ANNOUNCEMENTS

P5: File Systems - Only xv6;

- Test scripts available
- Due Monday, 12/14 at 9:00 pm
- Fill out form if would like a new project partner

Exam 4: In-class Tuesday 12/15

- Not cumulative!
- Only covers Advanced Topics starting today
- Worth ½ of other midterms
- No final exam in final exam period (none on 12/23)

Advanced Topics:

- Distributed Systems, Dist File Systems (NFS, AFS, MapReduce, GFS)

Course Feedback – Today and Tomorrow ONLY

Read as we go along: Technical Paper on MapReduce

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

ADVANCED TOPICS: MAP-REDUCE

Questions answered in this lecture:

Review: When and how do NFS and AFS clients contact server?

Why is map-reduce model useful?

What types of application can be expressed with map-reduce?

What does a mapper do? What does a reducer do?

How does the system and GFS support map-reduce?

NFS: STATELESS SERVER WITH CACHED STATS

Upon open of file A:
Contact server to get file handle for future interactions

Upon close, client flush individual blocks of file to server
(Individual blocks may be flushed before this point)

Write():
Keep data local (as long as sufficient space)

Read():
Is this block locally cached on this client?
No – Fetch block from server; record time block was fetched
Yes – Have the attributes for this file expired? (every 3 secs)
No – Use locally cached copy of this block
Yes – Send STAT (or getattr) to server
Has the file been modified on server since client's copy?
Yes – Refetch that block from server
No – Use locally cached copy of this block

AFS: CALLBACKS AND WHOLE-FILE CACHING

Upon open of file A:
If file A is cached locally and callback to server still exists, use cached copy
Else, fetch whole file from server (storing in local memory or disk)

Upon close, client flushes file to server (if file was written)

Convenient and intuitive semantics:

- AFS needs to do work only for open/close
  - Only check callback on open, not every read

- reads/writes are local

Use same version of file entire time between open and close

AFS VS NFS PROTOCOLS

Can you summarize the consistency semantics provided by NFSv2?

Time	Client A	Client B	Server Action?
0	fd = open("file A");		
10	read(fd, block1);		
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	
50		write(fd, block1);	
60	read(fd, block1);		
70		close(fd);	
80	read(fd, block1);		
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		
110	read(fd, block1);		
120	close(fd);		

When will server be contacted for NFS? For AFS?
What data will be sent? What will each client see?

NFS PROTOCOL

Time	Client A	Client B	Server Action?
0	fd = open("file A");		lookup()
10	read(fd, block1); read		read
20	read(fd, block2); read		read
30	read(fd, block1); check cache: attr expired getattr(), okay, use local		getattr
31	read(fd, block2); attr not expired, use local		
40		fd = open("file A");	lookup
50		write(fd, block1); keep local	
60	read(fd, block1); attr. expired use local data		getattr()
70		close(fd); write bl to server!	write to disk
80	read(fd, block1); attr expired. get attr. CHANGED FILE - kick out		read()
81	read(fd, block2); not in cache → read		read()
90	close(fd);		
100	fd = open("fileA");		lookup
110	read(fd, block1); attr expire; get new attr local ok		getattr
120	close(fd);		

AFS PROTOCOL

Time	Client A	Client B	Server Action?
0	fd = open("file A");		setup callback for file A
10	read(fd, block1);		send all of file A
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	setup callback
50		write(fd, block1);	send all of A
60	read(fd, block1);		
70		close(fd);	send back changes of A, break callbacks
80	read(fd, block1);		
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		
110	read(fd, block1);		
120	close(fd);		

MAP-REDUCE MOTIVATION

Datasets are too large to process with single thread

Good concurrent programmers are rare

Want concurrent programming framework that is:

- easy to use (no locks or CVs)
- general (works for many problems)

MAP-REDUCE FRAMEWORK

Google published details in 2004

- Open source implementation: Hadoop
- Co-designed with Google File System (next lecture)

Input: set of key/value pairs

Output: set of key/value pairs

Strategy:
Group data into logical buckets and then compute over each bucket

MAP-REDUCE STRATEGY

First set of processes groups and transforms data into logical buckets

- Mappers

Each bucket has a single process that computes over it

- Reducers

Claim:
If no bucket has too much data, no single process has to do too much work.

MAPREDUCE OVERVIEW

~~Motivation~~

MapReduce Programming

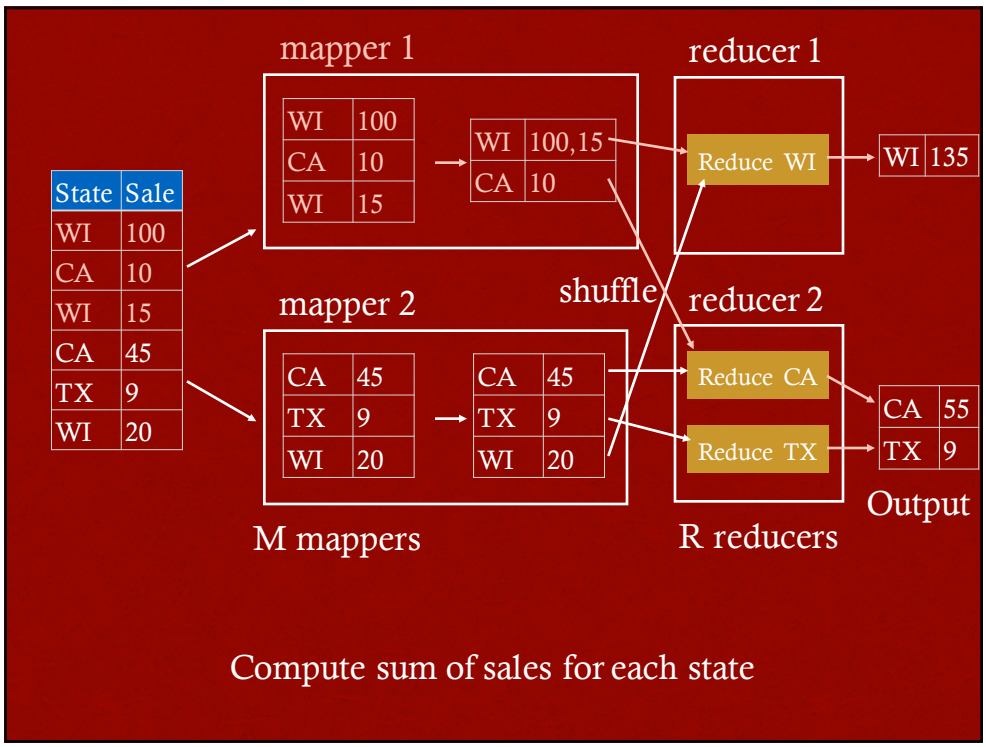
Implementation

EXAMPLE: REVENUE PER STATE

State	Sale	ClientID
WI	100	9292
CA	10	9523
WI	15	9331
CA	45	9523
TX	9	8810
WI	20	9292

How to quickly sum **sales** in every state without any one machine iterating over all results?

Pretend this table is huge…



REVENUE PER STATE

State	Sale	ClientID
WI	100	9292
CA	10	9523
WI	15	9331
CA	45	9523
TX	9	8810
WI	20	9292

Mappers could have grouped by any field desired (e.g., by ClientID)

SQL EQUIVALENTS

```
SELECT sum(sale)
FROM tbl_sales
GROUP BY state;
```

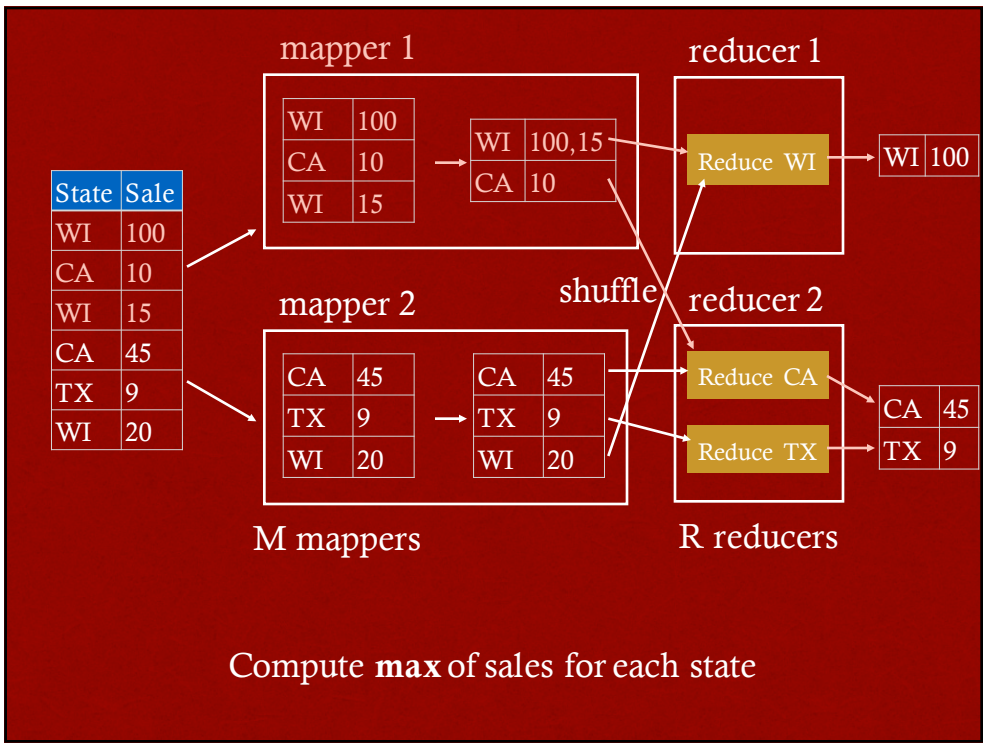
```
SELECT sum(sale)
FROM tbl_sales
GROUP BY clientID;
```

reduce

```
SELECT max(sale)
FROM tbl_sales
GROUP BY clientID;
```

map

HOW TO CHANGE REDUCER?



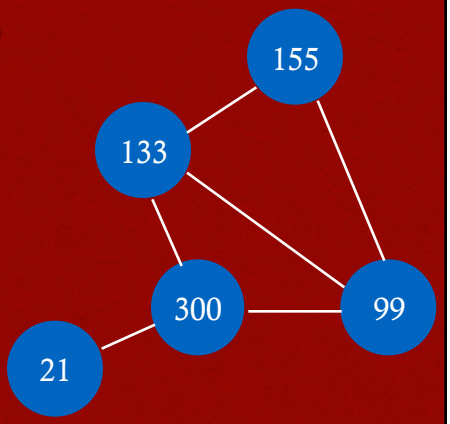
MAPPER OUTPUT

Sometimes mappers simply classify records
(e.g., state revenue)

Sometimes mappers produce multiple intermediate records per input
(e.g., friend counts)

EXAMPLE: COUNTING FRIENDS

friend1	friend2
133	155
133	99
133	300
300	99
300	21
99	155



What is input to each mapper?

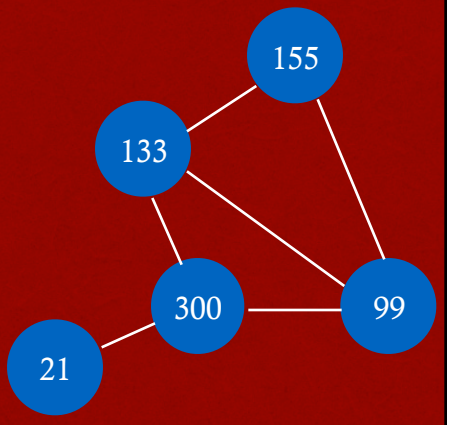
mapper 1

133	155
133	99
133	300

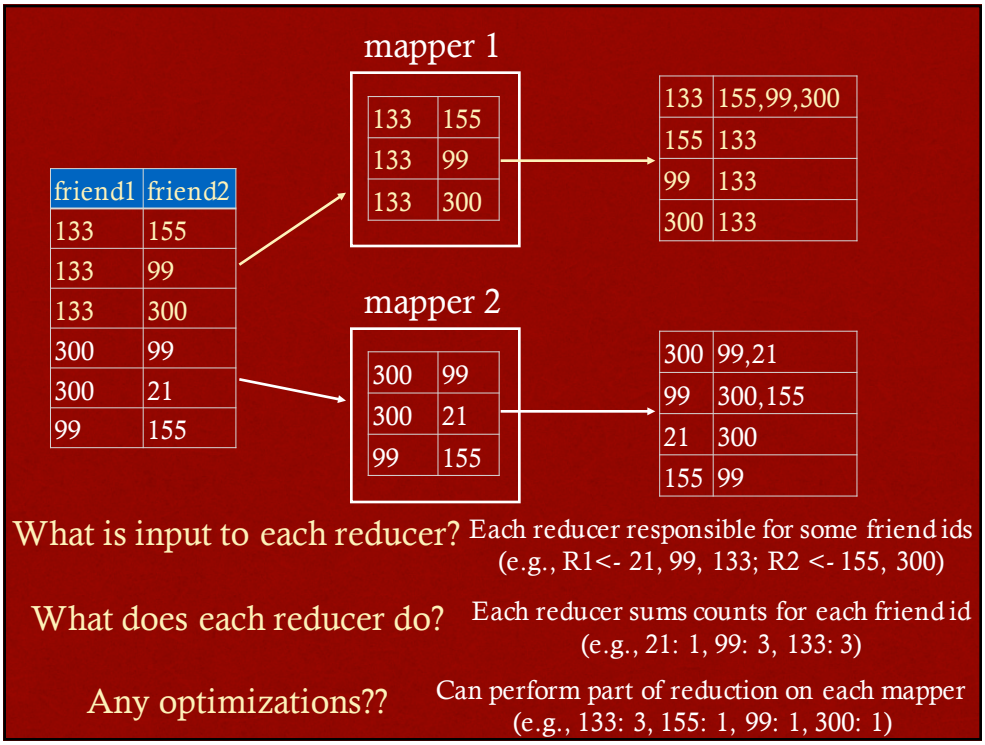
mapper 2

300	99
300	21
99	155

friend1	friend2
133	155
133	99
133	300
300	99
300	21
99	155



What does each mapper produce?



MANY OTHER BIG-DATA WORKLOADS

Distributed grep (over text files)

URL access frequency (over web request logs)

Distributed sort (over strings)

PageRank (over all web pages)

…

MAP/REDUCE FUNCTION TYPES

```
map(k1,v1) -> list(k2,v2)

reduce(k2,list(v2)) -> list(k3,v3)
```

HADOOP API

```
map(k1,v1) -> list(k2,v2)

reduce(k2,list(v2)) -> list(k3,v3)
```

```
public void map(LongWritable key, Text value) {

      // WRITE CODE HERE

}

public void reduce(Text key, Iterator<IntWritable>
values) {

      // WRITE CODE HERE

}
```

WHAT DOES THIS DO?

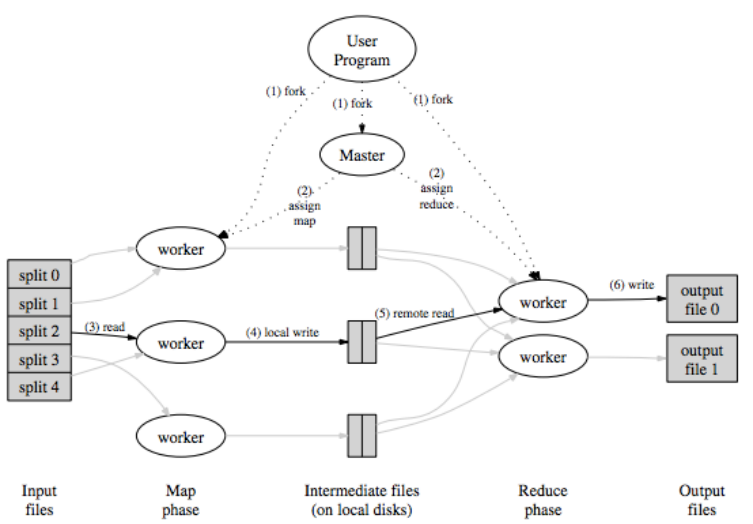
```
map(k1,v1) -> list(k2,v2)
public void map(LongWritable key, Text value) {
  String line = value.toString();
  StringToke st = new StringToke(line);
  while (st.hasMoreTokens())
      output.collect(st.nextToken(), 1);
}
reduce(k2,list(v2)) -> list(k3,v3)
public void reduce(Text key,
                   Iterator<IntWritable> values) {
  int sum = 0;
  while (values.hasNext())
      sum += values.next().get();
  output.collect(key, sum);
```

MAPREDUCE OVERVIEW

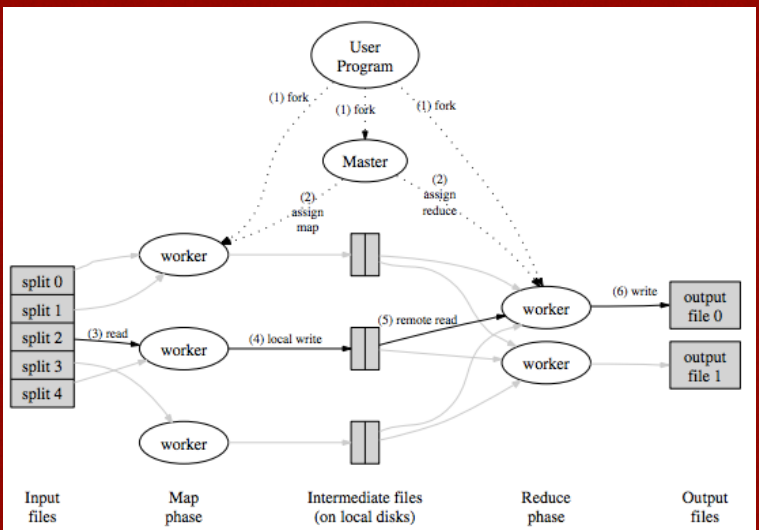
~~Motivation~~

~~MapReduce Programming~~

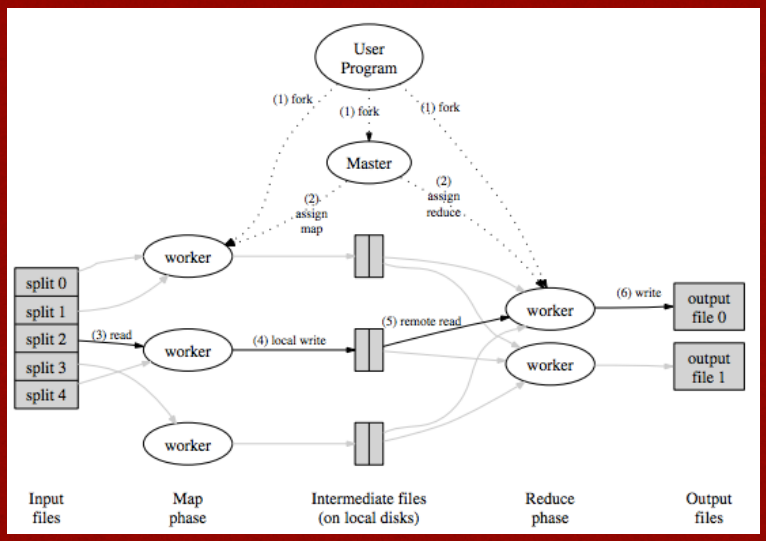
Implementation



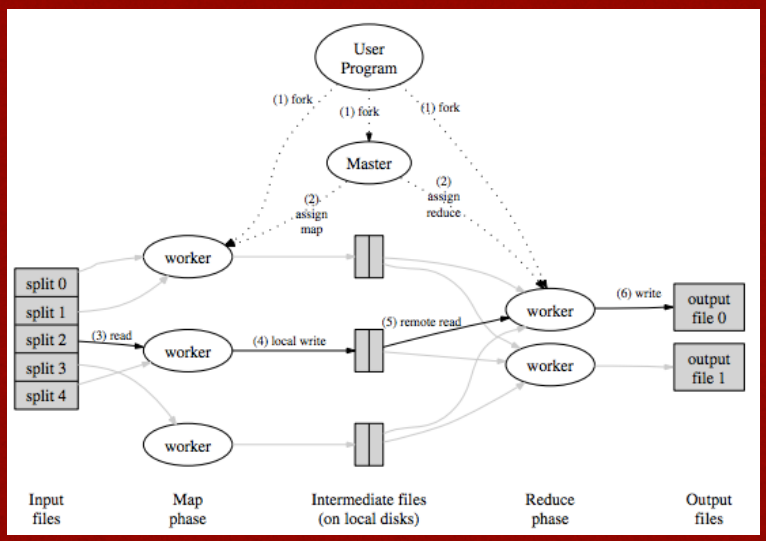
1. Split input files into M pieces of 16 MB-64 MB. Start up many copies of program on cluster of machines



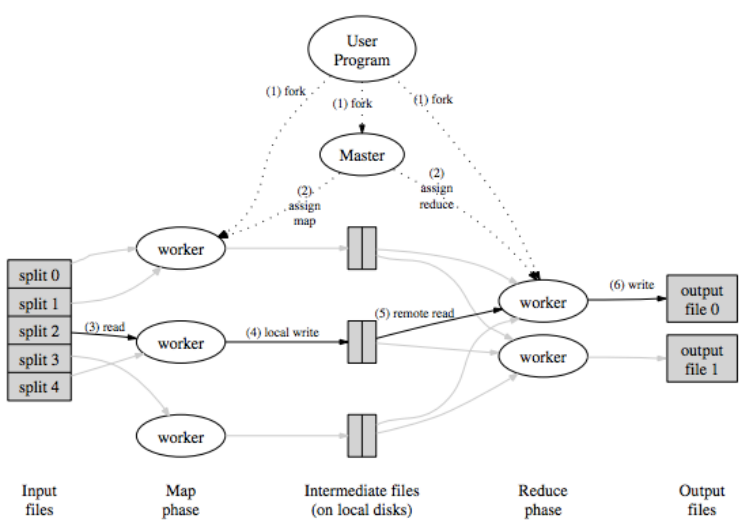
2. Master picks idle workers and assigns each map task or a reduce task (portion of key space)



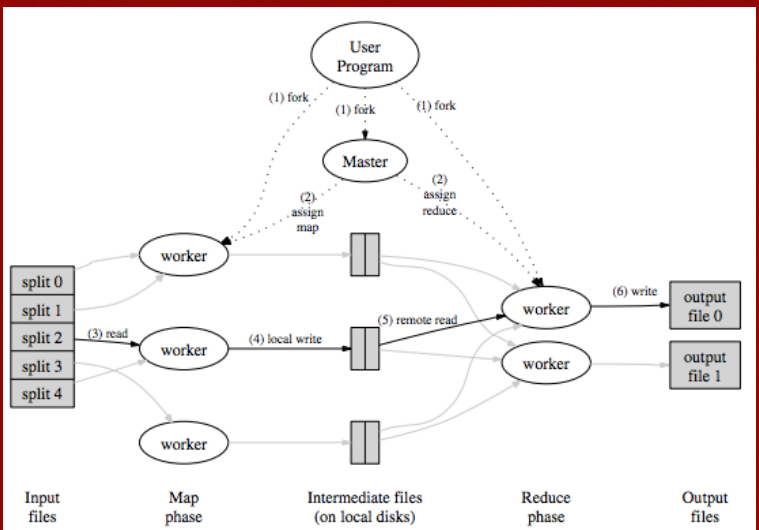
3. Mapper **reads()** contents of corresponding input split. Parses key/value pairs and passes each pair to Map() function. Intermediate key/value pairs buffered in memory.



4. Periodically, buffered pairs written to local disk, partitioned into R regions. Locations are passed to master, who forwards these locations to reducers.



5. Shuffle: After reducer is notified of locations, uses RPC to read data from workers' disks. When reducer has all intermediate data, sorts data so same keys are adjacent.



6. Reducers iterate over sorted intermediate data; pass each unique key and values to Reduce function. Output is **appended** to final output file for this reduce partition.

MAPREDUCE OVER GFS

MapReduce writes/reads data to/from GFS (next lecture)

GFS makes 3 replicas of each file

MapReduce workers run on same machines as GFS workers

Why not store intermediate files in GFS?

Don't need to access outside map-reduce job

Don't need replication for long-term life-time

What if machine holding local files dies?

Re-run mapper to generate output again

MAPREDUCE OVER GFS

MapReduce writes/reads data to/from GFS (next lecture)

GFS makes 3 replicas of each file

MapReduce workers run on same machines as GFS workers

Which edges involve network I/O? Edges 3+4. Maybe 1.

How to avoid I/O for 1? Place mapper on same machine as one of the GFS replicas

EXPOSING LOCATION

GFS exposes which servers store which files
(not transparent, but very useful!)

Hadoop example:

```
BlockLocation[]

getFileBlockLocations(Path  p, long start, long len);
```

Spec: return an array containing hostnames,
offset and size of portions of the given file.

MAPREDUCE POLICY

MapReduce needs to decide which machines to use for map and reduce tasks

Potential factors?

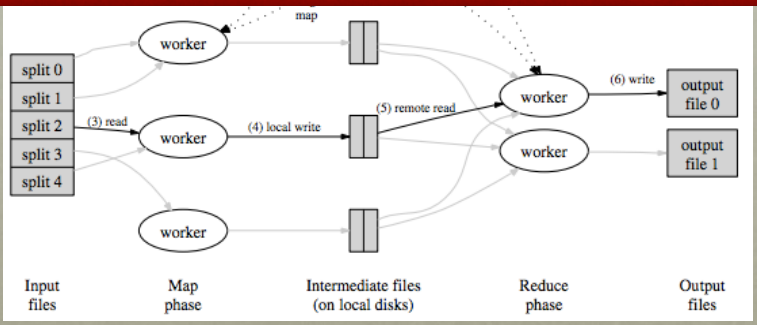
- try to put mappers near one of the three replicas

- for reducers, store one output replica locally

- try to use underloaded machines

- consider network topology

NUMBER OF MAPPERS AND REDUCERS



What does the value of M (number of mappers) influence?

Communication to input file, some disk IO, M to R communication

What if M is too big?

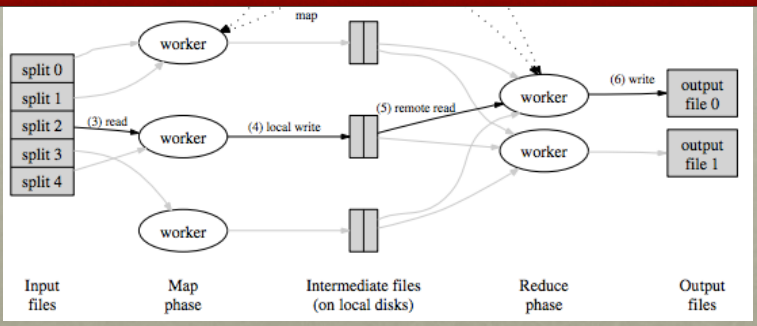
data is really small; too much overhead per data piece

What if M is too small?

less parallelism; affect load balancing. E.g. 5 nodes, 5 map task

Choose M to control size of input data

NUMBER OF MAPPERS AND REDUCERS



What if R is too big?

- Large number of output files

What if R is too small?

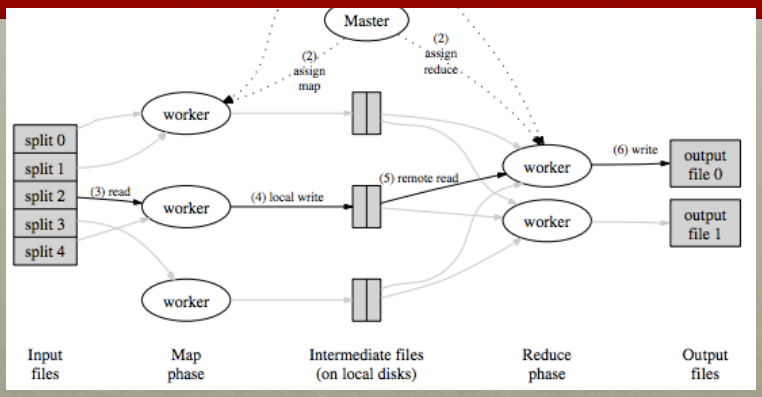
- Not enough parallelism

Goal:

M and R much larger than number of worker machines

- Each worker performing many different tasks improves dynamic load balancing
- Speeds up recovery if worker fails: its many completed map tasks can allocated across many other machines

FAILED TASKS

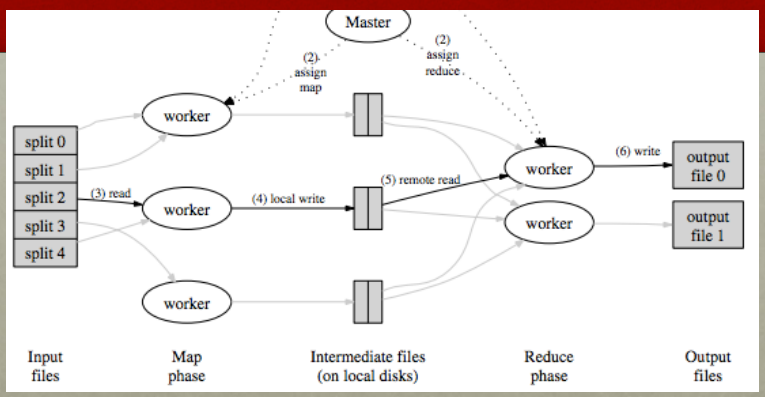


MapReduce master tracks status of all map and reduce tasks

If any tasks don't respond to pings, what should master do?

Restart mapper or reducer tasks on new machines;
Possible because tasks are deterministic and idempotent
System still has all inputs

SLOW TASKS



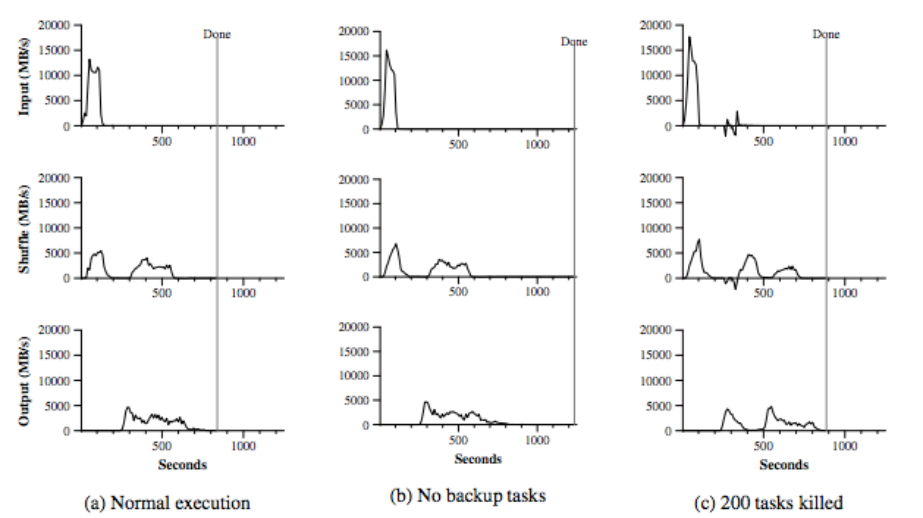
Sometimes machine is overloaded or network link is slow

- With 1000's of tasks, this will always happen

What can master do?

Spawning duplicate tasks when there are only a few stragglers left reduces some job times by 30%

MAP-REDUCE PERFORMANCE



MAPREDUCE SUMMARY

MapReduce makes concurrency easy!

Limited programming environment, but works for a fairly wide variety of applications

Machine failures are easily handled