CS 744: MAPREDUCE

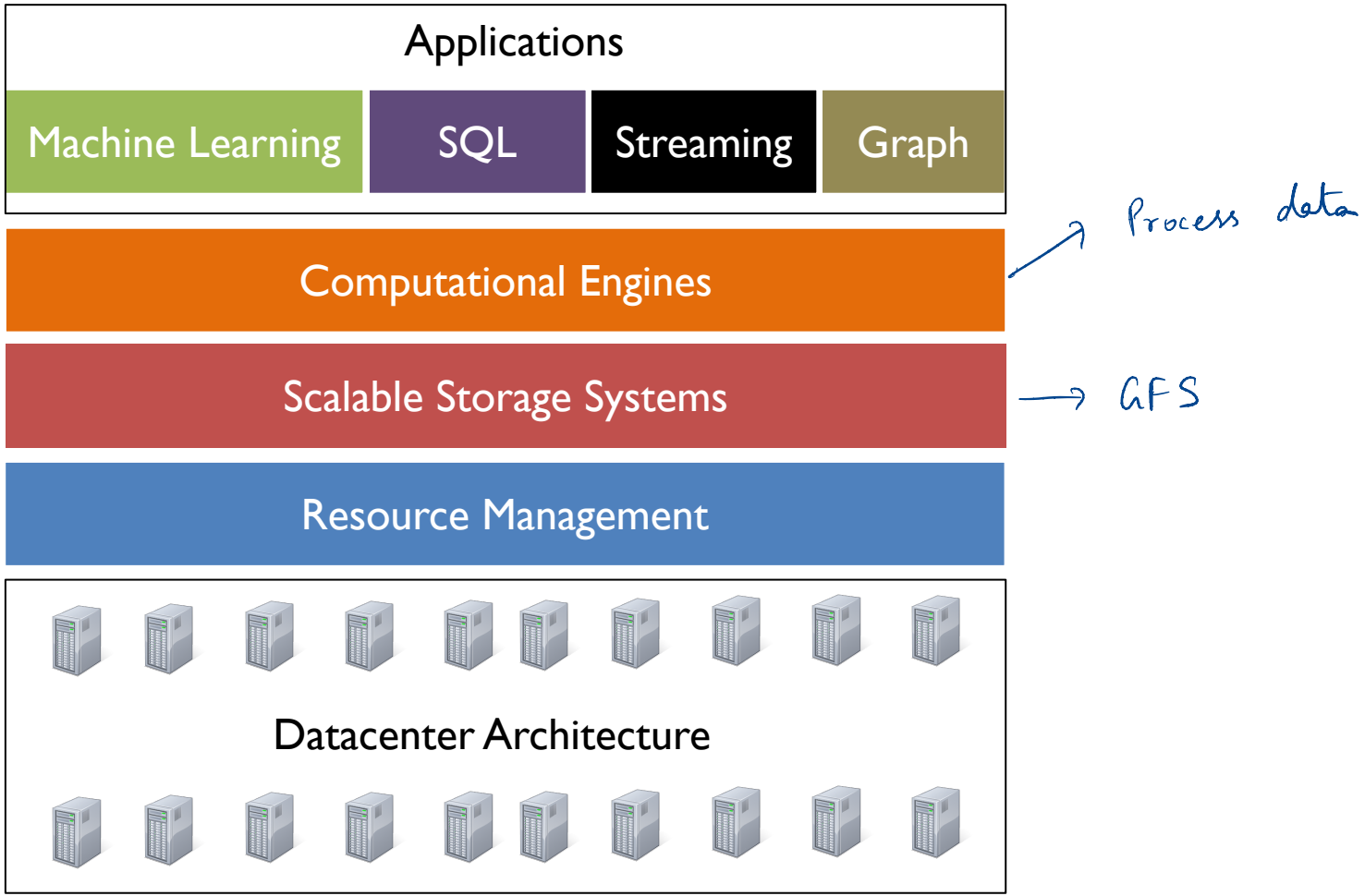
Shivaram Venkataraman

Fall 2021

ANNOUNCEMENTS

- Assignment 1 deliverables
  - Code (comments, formatting)
  - Report
    - Partitioning analysis (graphs, tables, figures etc.)
    - Persistence analysis (graphs, tables, figures etc.)
    - Fault-tolerance analysis (graphs, tables, figures etc.)

- See Piazza for Tutorial


Applications
Machine Learning
SQL
Streaming
Graph
Computational Engines
Process data
Scalable Storage Systems
GFS
Resource Management
Datacenter Architecture

BACKGROUND: PTHREADS

```
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);
}
```

- function run by thread
- threads share memory file descriptors

locks
Conditional variables

→ Creating a new thread

BACKGROUND: MPI

machines to launch processes

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

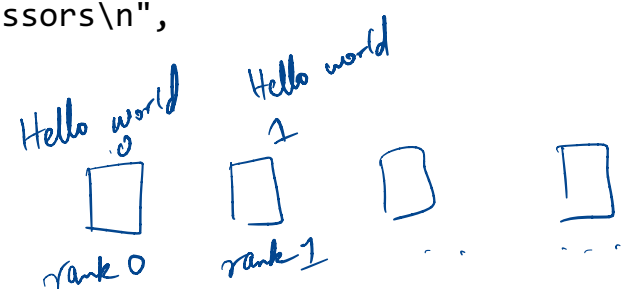
address space is not shared

MPI send, recv

```
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();
}
```

world_size → 4

rank = index of the process



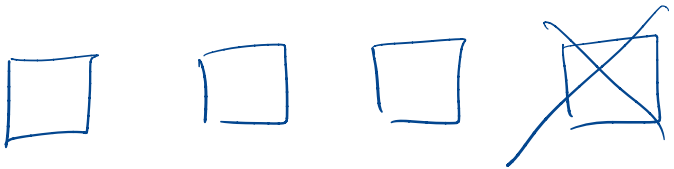
MOTIVATION

Build Google Web Search

- Crawl documents, build inverted indexes etc.

Need for

- automatic parallelization → Automate how many processes and where they are run
- network, disk optimization
- handling of machine failures



OUTLINE

- Programming Model
- Execution Overview
- Fault Tolerance
- Optimizations

PROGRAMMING MODEL

GFS

Data type: Each record is (key, value)

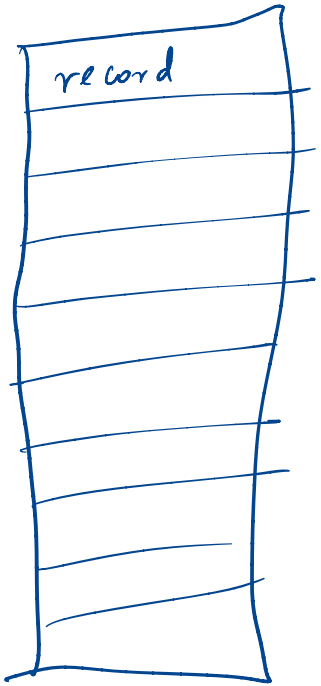
Map function:

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

Reduce function:

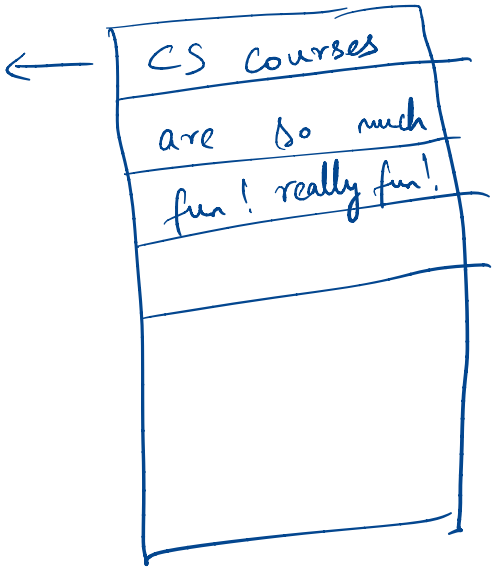
group

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



EXAMPLE: WORD COUNT

each line is input to map




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

(CS, 1) (courses, 1)

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

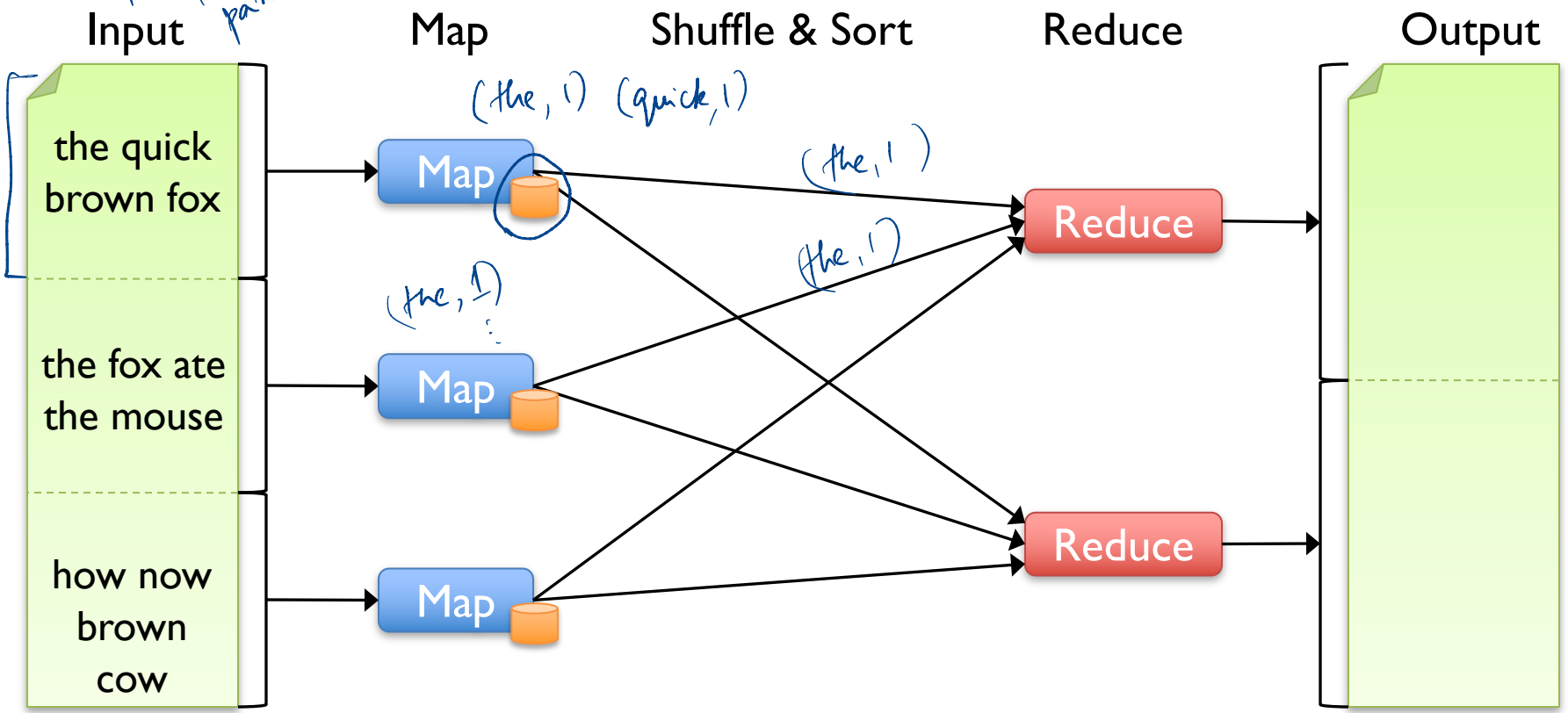
CS → key

(CS, 3)

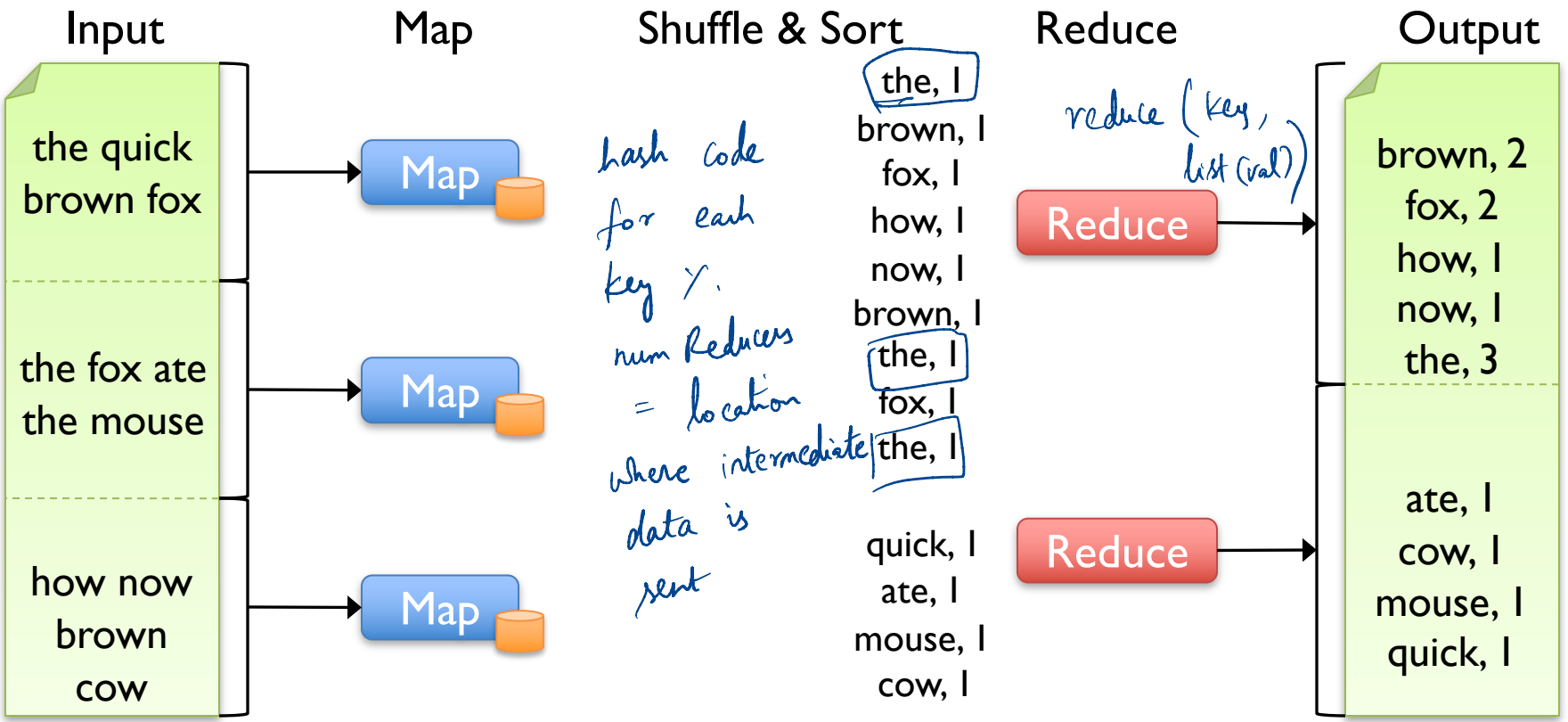
[1, 1, 1...] → values

WORD COUNT EXECUTION: PART 1

each block of data is operated on in parallel!



WORD COUNT EXECUTION: PART2



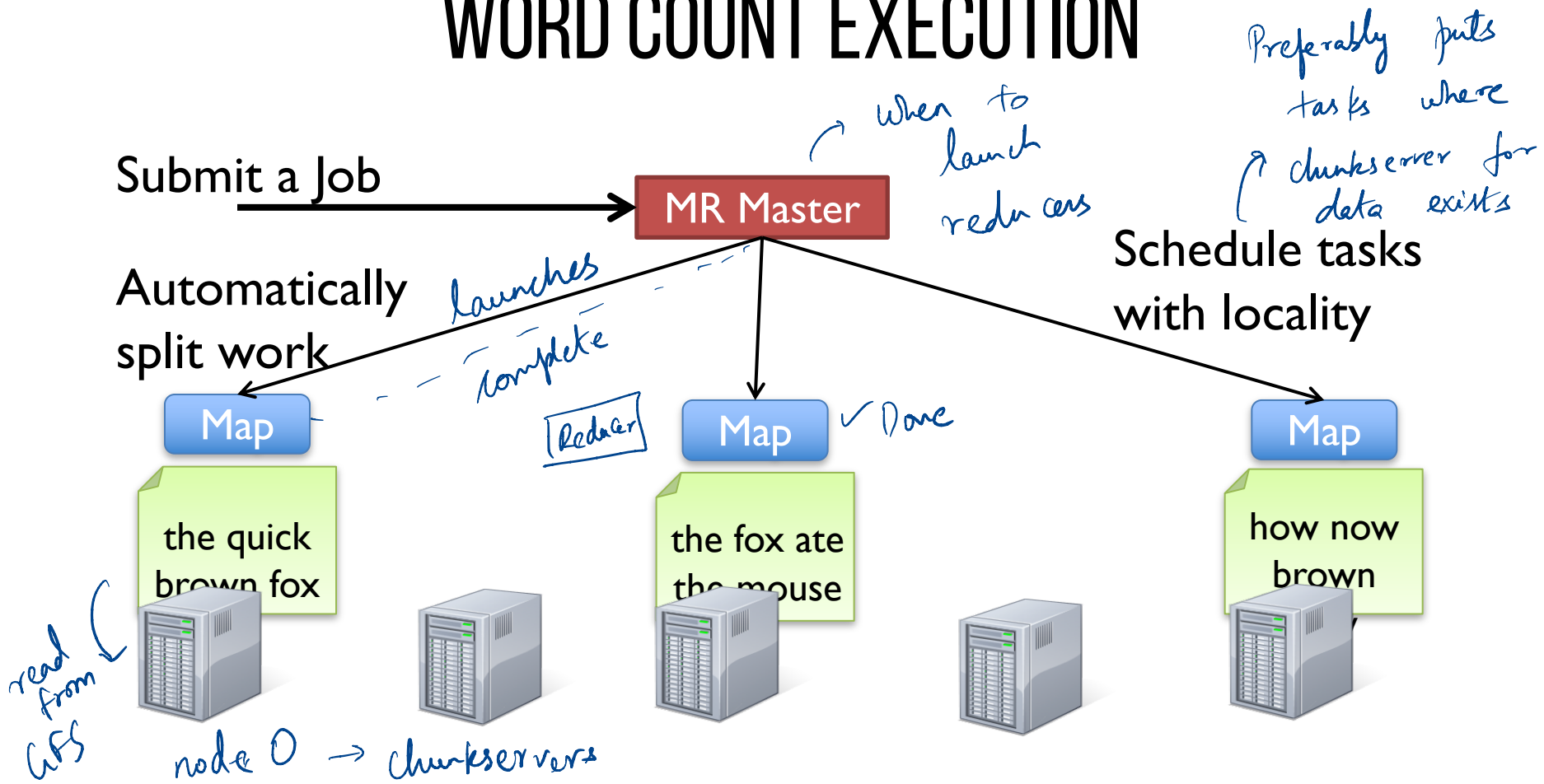
ASSUMPTIONS

1. Failures are the norm
2. Inputs need to be splittable
3. Developers can convert/express computations as MapReduce
4. Cluster resources are automatically managed

ASSUMPTIONS

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

WORD COUNT EXECUTION



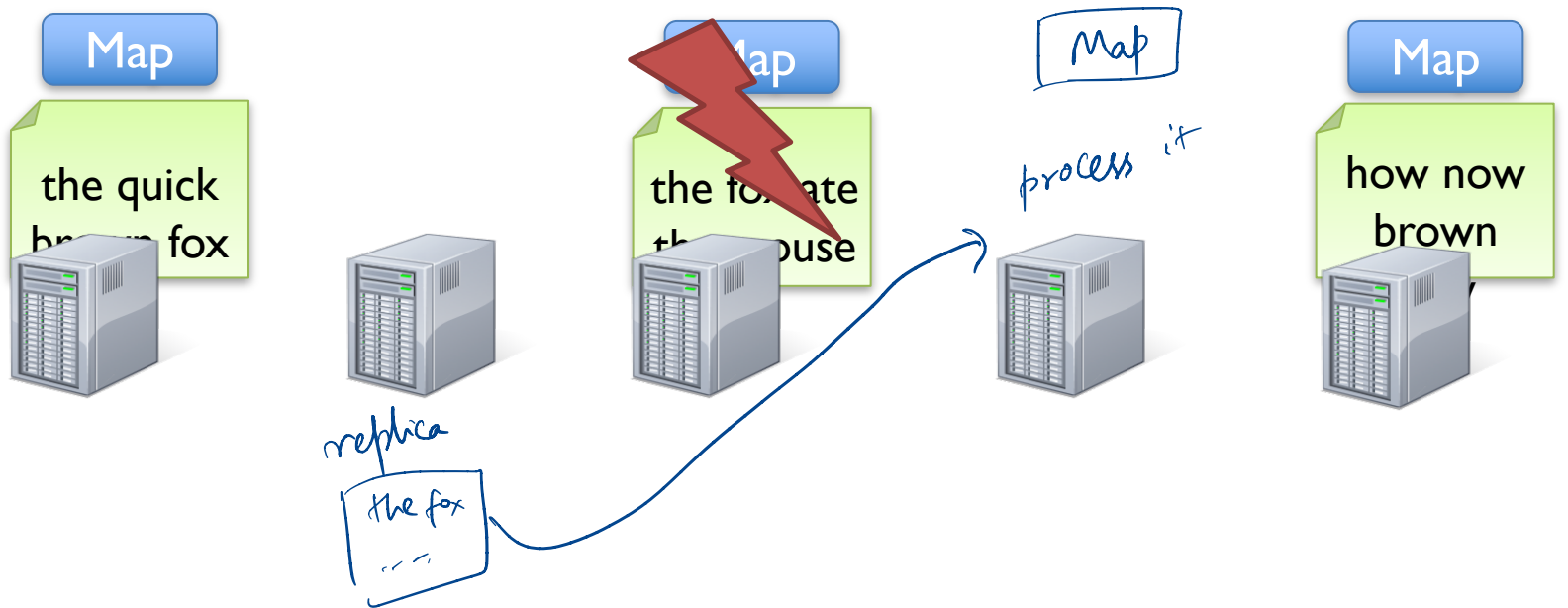
FAULT RECOVERY

Inputs are replicated

Tasks are deterministic

If a task crashes:

- Retry on another node
- If the same task repeatedly fails, end the job

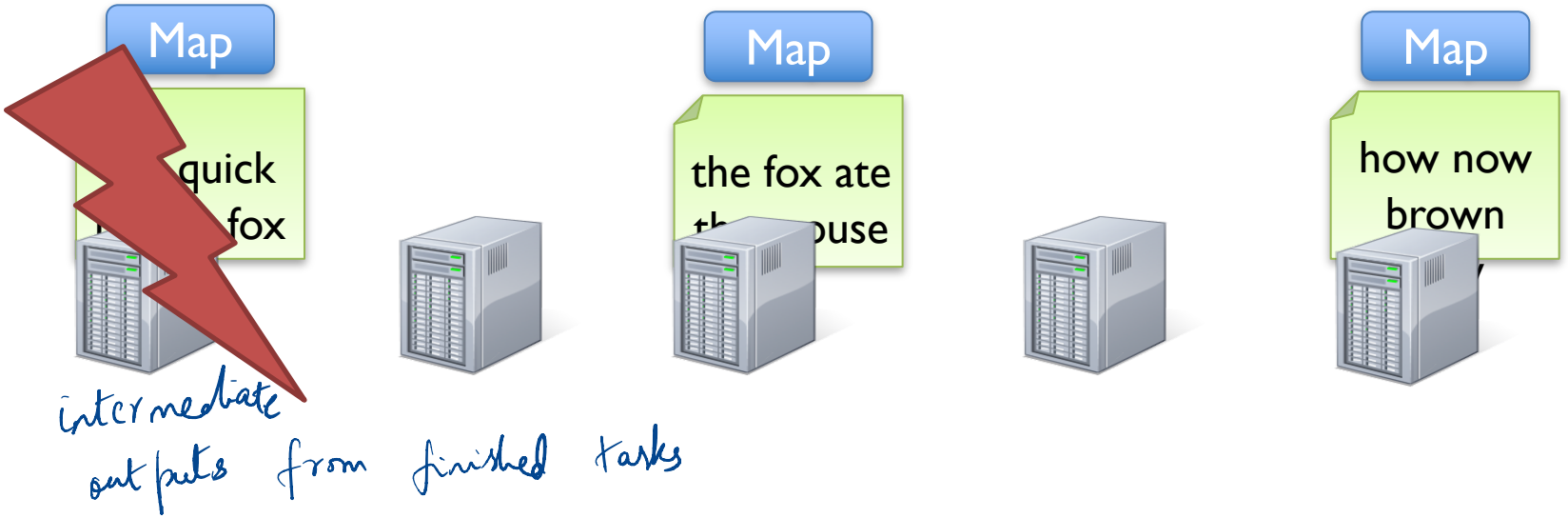


FAULT RECOVERY

If a node crashes:

- Relaunch its current tasks on other nodes

What about task inputs ? File system replication

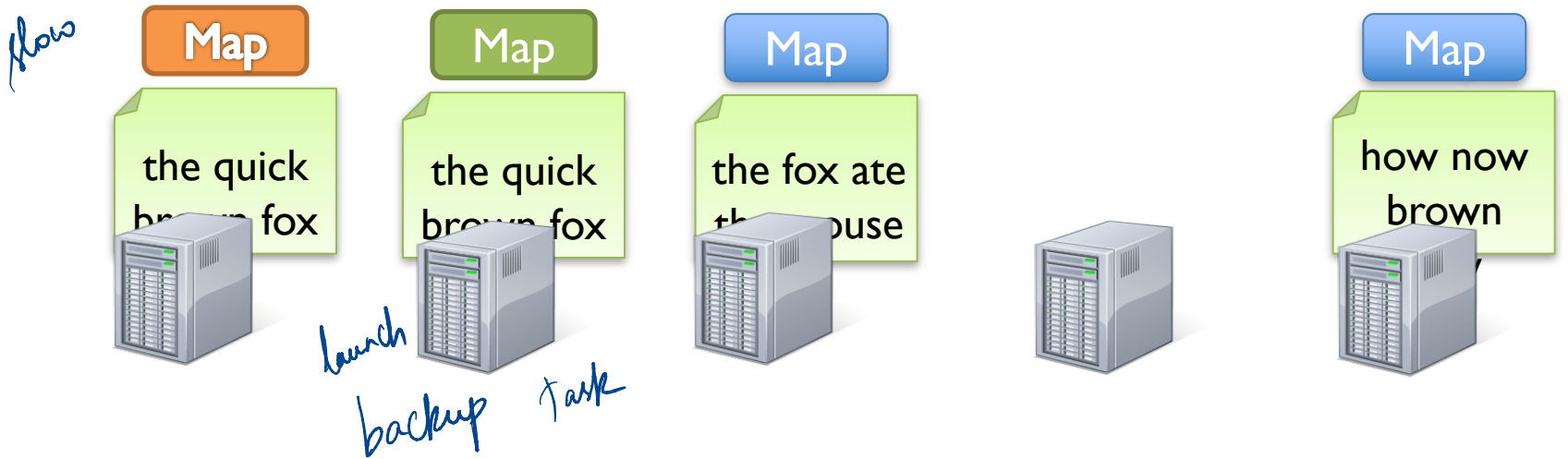


FAULT RECOVERY

some tasks make very slow progress (even for same data)

If a task is going slowly (straggler):

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



MORE DESIGN

Master failure

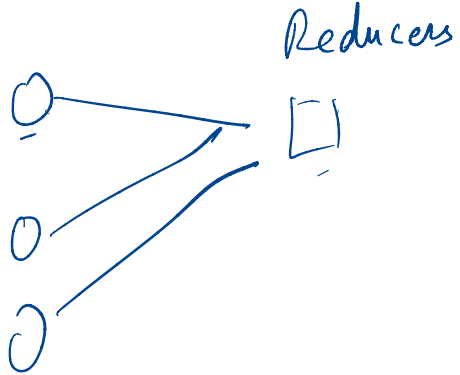
↳ Abort the MR execution

Only 1 master, low probability of failure

Locality

↳ Disk access from chunk server

Reducers ?



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

DISCUSSION

https://forms.gle/prXWmR97A3xozHkH9

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.
2. Remove them and build the index.

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

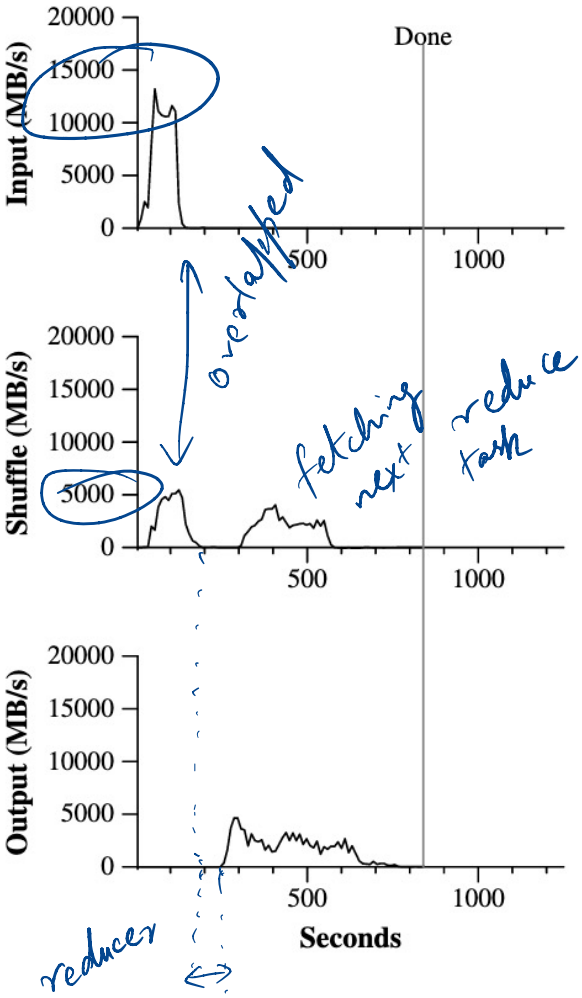
One MapReduce job to compute most common words

Second MR job to build index

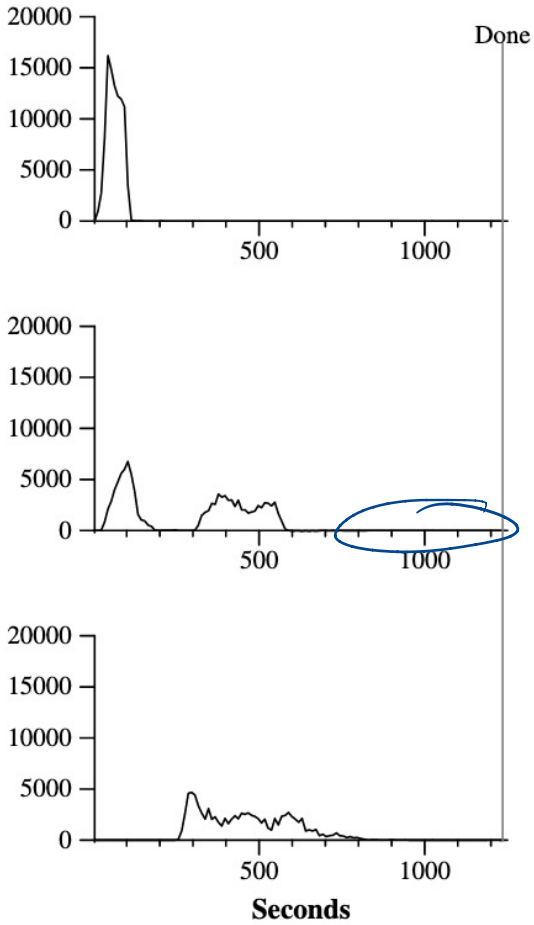
(1) Store the most common words

(2) Read the input data multiple times

Can you combine these two things in one MR?



(a) Normal execution



(b) No backup tasks

(a) & (b) look very similar. but backup tasks do speed up execution

Input rates are much higher

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!