Welcome back!

CS 744: MAPREDUCE

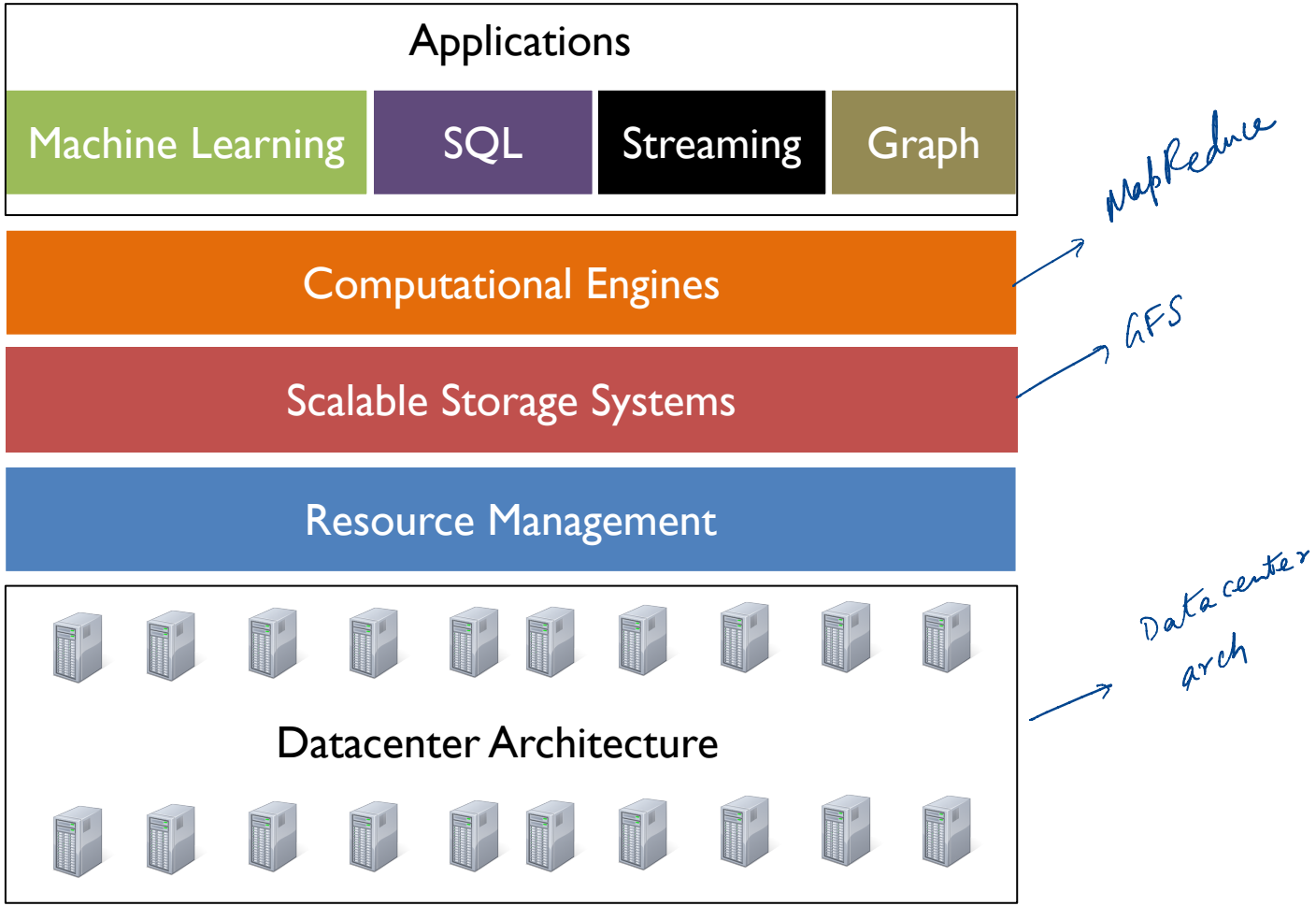
Shivaram Venkataraman

Fall 2020

ANNOUNCEMENTS

- Assignment 1 deliverables
  - Code (comments, formatting)
  - Report → similar to evaluation sections in papers
    - Partitioning analysis (graphs, tables, figures etc.)
    - Persistence analysis (graphs, tables, figures etc.)
    - Fault-tolerance analysis (graphs, tables, figures etc.)

- See Piazza for Spark installation


Applications
Machine Learning
SQL
Streaming
Graph
Computational Engines
Scalable Storage Systems
Resource Management
Datacenter Architecture
MapReduce
GFS
Data center arch

BACKGROUND: PTHREADS

limited set of args

```
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Hello World\n");
    return NULL;
}

int main()
{
    pthread_t thread_id_1, thread_id_2;
    pthread_create(&thread_id_1, NULL, myThreadFun, NULL);
    pthread_create(&thread_id_2, NULL, myThreadFun, NULL);
    pthread_join(thread_id_1, NULL);
    pthread_join(thread_id_2, NULL);
    exit(0);
}
```

execute this in parallel

create two threads

wait for both of these child threads

1. shared memory heap

2. Locks, CVs
   ↳ Synchronization across threads

Single core → Multi core

BACKGROUND: MPI

0, 1, 2 ... , n-1

ranks for n processes

~ 80s

Supercomputing

fine grained synchronization

```
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Print off a hello world message
    printf("Hello world from rank %d out of %d processors\n",
           world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
mpirun -n 4 -f host_file ./mpi_hello_world
```

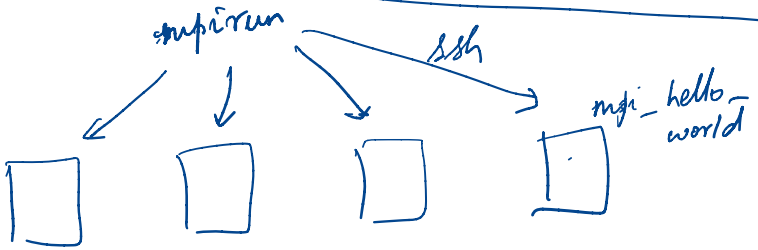
4 processes (-n)

get total number of processes

serial program

hosts for this job are in host_file

MPI_Send to send a message to another process



MOTIVATION

Build Google Web Search

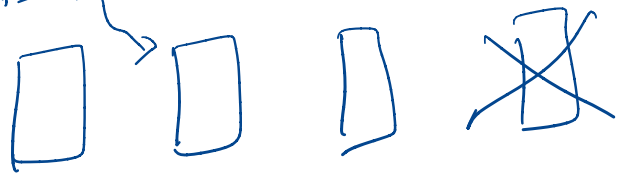
- Crawl documents, build inverted indexes etc.

Need for

- automatic parallelization → Didn't want programmers to worry about it
- network, disk optimization
- handling of machine failures

mpirun → crashes

MPI



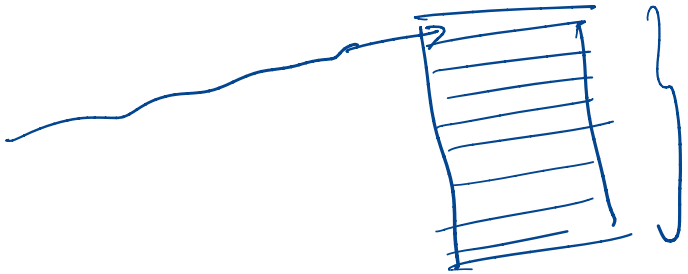
OUTLINE

- Programming Model
- Execution Overview
- Fault Tolerance
- Optimizations

PROGRAMMING MODEL

collection of records

Data type: Each record is (key, value)



log file
↳ record is log entry

crawl
↳ record is web page

Map function:

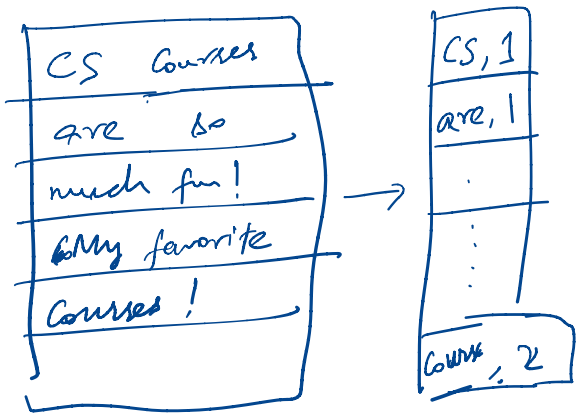
$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

Reduce function:

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

EXAMPLE: WORD COUNT

input to map function




```
def mapper(line):
    for word in line.split():
        output(word, 1)
```

intermediate key

intermediate value

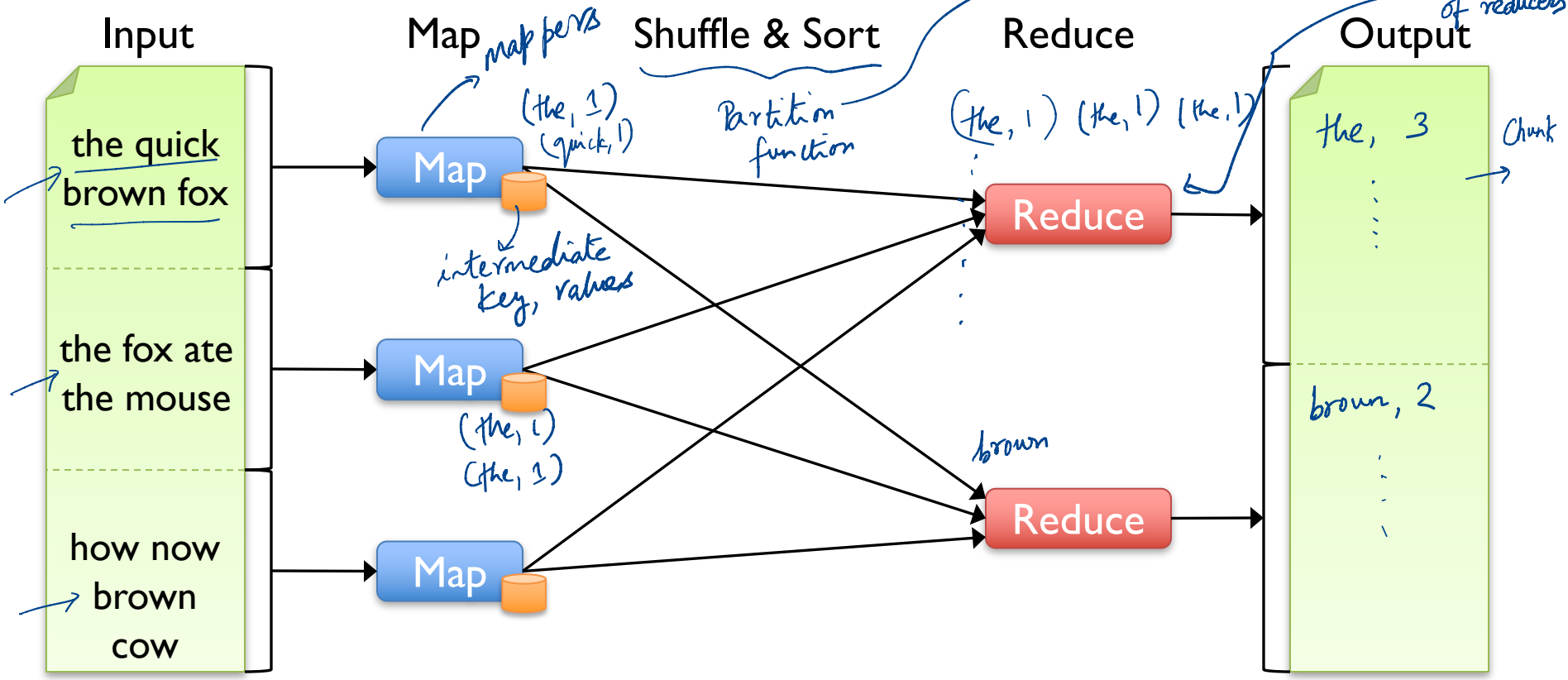
```
def reducer(key, values):
    output(key, sum(values))
```

courses , list(1,1) → (courses, 2)

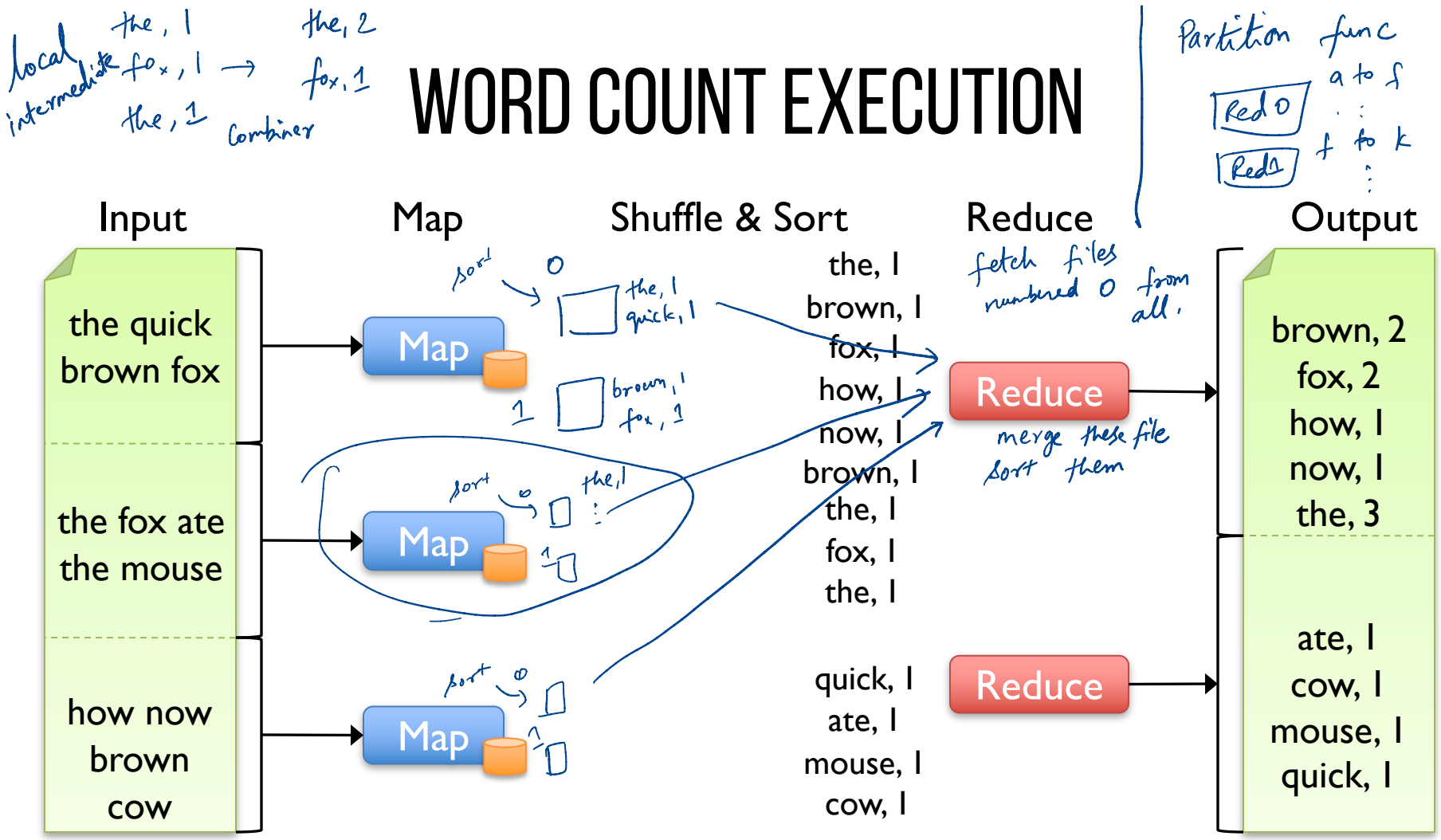
GFS
↳ data is chunked
shared

WORD COUNT EXECUTION

hashing
the → 100 % 2 = 0
brown → 535 % 2 = 1
number of reducers



WORD COUNT EXECUTION



ASSUMPTIONS

1. Failures are norm → Only have tasks
   No shared memory or message passing
   other than intermediate KVs.

2. Local storage (disk) is cheap & abundant

3. Applications can be written in this model

4. Input is splittable (records collection)

ASSUMPTIONS

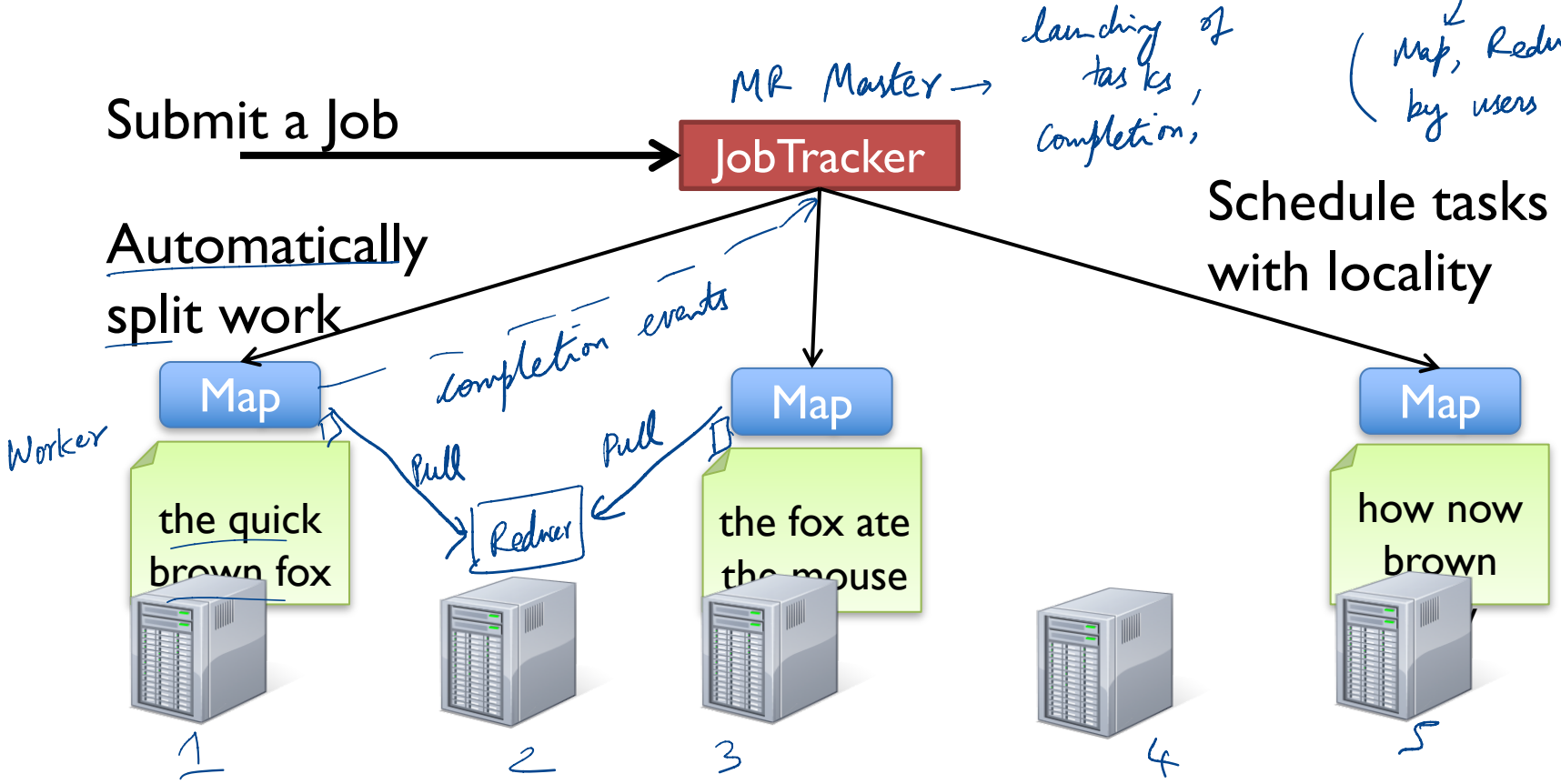
1. Commodity networking, less bisection bandwidth
2. Failures are common
3. Local storage is cheap
4. Replicated FS

WORD COUNT EXECUTION

MapReduce frameworks

MR job

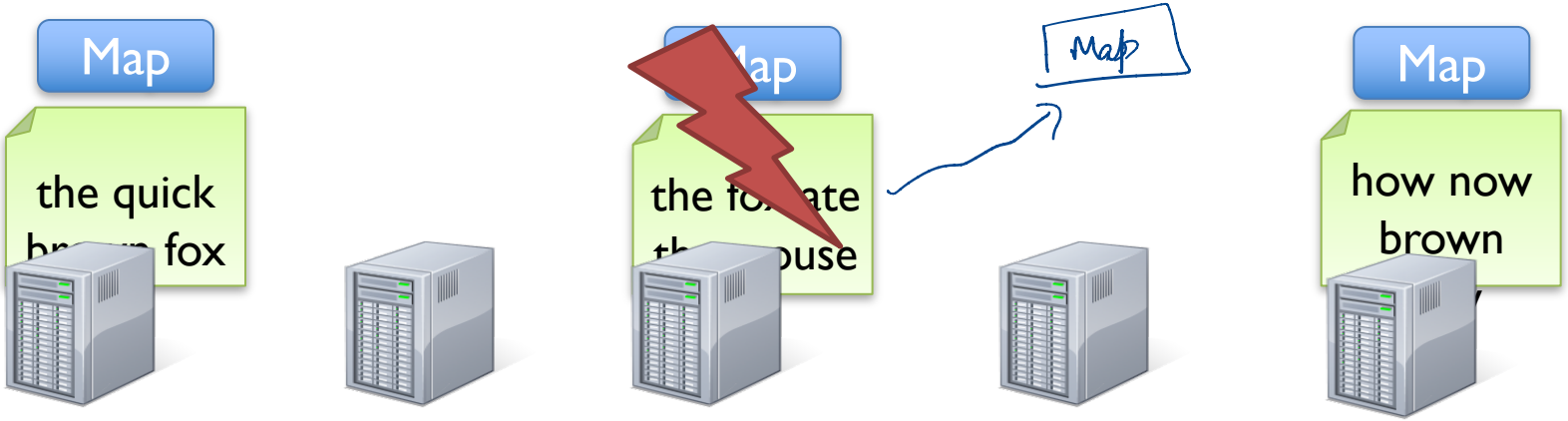
(Map, Reduce tasks by users)



FAULT RECOVERY

If a task crashes:

- Retry on another node → Input can be still read (replication)
- If the same task repeatedly fails, end the job

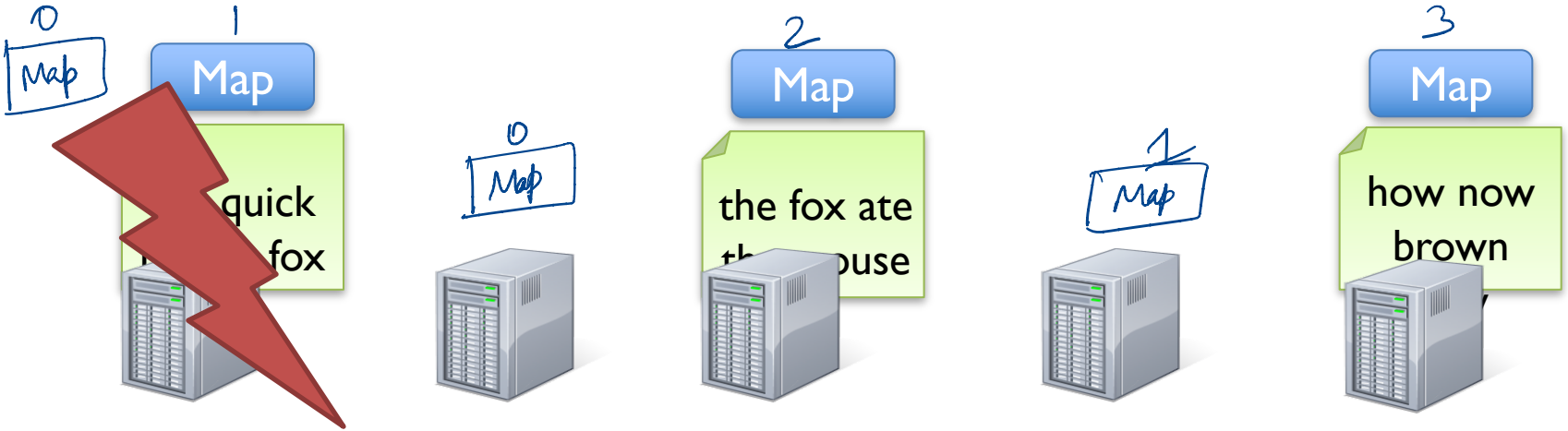


FAULT RECOVERY

If a node crashes:

- Relaunch its current tasks on other nodes

What about task inputs ? File system replication



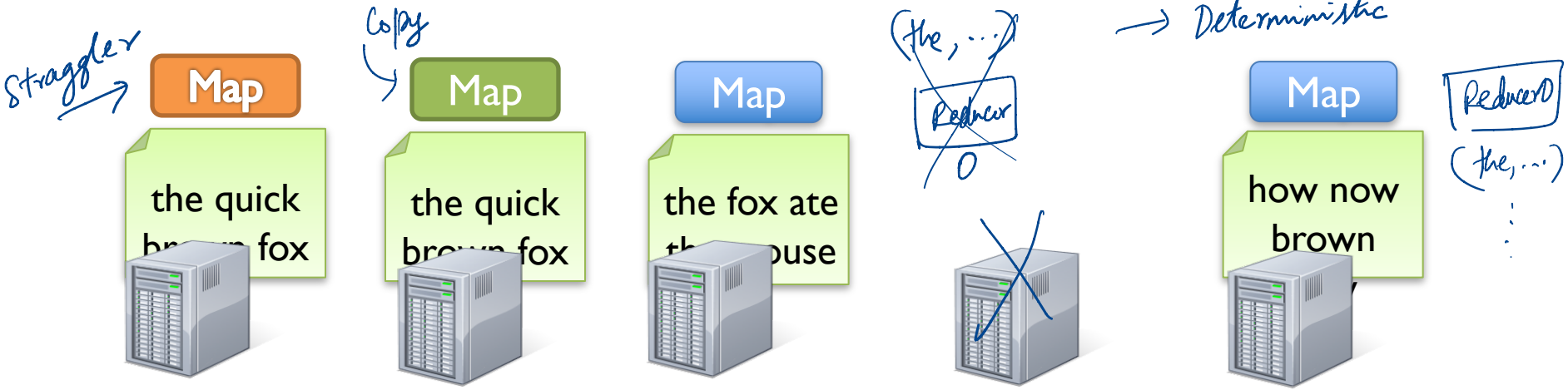
FAULT RECOVERY

→ a task runs much slower than other mappers

If a task is going slowly (straggler):

- Launch second copy of task on another node
- Take the output of whichever finishes first

Assumption
→ something about the node is making it slow
→ Deterministic



MORE DESIGN

Master failure → checkpoints

retry the job → probability of master failing

GFS Master → Metadata

Locality → Map tasks where GFS chunks are

Task Granularity

MAPREDUCE: SUMMARY

- Simplify programming on large clusters with frequent failures
- Limited but general functional API
  - Map, Reduce, Sort
  - No other synchronization / communication
- Fault recovery, straggler mitigation through retries

Intermediate data

"Pull data" from Mapper disk

vs.

"Push data"

DISCUSSION

https://forms.gle/mAHD4QuMXko7vnjB6

DISCUSSION

List one similarity and one difference between MPI and MapReduce

	MPI	MapReduce
Similarity	High perf Parallel Computing	=
Diff	Expressive programming Model	Limited
	Fault tolerance	
	Fine grained Message Passing (network)	Storage Intermediate KV pairs

DISCUSSION

Indexing pipeline where you start with HTML documents. You want to index the documents after removing the most commonly occurring words.

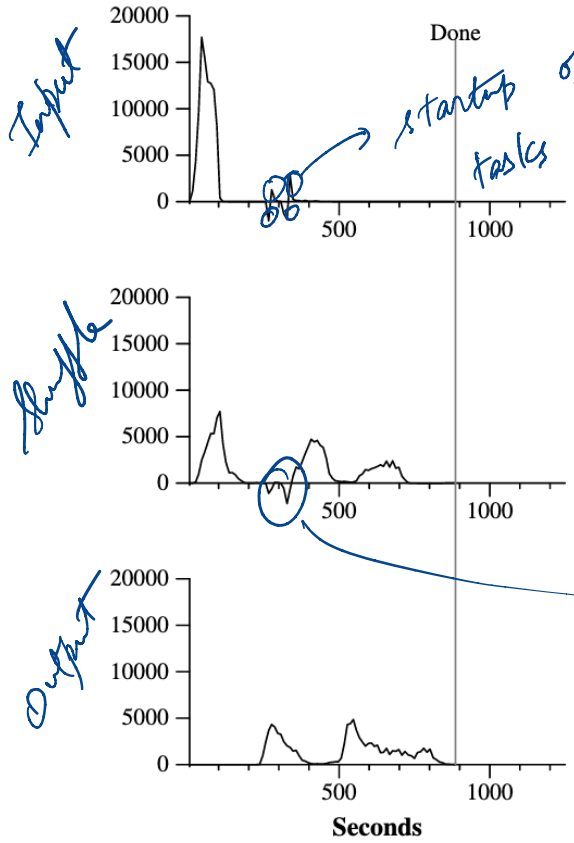
1. Compute most common words. (the, a, ...)
2. Remove them and build the index.

What are the main shortcomings of using MapReduce to do this?

① How to access the list of common words →

② Both MR ops read same inputs

↳ Compose to avoid repeated disk I/O



(c) 200 tasks killed

startup of tasks is fast

Failures resiliency
↳ Barely any longer than without failures

200 process killed

if map is redone also need to redo parts of the shuffle.

MapReduce Usage Statistics Over Time

	Aug, ‘04	Mar, ‘06	Sep, '07	Sep, ’09
Number of jobs	29K	171K	2,217K	3,467K
Average completion time (secs)	634	874	395	475
Machine years used	217	2,002	11,081	25,562
Input data read (TB)	3,288	52,254	403,152	544,130
Intermediate data (TB)	758	6,743	34,774	90,120
Output data written (TB)	193	2,970	14,018	57,520
Average worker machines	157	268	394	488

Jeff Dean, LADIS 2009

NEXT STEPS

- Next lecture: Spark
- Assignment 1: Use Piazza!

grep program → 15,000 inputs
1 reducer

Shuffle & Sort