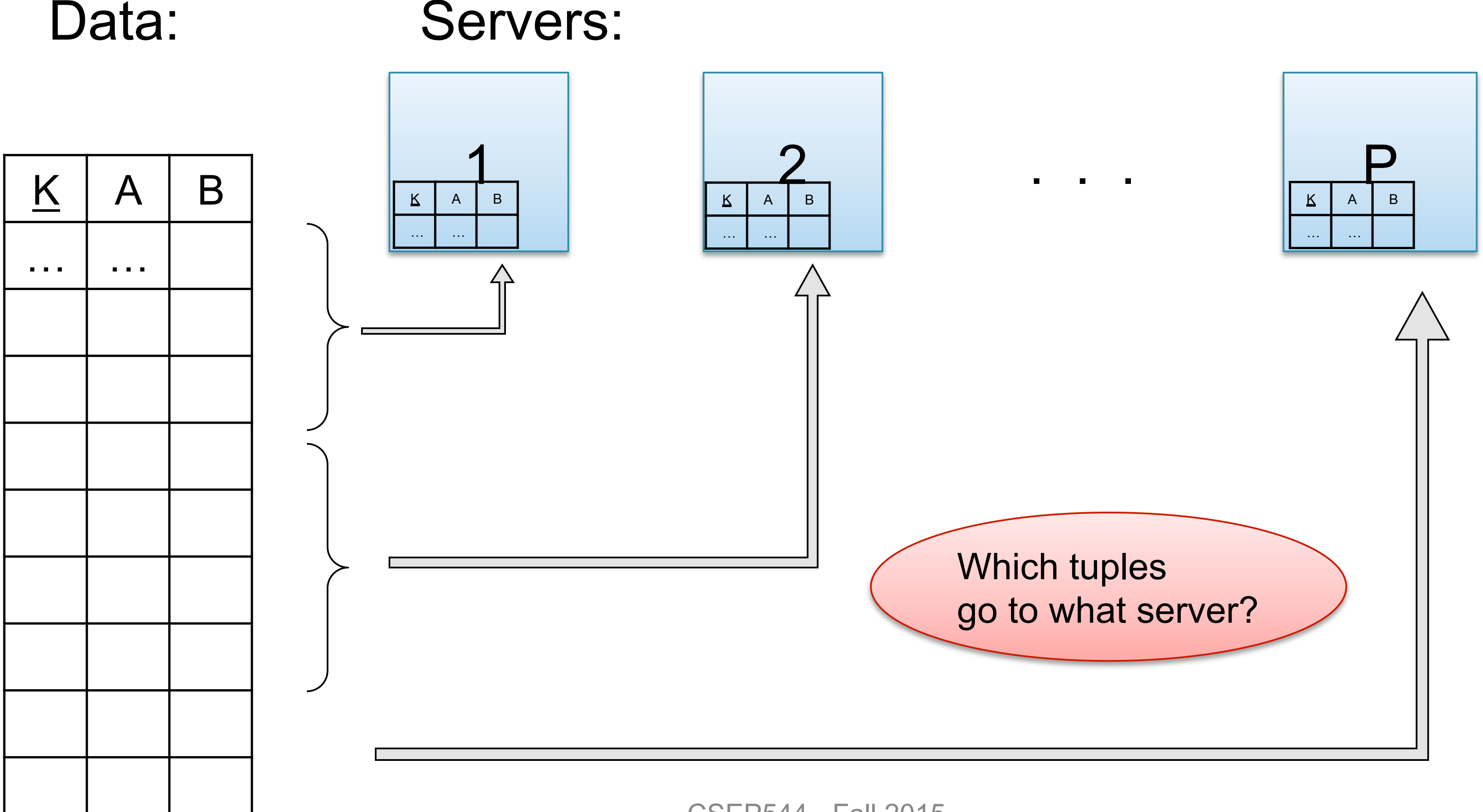


Servers:

Horizontal Data Partitioning



Horizontal Data Partitioning



Horizontal Data Partitioning

- Block Partition:
  - Partition tuples arbitrarily s.t. $size(R_1) \approx \ldots \approx size(R_P)$
- Hash partitioned on attribute A:
  - Tuple t goes to chunk i, where $i = h(t.A) \bmod P + 1$
- Range partitioned on attribute A:
  - Partition the range of A into $-\infty = v_0 < v_1 < \ldots < v_P = \infty$
  - Tuple t goes to chunk i, if $v_{i-1} < t.A < v_i$

Parallel Hash-Partitioned GroupBy

Data: R(<u>K</u>,A,B,C)

Query: $\gamma_{A,sum(C)}(R)$

Discuss in class how to compute in each case:

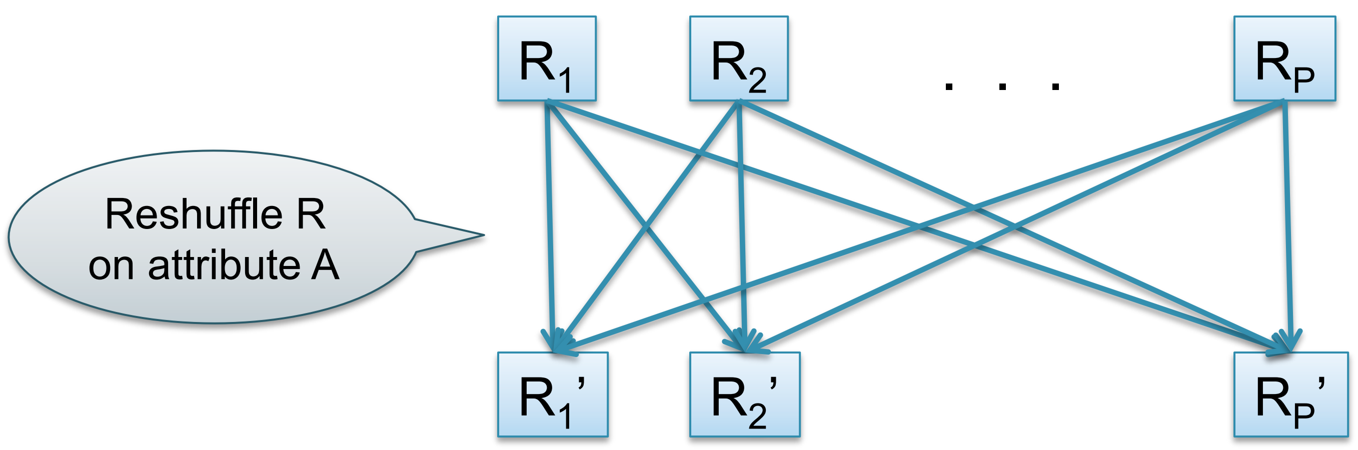
- R is hash-partitioned on A
- R is block-partitioned
- R is hash-partitioned on K

Parallel Hash-Partitioned GroupBy

Data: R(K,A,B,C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K



Parallel Hash-Partitioned Join

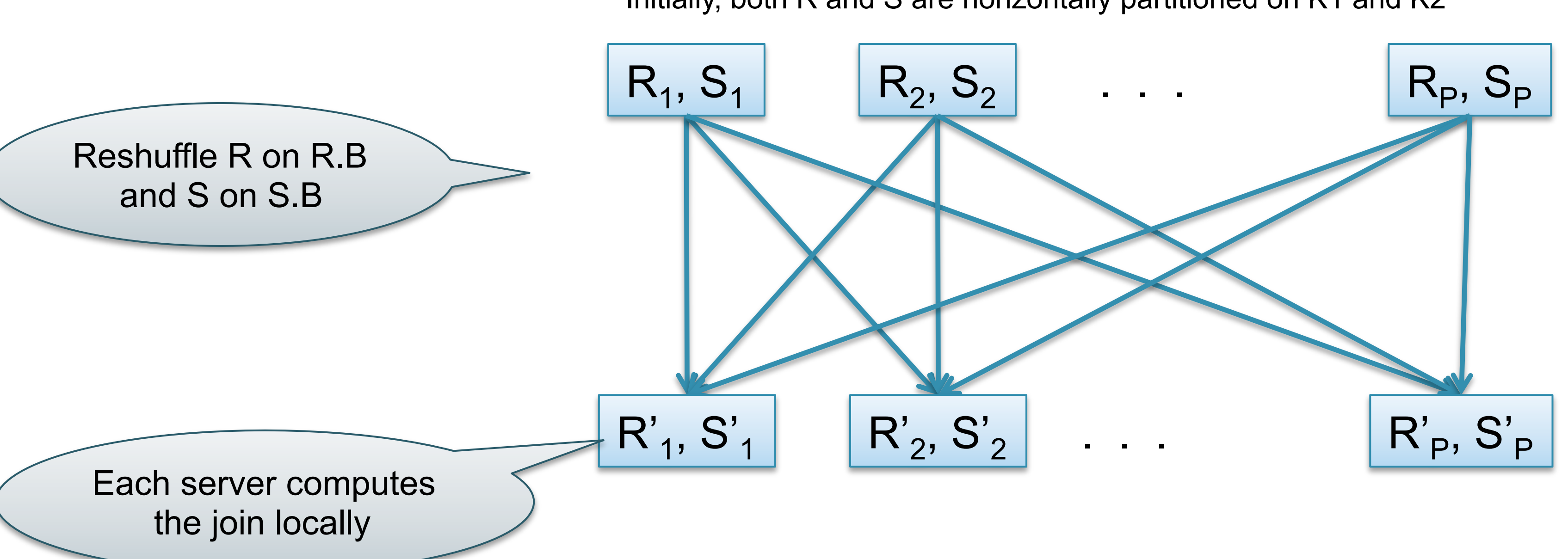
- Data: R(K1,A, B), S(K2, B, C)
- Query: R(K1,A,B) $\bowtie$ S(K2,B,C)

Initially, both R and S are horizontally partitioned on K1 and K2

R_1, S_1 R_2, S_2 R_P, S_P

Parallel Hash-Partitioned Join

- Data: R(<u>K1</u>,A, B), S(<u>K2</u>, B, C)
- Query: R(<u>K1</u>,A,B) ⋈ S(<u>K2</u>,B,C)



Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P, what is the new running time?

- If we double both P and the size of R, what is the new running time?

Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P, what is the new running time?
  - Half (each server holds ½ as many chunks)
- If we double both P and the size of R, what is the new running time?
  - Same (each server holds the same # of chunks)

Uniform Data v.s. Skewed Data

- Let R(K,A,B,C); which of the following partition methods may result in skewed partitions?

- Block partition

- Hash-partition
  - On the key K
  - On the attribute A

Uniform Data v.s. Skewed Data

- Let R(K,A,B,C); which of the following partition methods may result in skewed partitions?

- Block partition

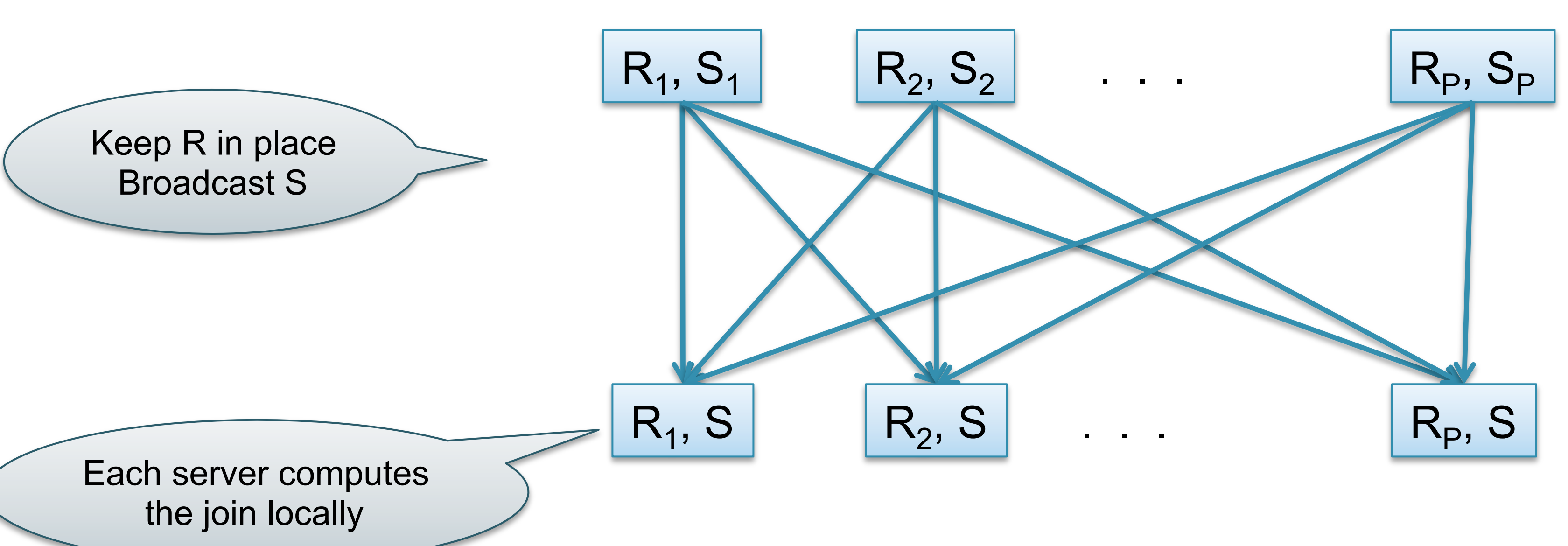
Assuming good hash function

- Hash-partition
  - On the key K
  - On the attribute A

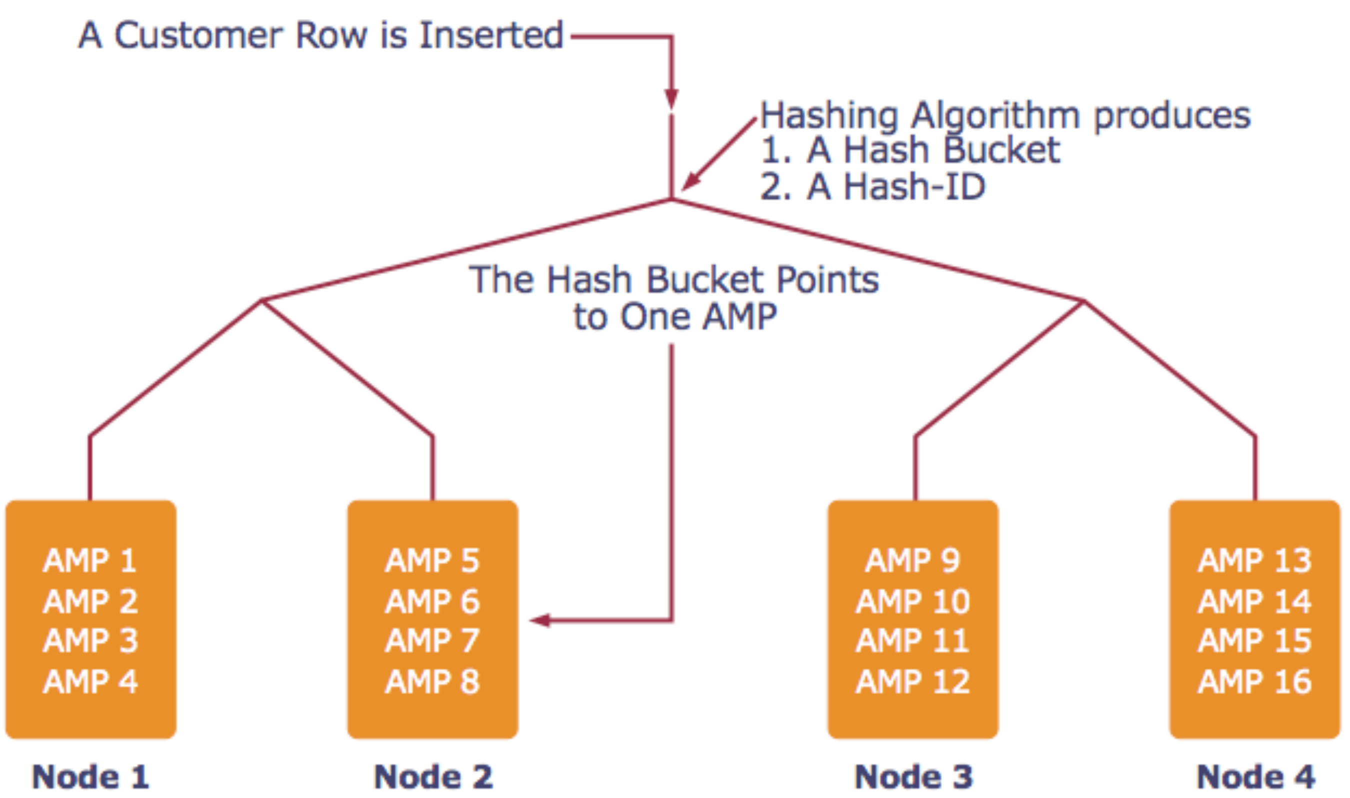
E.g. when all records have the same value of the attribute A, then all records end up in the same partition

Broadcast Join

- Data: R(K1,A, B), S(K2, B, C)
- Query: R(K1,A,B) ⋈ S(K2,B,C)



Example: Teradata – Loading

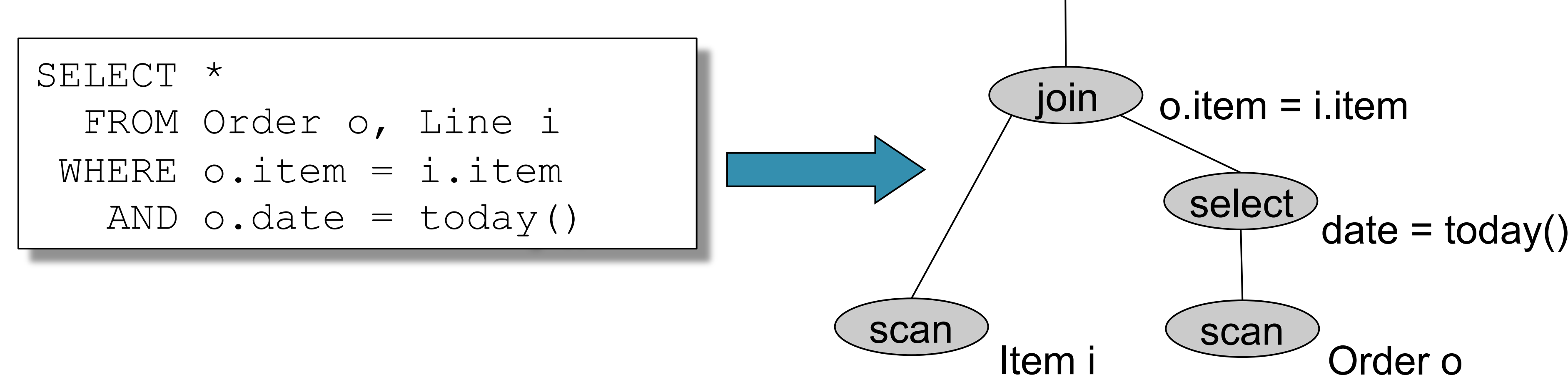


AMP = "Access Module Processor" = unit of parallelism

Order(oid, item, date), Line(item, …)

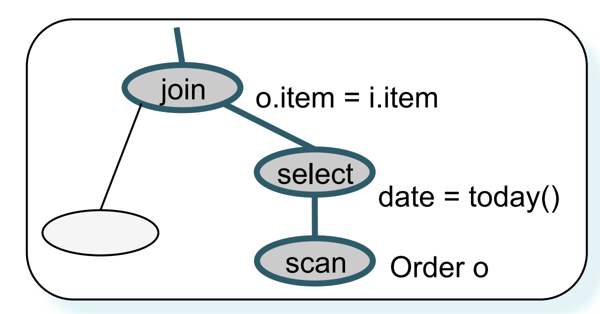
Example: Teradata – Query Execution

Find all orders from today, along with the items ordered



Order(oid, item, date), Line(item, …)

Query Execution



AMP 1 AMP 2 AMP 3

hash h(o.item)
select date=today()
scan Order o

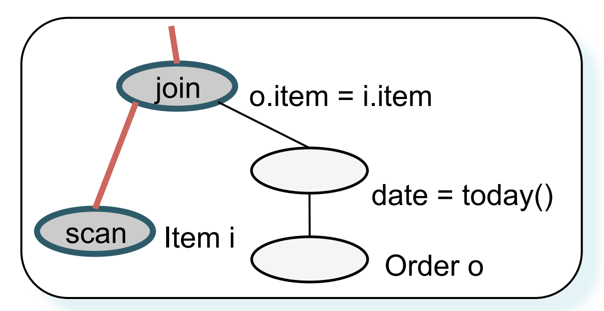
hash h(o.item)
select date=today()
scan Order o

hash h(o.item)
select date=today()
scan Order o

AMP 1 AMP 2 AMP 3

Order(oid, item, date), Line(item, …)

Query Execution



AMP 1

AMP 2

AMP 3

hash

h(i.item)

scan

Item i

hash

h(i.item)

scan

Item i

hash

h(i.item)

scan

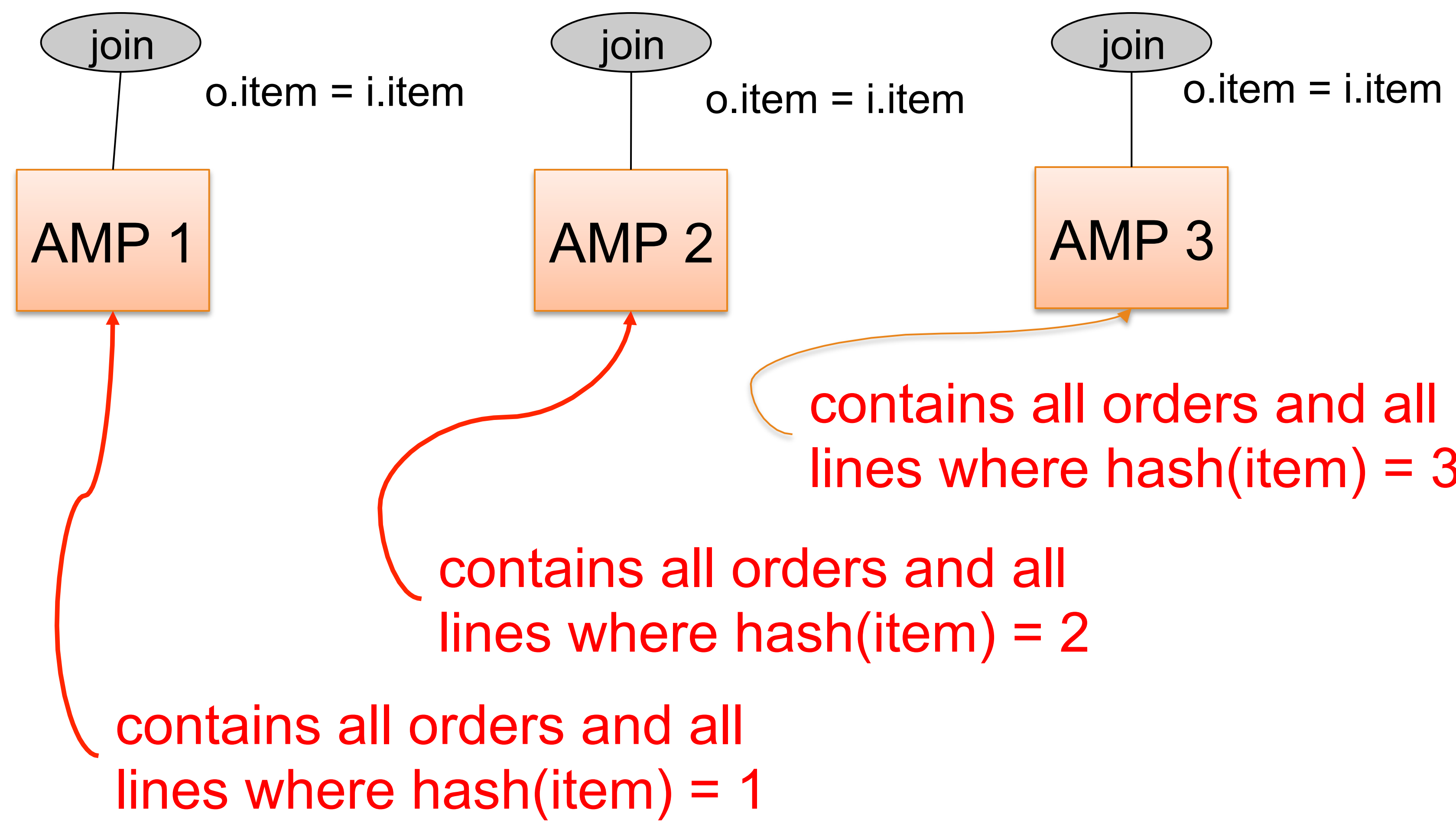
Item i

AMP 1

AMP 2

AMP 3

Query Execution



MapReduce

Cluster Computing

- Commodity servers, high speed network
- Servers → Racks → Data centers
- 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
- File is partitioned into *chunks*, e.g. 64MB
- Each chunk is replicated, e.g. 3 times
- 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 MapReduce 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

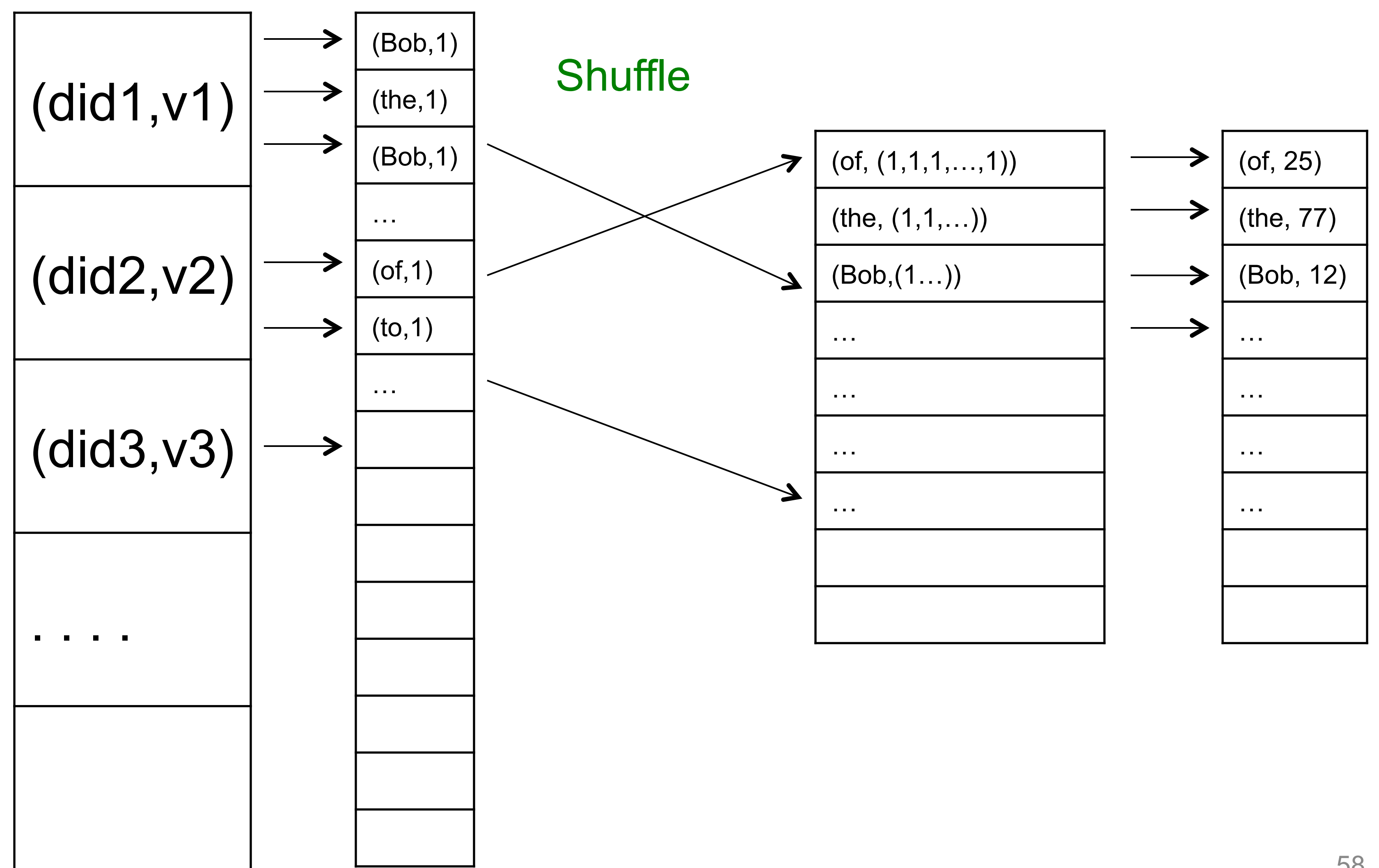
- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

```
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



Jobs v.s. Tasks

What is the difference?

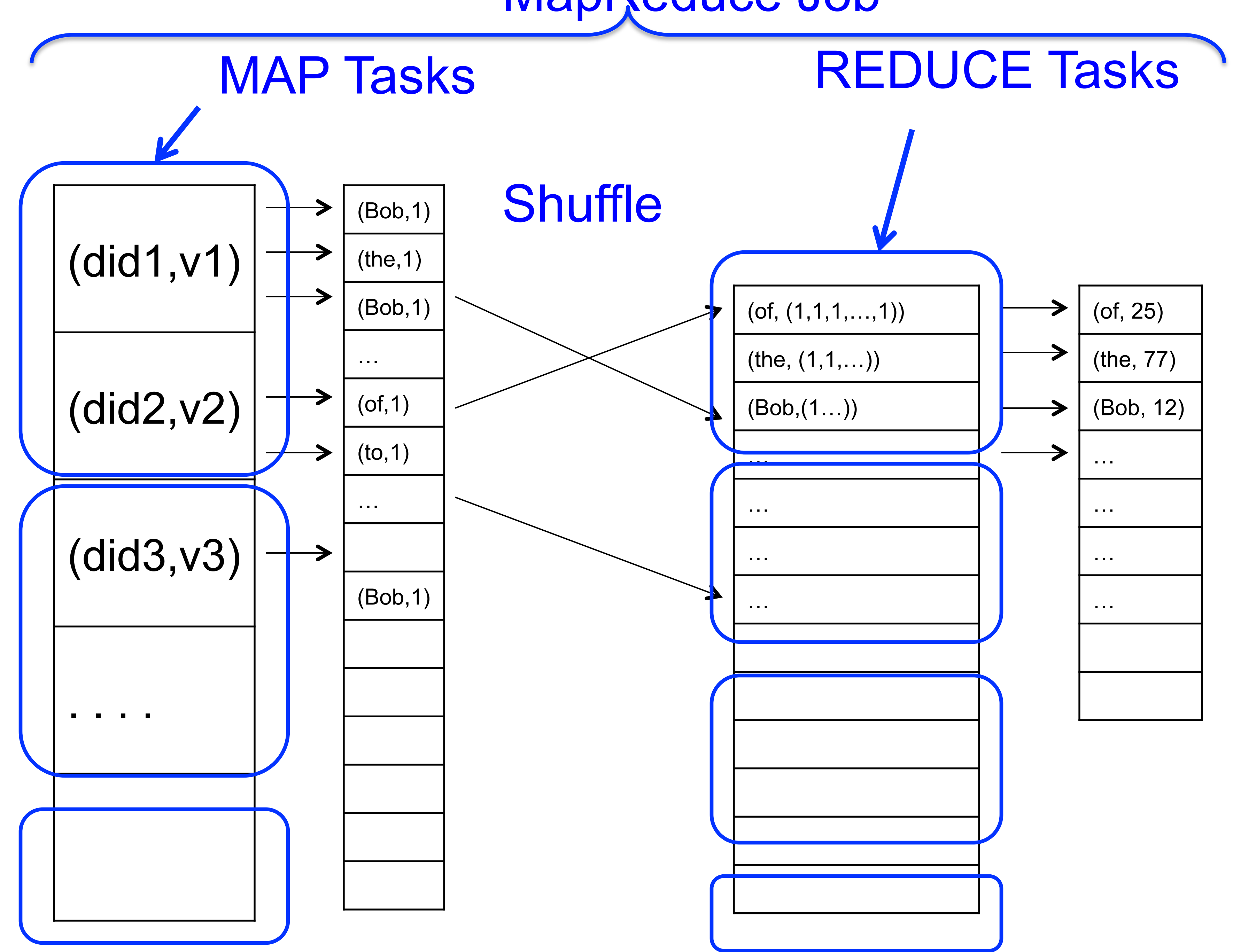
Jobs v.s. Tasks

What is the difference?

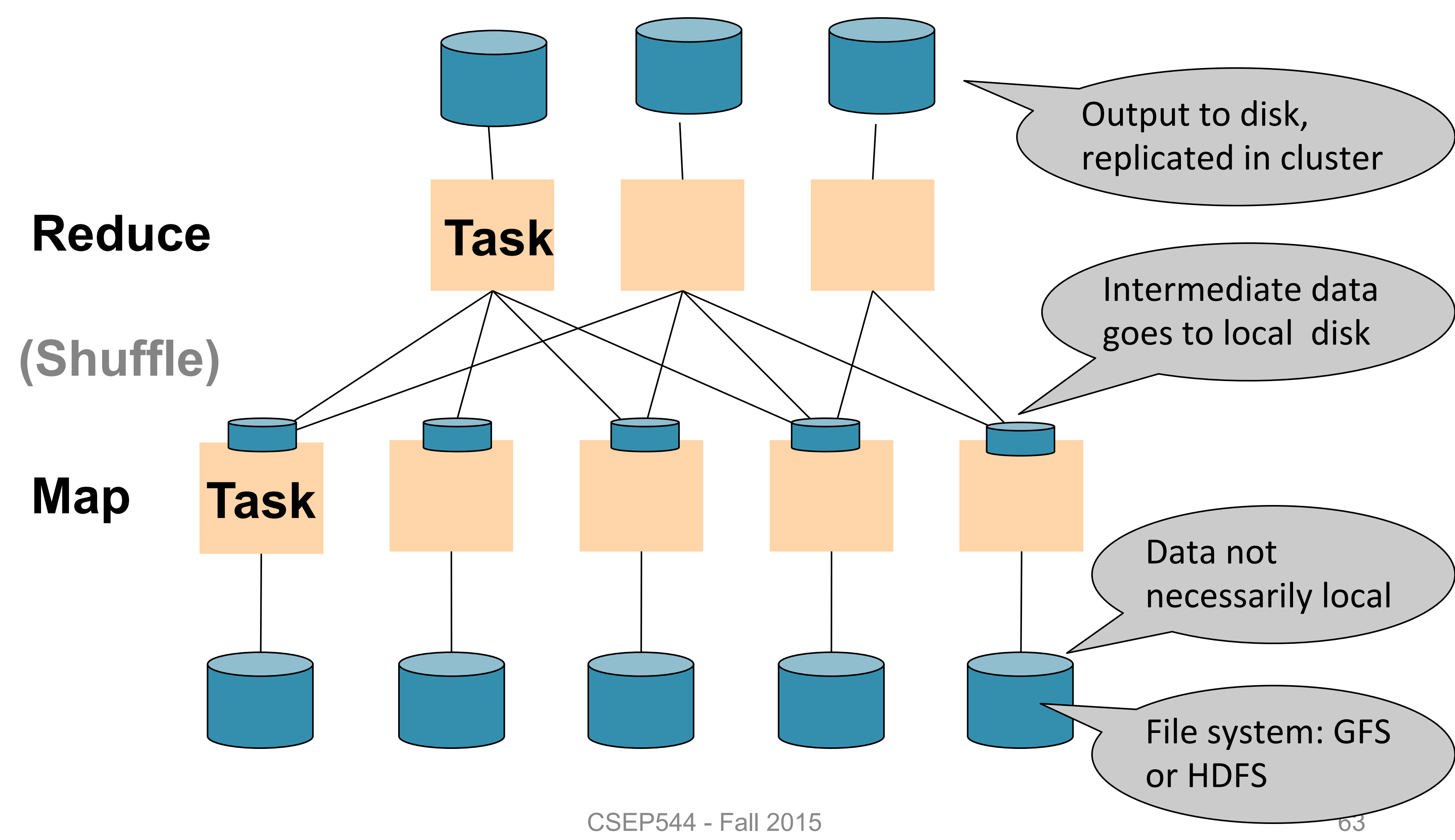
- A MapReduce Job
  - One single “query”, e.g. count the words in all docs
  - More complex queries may consists of multiple jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

Map Tasks, Reduce Tasks

- What are they?
- How is their number determined?
- What are the pros/cons in having small/ large number of tasks?


MapReduce Job
MAP Tasks
REDUCE Tasks
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(Bob,1)
(the,1)
(Bob,1)
…
(of,1)
(to,1)
…
(Bob,1)
(of, (1,1,1,…,1))
(the, (1,1,…))
(Bob,(1…))
…
…
…
…
(of, 25)
(the, 77)
(Bob, 12)
…
…
…
…

MapReduce Execution Details

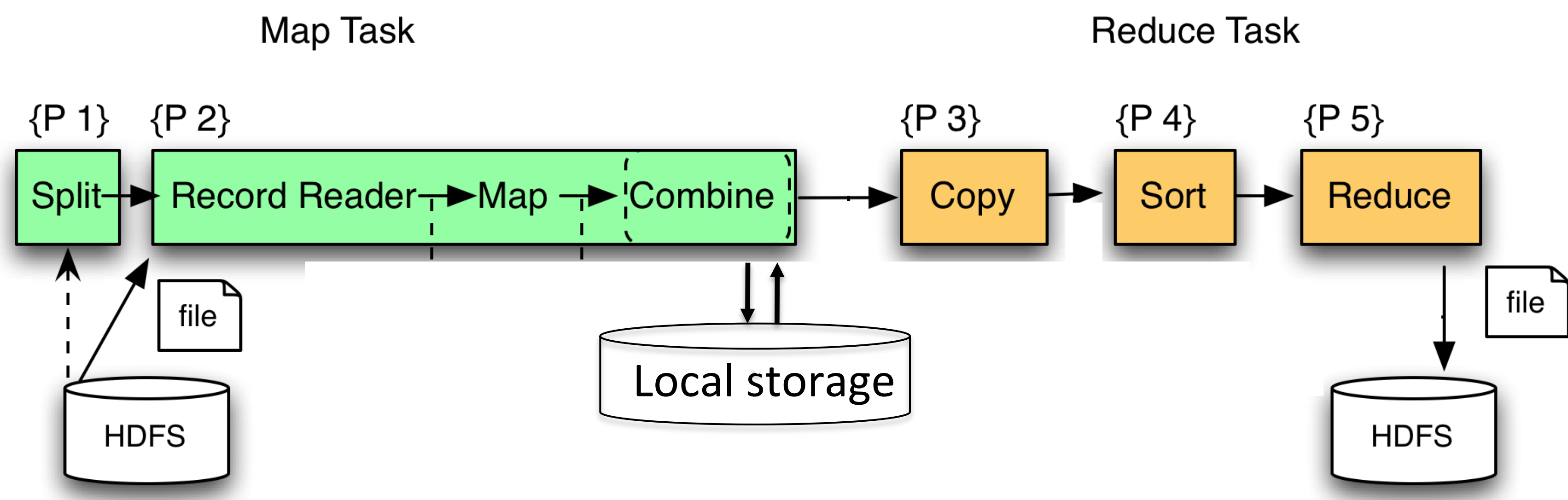


Workers

- A worker is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node

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*

MapReduce Summary

- Hides scheduling and parallelization details
- However, very limited queries
  – Difficult to write more complex queries
  – Need multiple MapReduce jobs
- Solution: declarative query language
  – PigLatin, Dremel (SQL), HiveQL (SQL)

User(name, age) ⋈ Page(user, url)

Hash Join in MR

```
map([String key], String value):
    // value.relation is either 'User' or 'Page'
```

```
reduce(String user, Iterator values):
    User = empty;  Page = empty;
```

Relying entirely on the MR system to do the hashing

User(name, age) $\bowtie_{name=user}$ Page(user, url)

Hash Join in MR

```
map([String key], String value):
    // value.relation is either 'User' or 'Page'
    if value.relation='User':
          EmitIntermediate(value.name, (1, value));
    else // value.relation='Page':
          EmitIntermediate(value.user, (2, value));
```

Relying entirely on the MR system to do the hashing

```
reduce(String user, Iterator values):
    User = empty;  Page = empty;
    for each v in values:
          if v.type = 1: User.insert(v)
      else Page.insert(v);
    foreach v1 in User, v2 in Page
          Emit(v1,v2);
```

User(name, age) $\bowtie_{name=user}$ Page(user, url)

Hash Join in MR

```
map([String key], String value):
    // value.relation is either 'User' or 'Page'
    if value.relation='User':
        EmitIntermediate(h(value.name), (1, value));
    else // value.relation='Page':
        EmitIntermediate(h(value.user), (2, value));
```

Controlling the hash function

```
reduce(String user, Iterator values):
    User = empty;  Page = empty;
    foreach v in values:
        if v.type = 1: User.insert(v)
     else Page.insert(v);
    foreach v1 in User, v2 in Page
       if v1.name=v2.user: Emit(v1,v2);
```

$User(\underline{name}, age) \bowtie_{name=user} Page(user, url)$

Broadcast Join in MR

Assume Page is huge, User is smaller
Broadcast join does not shuffle Page; instead broadcasts User
Sketch the Map and Reduce functions (in class):

Transactions

Outline

- Transaction basics
- Recovery
  - Start today, continue next week
- Concurrency control

Reading Material for Lectures 6/7

Textbook (Ramakrishnan): Ch. 16, 17, 18

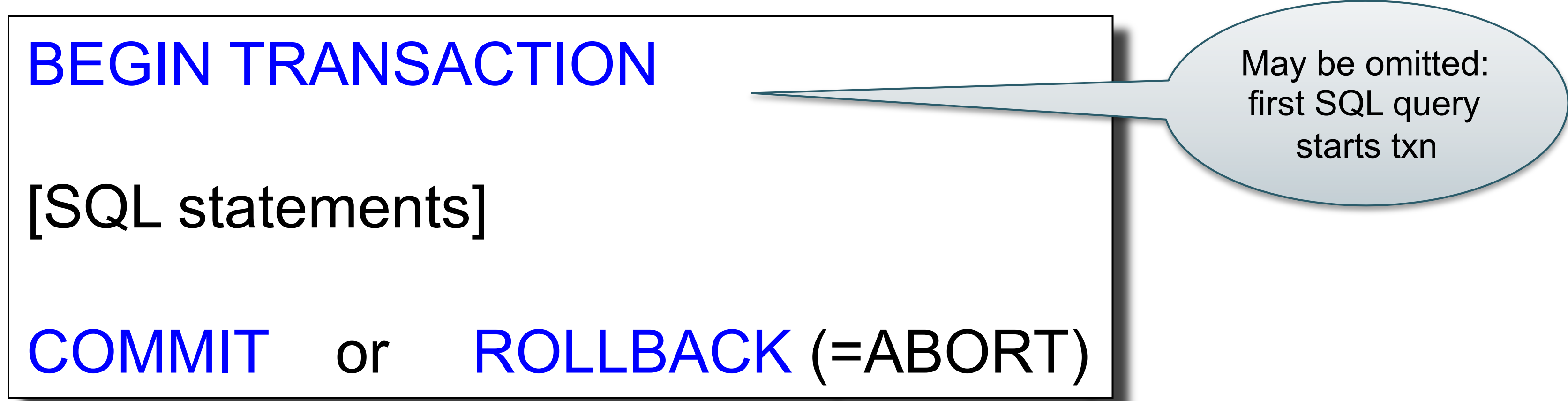
Second textbook (Garcia-Molina)

- Ch. 17.2, 17.3, 17.4
- Ch. 18.1, 18.2, 18.3, 18.8, 18.9

Optional: M. Franklin, *Concurrency Control and Recovery*

Transaction

Definition: a transaction is a sequence of updates to the database with the property that either all complete, or none completes (all-or-nothing).



In ad-hoc SQL: each statement = one transaction

Implementing Transactions

- System crash
  - Software failure (e.g. division by 0
  - Hardware failure (e.g power failure)

- Interferences with other users
  - “Anomalies” – 3 have famous names

System Crash

```
Client 1:
BEGIN TRANSACTION
UPDATE Account1
SET balance= balance – 500


UPDATE Account2
SET balance = balance + 500
COMMIT
```

Crash !

1st Famous Anomaly: Lost Update

Client 1:

```
BEGIN TRANSACTION
  UPDATE Account1
  SET balance= balance+$100
COMMIT
```

Client 2:

```
BEGIN TRANSACTION
  UPDATE Account1
  SET balance=balance-$100
COMMIT
```

Lost update: two TXN's update the same element, but only one succeeds.

2nd Famous Anomaly: Inconsistent Read

```
Client 1: transfer $100
BEGIN TRANSACTION
  UPDATE Account1
  SET balance= balance+$100

  UPDATE Account2
  SET balance= balance+$100
COMMIT
```

```
Client 2: check total balance
BEGIN TRANSACTION
  SELECT sum(balance)
  FROM All_Accounts
COMMIT
```

Inconsistent read: TXN sees some updates by another TXN, but not all updates.

3rd Famous Anomaly: Dirty Reads

```
-- Client 1:
BEGIN TRANSACTION
  UPDATE Account1
  SET balance= balance+$100


  . . .
  ROLLBACK
```

```
-- Client 2: get cash $100
BEGIN TRANSACTION
X = Account1.balance
If (X>=100)
    { …dispense money…
      COMMIT }
. . .
```

Dirty read: TXN reads a value written by another transaction that later aborts.

ACID Properties

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

Outline

- Recovery from failures (the A in ACID)
  - Today and next week
- Concurrency Control (the C in ACID)
  - Next week

Log-based Recovery

Basics (based on Garcia-Molina Ch. 17.2, 17.3, 17.4)

- Undo logging
- Redo logging

Aries: (Ramakrishnan Ch. 18)

Transaction Abstraction

- Database is composed of ***elements***.
- 1 element can be either:
  - 1 page = physical logging
  - 1 record = logical logging
- Aries uses both (will discuss later)

Primitive Operations of Transactions

- READ(X,t)
  - copy element X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to element X

- INPUT(X)
  - read element X to memory buffer
- OUTPUT(X)
  - write element X to disk

Running Example

```
BEGIN TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t)
COMMIT;
```

Initially, A=B=8.

Atomicity requires that either
(1) T commits and A=B=16, or
(2) T does not commit and A=B=8.

```
READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)
```

	Transaction	Buffer pool		Disk	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

Yes it's bad: A=16, B=8….

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

Yes it's bad: A=B=16, but not committed

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Is this bad ?

No: that's OK

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Crash !

Typically, OUTPUT is after COMMIT (why?)

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Typically, OUTPUT is after COMMIT (why?)

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Atomic Transactions

- **FORCE or NO-FORCE**
  - Should all updates of a transaction be forced to disk before the transaction commits?
- **STEAL or NO-STEAL**
  - Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?

Force/No-steal

- **FORCE**: Pages of committed transactions must be forced to disk before commit

- **NO-STEAL**: Pages of uncommitted transactions cannot be written to disk

Easy to implement (how?) and ensures atomicity

No-Force/Steal

- **NO-FORCE**: Pages of committed transactions need not be written to disk
- **STEAL**: Pages of uncommitted transactions may be written to disk

In either case, Atomicity is violated; need WAL

Write-Ahead Log

The Log: append-only file containing log records

- Records every single action of every TXN
- Force log entry to disk
- After a system crash, use log to recover

Three types: UNDO, REDO, UNDO-REDO

UNDO Log

FORCE and STEAL

Undo Logging

Log records

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its *old* value was v

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

WHAT DO WE DO ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Crash !

WHAT DO WE DO ?

We UNDO by setting B=8 and A=8

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

What do we do now ?

Crash !

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

What do we do now ?

Nothing: log contains COMMIT

Recovery with Undo Log

…
…
<T6,X6,v6>
…
…
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

Crash !

Question1: Which updates are undone ?

Question 2:
How far back do we need to read in the log ?

Question 3:
What happens if there is a second crash, during recovery ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)					8	
READ(A,t)	8				8	
t:=t*2	16	8			8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

When must we force pages to disk ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

FORCE

RULES: log entry *before* OUTPUT *before* COMMIT

Undo-Logging Rules

U1: If T modifies X, then <T,X,v> must be written to disk before OUTPUT(X)

U2: If T commits, then OUTPUT(X) must be written to disk before <COMMIT T>

FORCE

- Hence: OUTPUTs are done *early*, before the transaction commits

REDO Log

NO-FORCE and NO-STEAL

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Is this bad ?

Yes, it's bad: A=16, B=8

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Is this bad ?

Yes, it's bad: lost update

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Is this bad ?

No: that’s OK.

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Redo Logging

One minor change to the undo log:

- <T,X,v>= T has updated element X, and its *new* value is v

Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**	**REDO Log**
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash !

How do we recover ?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash !

How do we recover ?

We REDO by setting A=16 and B=16

Recovery with Redo Log

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>

Crash !

Show actions during recovery

Action	t	Mem A	Me[illegible]	[illegible]k A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8	[illegible]	[illegible]	8	
t:=t*2	16	8	[illegible]	[illegible]	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

When must
we force pages
to disk ?

?

?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

NO-STEAL

RULE: OUTPUT *after* COMMIT

Redo-Logging Rules

R1: If T modifies X, then both <T,X,v> and <COMMIT T> must be written to disk before OUTPUT(X)

NO-STEAL

- Hence: OUTPUTs are done *late*

Comparison Undo/Redo

- Undo logging: OUTPUT must be done early:
  - Inefficient

- Redo logging: OUTPUT must be done late:
  - Inflexible