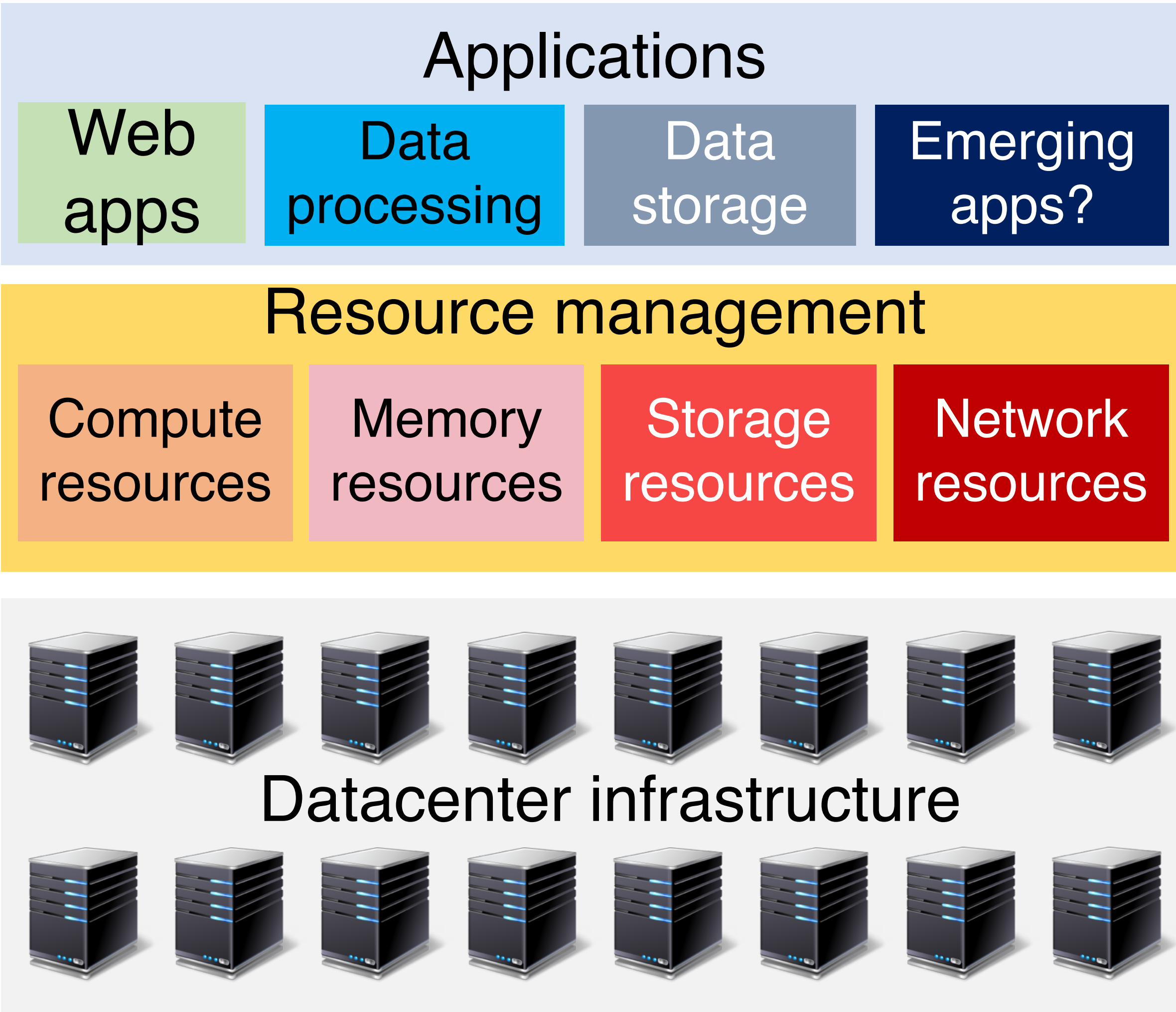
MapReduce and Concurrency in Go

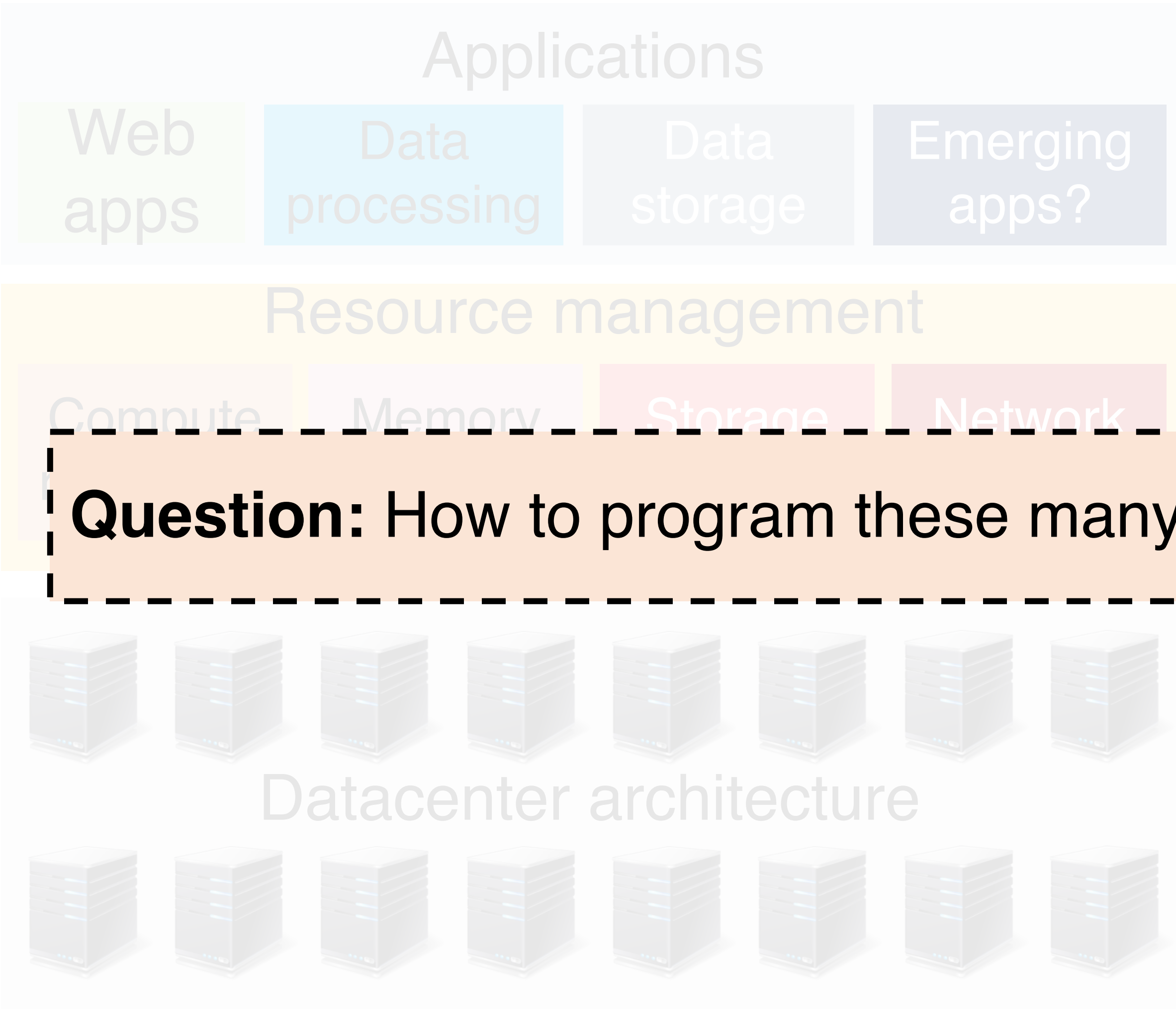
CS 675: Distributed Systems (Spring 2020)

Lecture 3

Yue Cheng

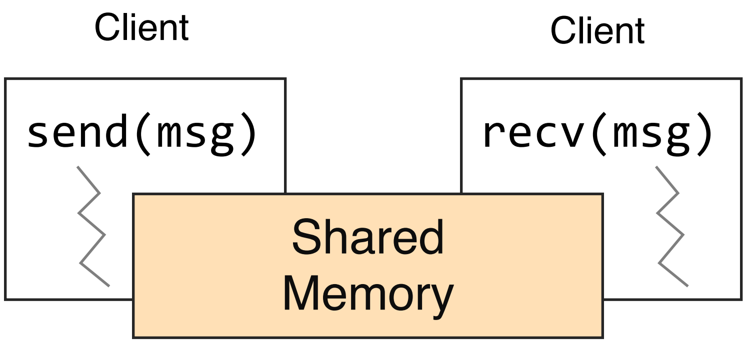
Some material taken/derived from:
- Princeton COS-418 materials created by Michael Freedman and Wyatt Lloyd.
- MIT 6.824 by Robert Morris, Frans Kaashoek, and Nickolai Zeldovich.
- Utah CS6450 by Ryan Stutsman.
- Wisconcin CS744 by Shivaram Venkataraman.


Applications
Web apps
Data processing
Data storage
Emerging apps?
Resource management
Compute resources
Memory resources
Storage resources
Network resources
Datacenter infrastructure



Question: How to program these many computers?

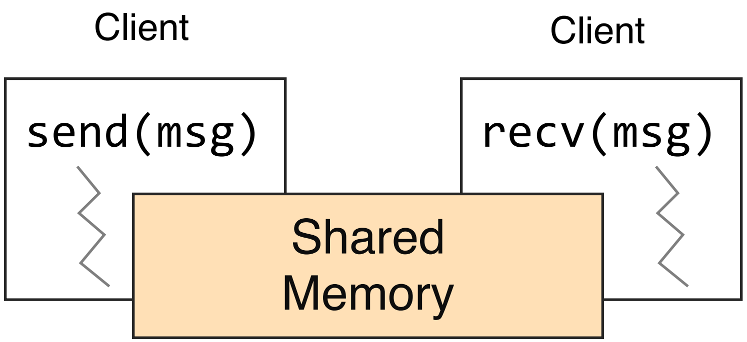
Shared memory



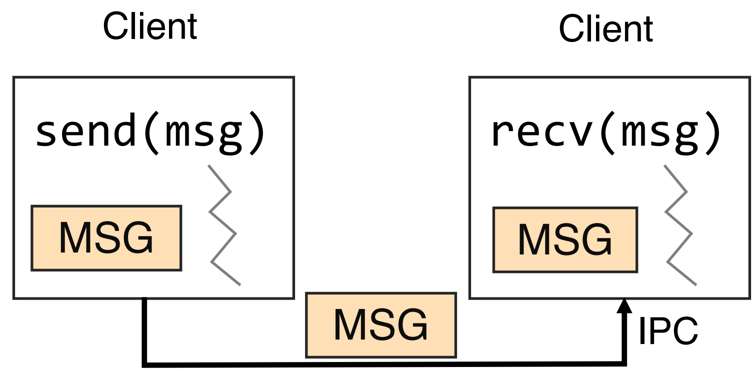
- Shared memory: all multiple processes to share data via memory

- Applications must locate and and map shared memory regions to exchange data

Shared memory vs. Message passing



- Shared memory: all multiple processes to share data via memory
- Applications must locate and and map shared memory regions to exchange data



- Message passing: exchange data explicitly via IPC
- Application developers define protocol and exchanging format, number of participants, and each exchange

Shared memory vs. Message passing

Shared memory vs. Message passing

- Easy to program; just like a single multi-threaded machines
- Hard to write high perf. apps:
  - Cannot control which data is local or remote (remote mem. access much slower)
- Hard to mask failures

- Message passing: can write very high perf. apps
- Hard to write apps:
  - Need to manually decompose the app, and move data
- Need to manually handle failures

Background: Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX (e.g., Linux) OSes

Background: Pthread

```
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);
}
```

Background: MPI

- MPI – Message Passing Interface
  - Library standard defined by a committee of vendors, implementers, and parallel programmers
  - Used to create parallel programs based on message passing

- Portable: one standard, many implementations
  - Available on almost all parallel machines in C and Fortran
  - De facto standard platform for the HPC community

Background: MPI

```
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, workld_size);

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

Background: MPI

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

```
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, workld_size);

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

Today’s outline

1. Google MapReduce

2. Concurrency in Go

The big picture (motivation)

- Datasets are too big to process using a single computer

The big picture (motivation)

- Datasets are too big to process using a single computer
- Good parallel processing engines are rare (back then in the late 90s)

The big picture (motivation)

- Datasets are too big to process using a single computer
- Good parallel processing engines are rare (back then in the late 90s)
- Want a parallel processing framework that:
  - is **general** (works for many problems)
  - is **easy to use** (no locks, no need to explicitly handle communication, no race conditions)
  - can **automatically parallelize** tasks
  - can **automatically handle machine failures**

Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, unreliable hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete before failures

Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, unreliable hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete before failures
- **Key question:** how can every Google engineer be imbued with the ability to write parallel, scalable, distributed, fault-tolerant code?
- **Solution:** abstract out the redundant parts
- Restriction: relies on job semantics, so restricts which problems it works for

Application: Word Count

```
cat data.txt
     | tr –s ‘[[:punct:][:space:]]’ ‘\n’
     | sort | uniq -c
```

```
SELECT count(word), word FROM data
     GROUP BY word
```

Deal with multiple files?

1. Compute word counts from individual files

Deal with multiple files?

1. Compute word counts from individual files

2. Then merge intermediate output

Deal with multiple files?

1. Compute word counts from individual files
2. Then merge intermediate output
3. Compute word count on merged outputs

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

2. Then merge intermediate output

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

2. Then merge intermediate output

3. Compute word count on merged intermediates

MapReduce: Programming interface

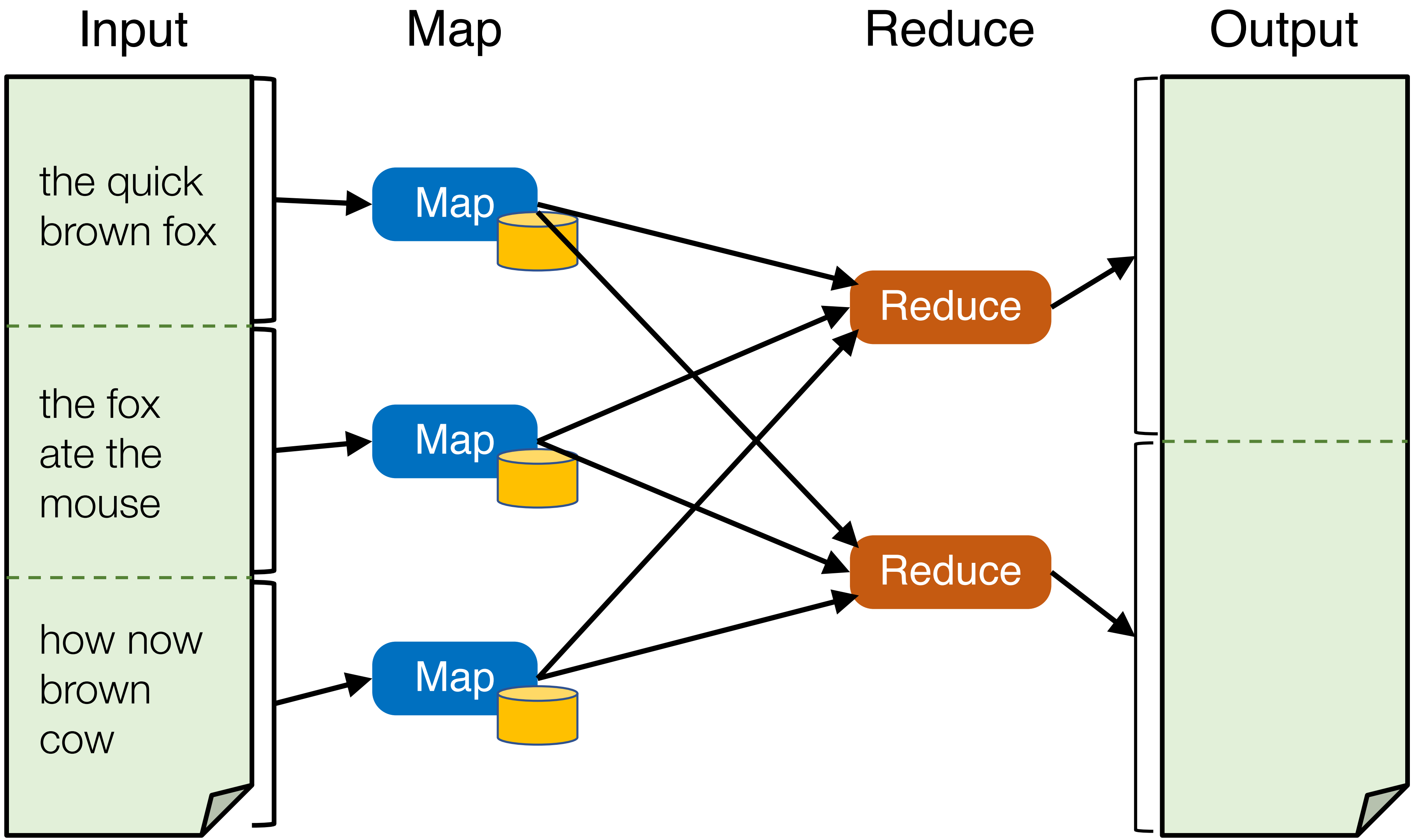
- `map(k1, v1)` → `list(k2, v2)`
  - Apply function to (`k1, v1`) pair and produce set of intermediate pairs (`k2, v2`)

- `reduce(k2, list(v2))` → `list(k3, v3)`
  - Apply aggregation (reduce) function to values
  - Output results

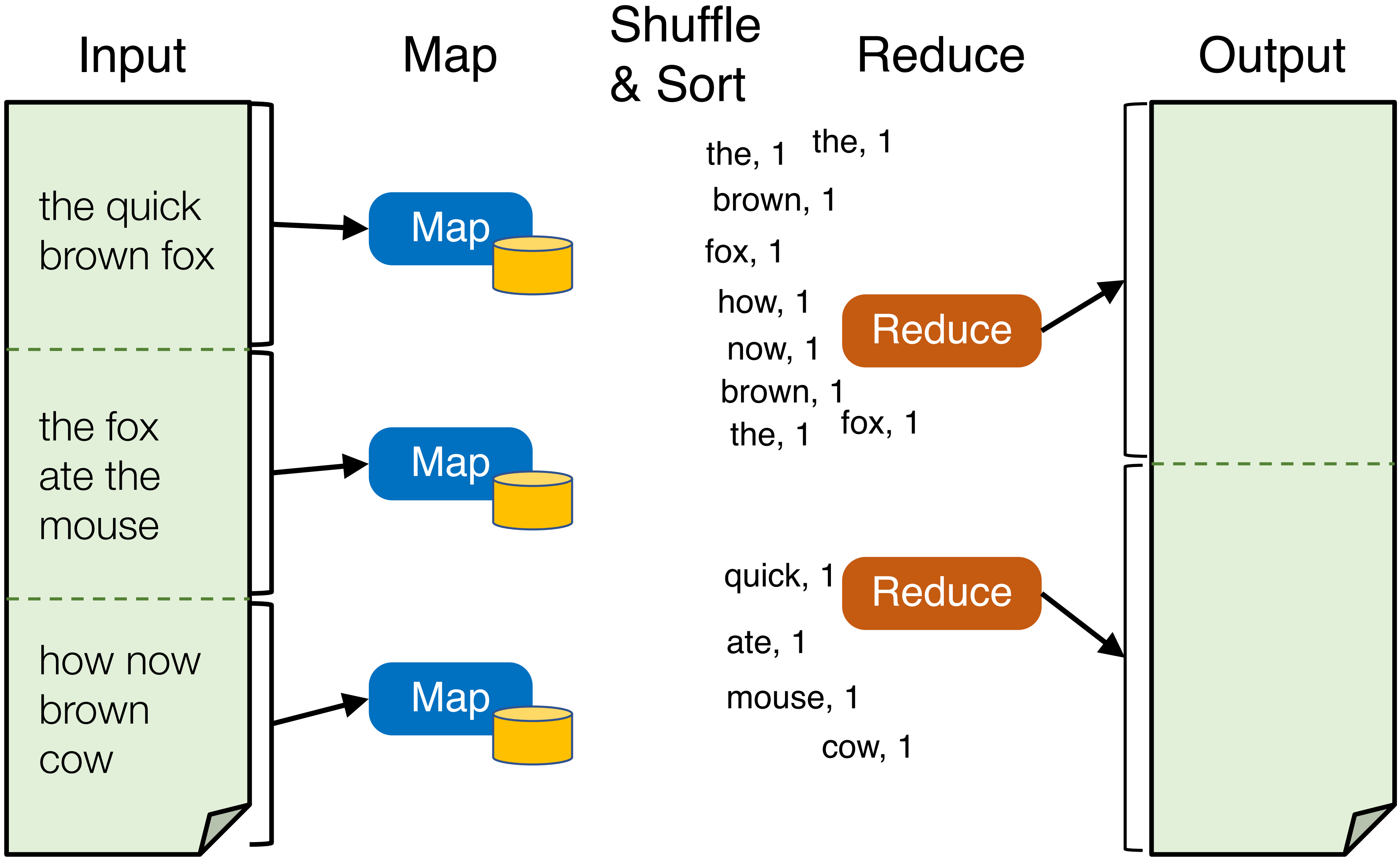
MapReduce: Word Count

```
map(key, value):
    for each word w in value:
        EmitIntermediate(w, “1”);

reduce(key, values):
    int result = 0;
    for each v in values:
        results += ParseInt(v);
    Emit(AsString(result));
```

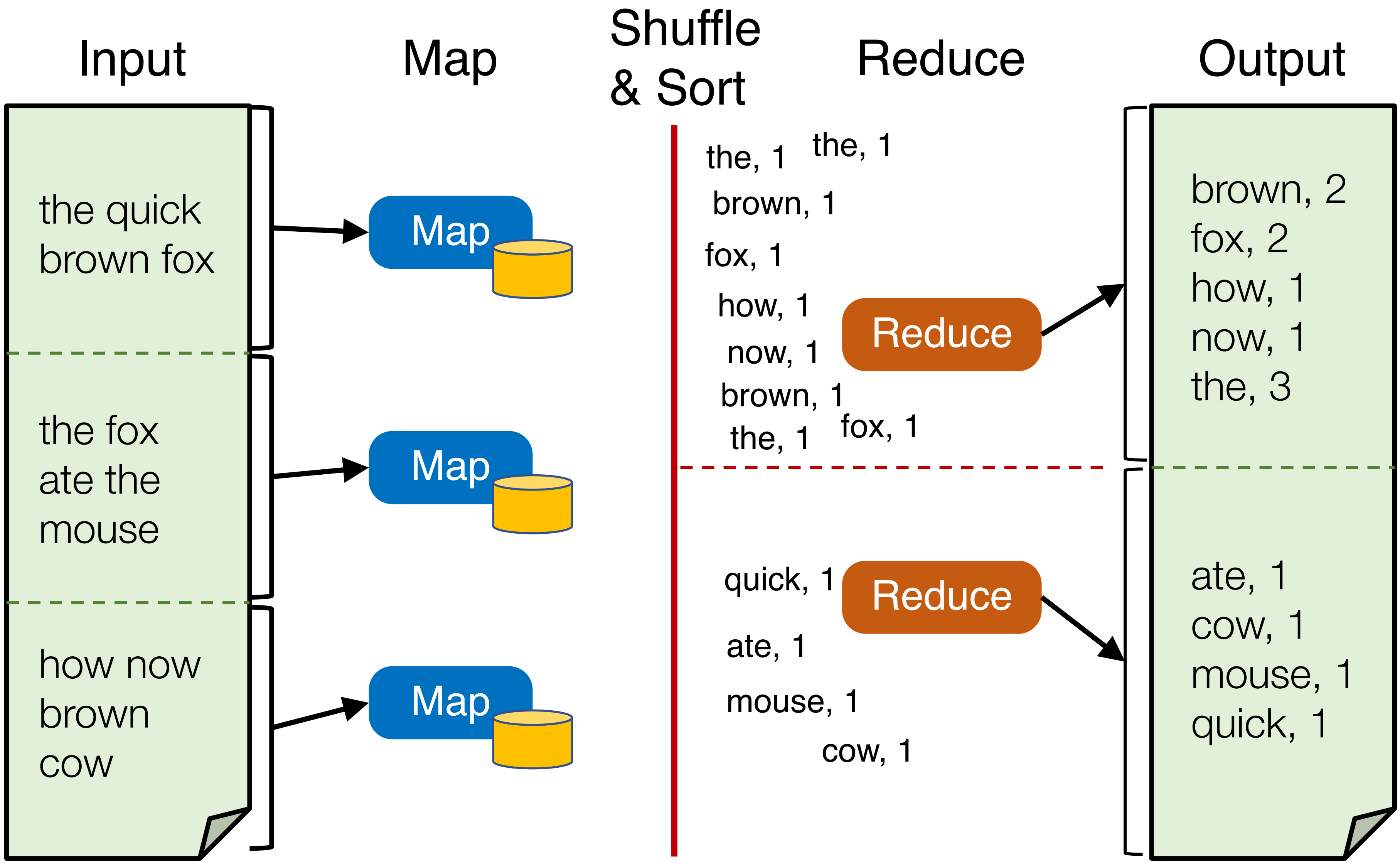
Word Count execution



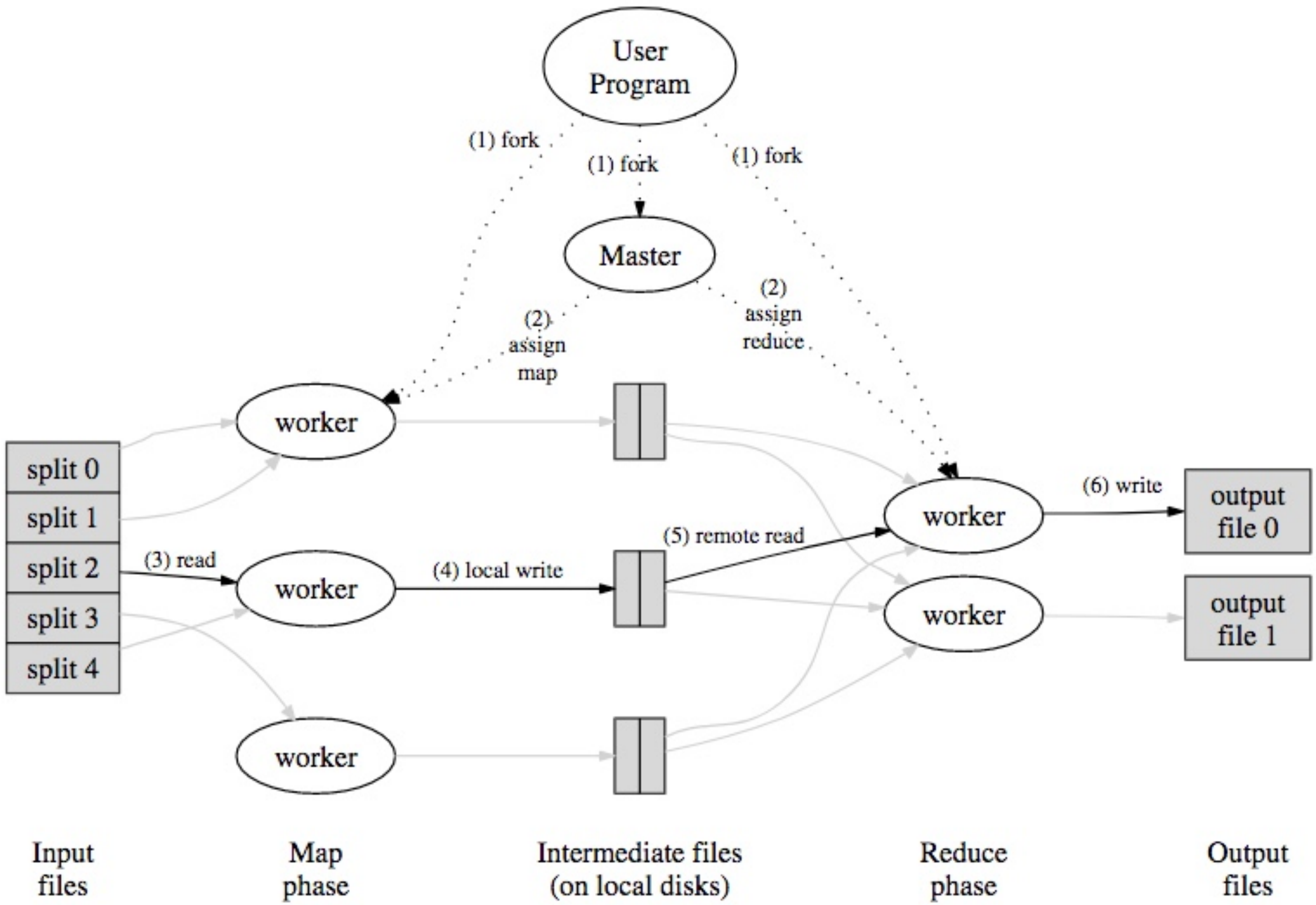
Word Count execution



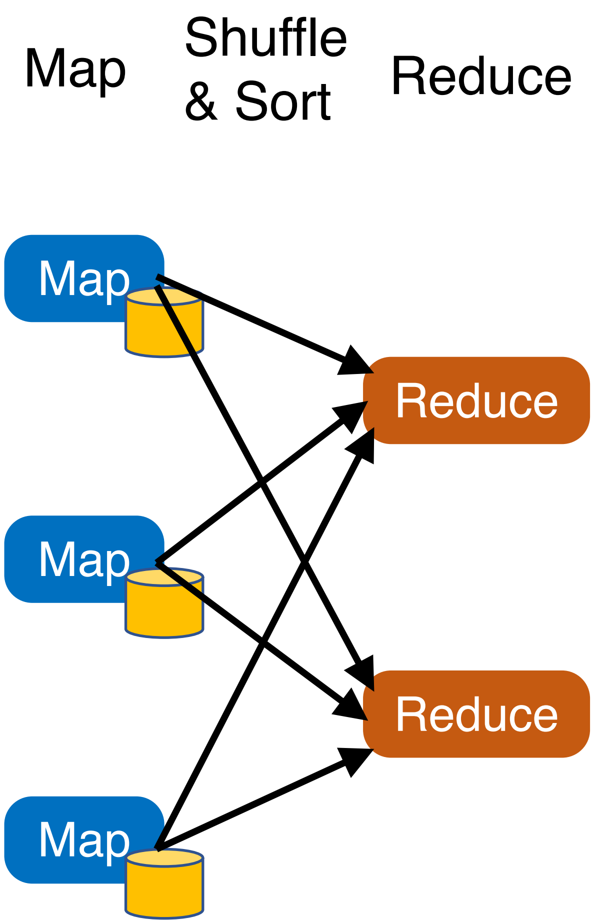
Word Count execution



MapReduce data flows



MapReduce processes



- Map workers write intermediate output to local disk, separated by partitioning. Once completed, tell master node
- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data
- Note:
  - "All-to-all" shuffle b/w mappers and reducers
  - Written to disk ("materialized") b/w each state

MapReduce assumptions

- Commodity hardware
  - Economies of scale!
  - Commodity networking with less bisection bandwidth
  - Commodity storage (hard disks) is cheap
- Failures are common
- Replicated, distributed file system for data storage

Fault tolerance

- If a task crashes:
  - Retry on another node
    - Why this is okay?
  - If the same task repeatedly fails, end the job

Fault tolerance

- If a task crashes:
  - Retry on another node
    - Why this is okay?
  - If the same task repeatedly fails, end the job

- If a node crashes:
  - Relaunch its current tasks on another node
    - What about task inputs?

Today's outline

1. Google MapReduce
    - Google File System

2. Concurrency in Go

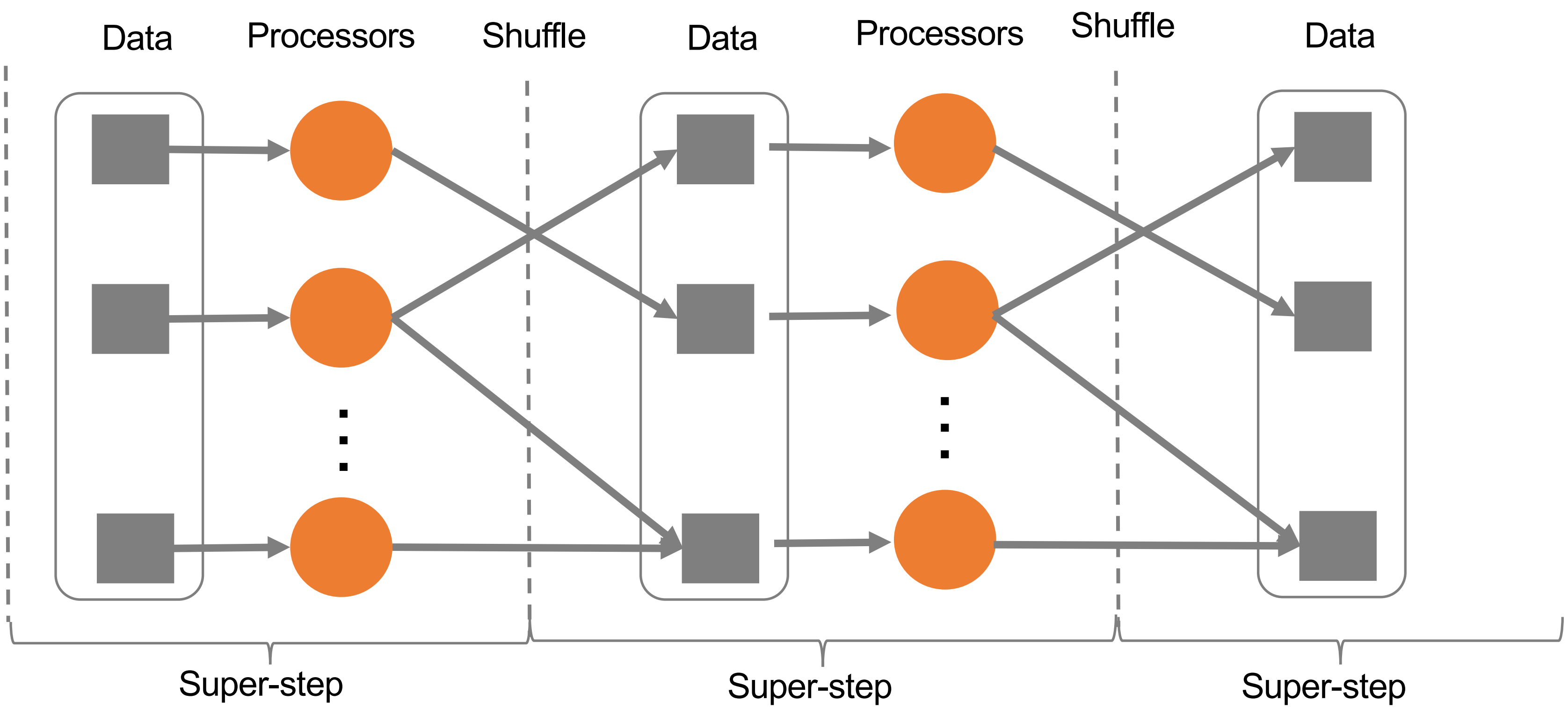
Google file system (GFS)

- Goal: a global (distributed) file system that stores data across many machines
  - Need to handle 100's TBs
- Google published details in 2003
- Open source implementation:
  - Hadoop Distributed File System (HDFS)

Workload-driven design

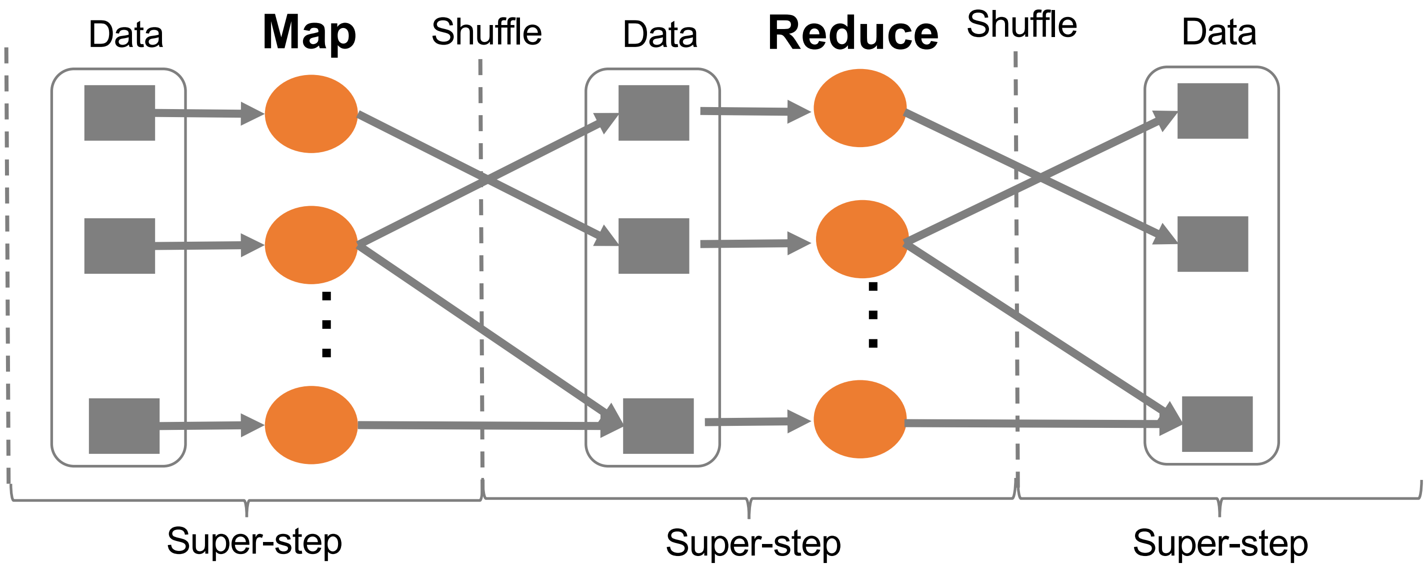
- MapReduce workload characteristics
  - Huge files (GBs)
  - Almost all writes are appends
  - Concurrent appends common
  - High throughput is valuable
  - Low latency is not

Example workloads: Bulk Synchronous Processing (BSP)



*Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990

MapReduce as a BSP system



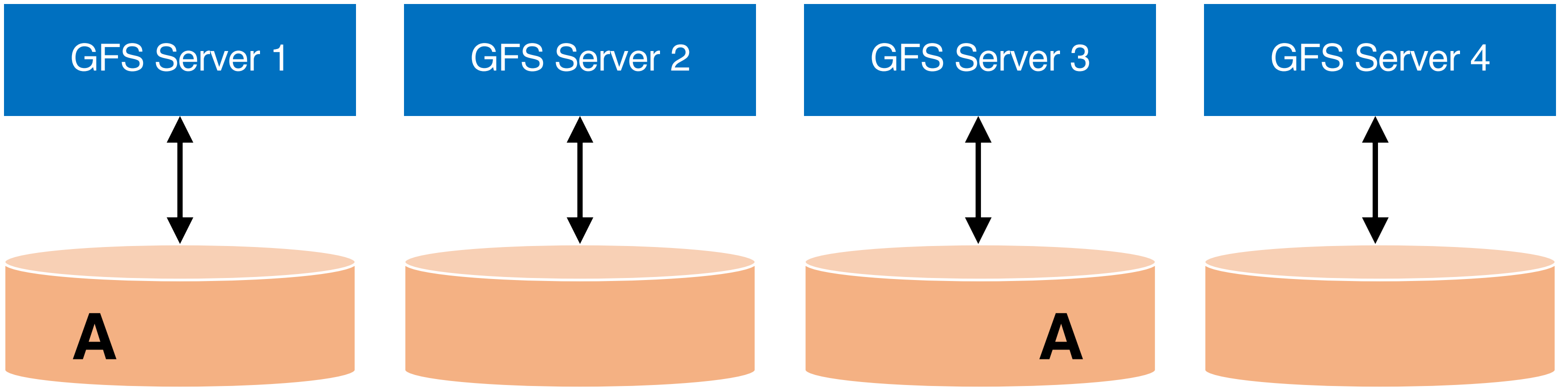
- Read entire dataset, do computation over it
  - Batch processing

- Producer/consumer: many producers append work to file concurrently; one consumer reads and does work

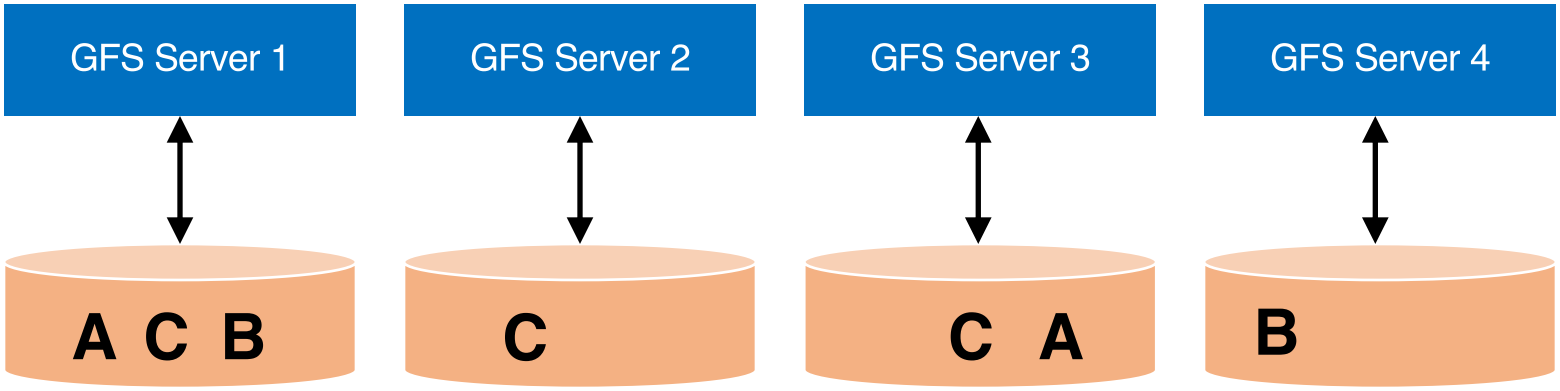
Workload-driven design

- Build a global (distributed) file system that incorporates all these application properties
- Only supports **features required by applications**
- Avoid difficult local file system features, e.g.:
  - rename dir
  - links

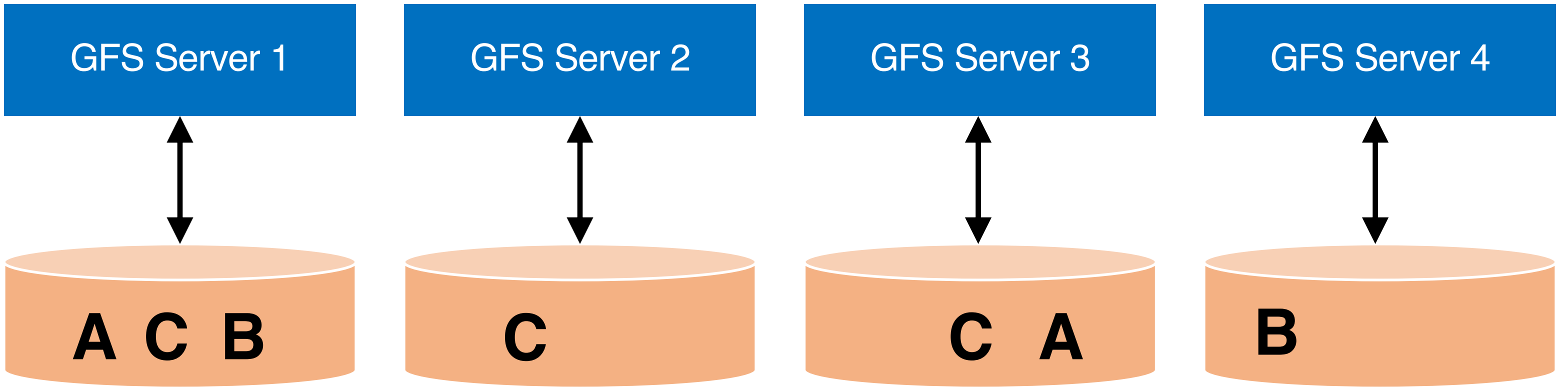
Replication



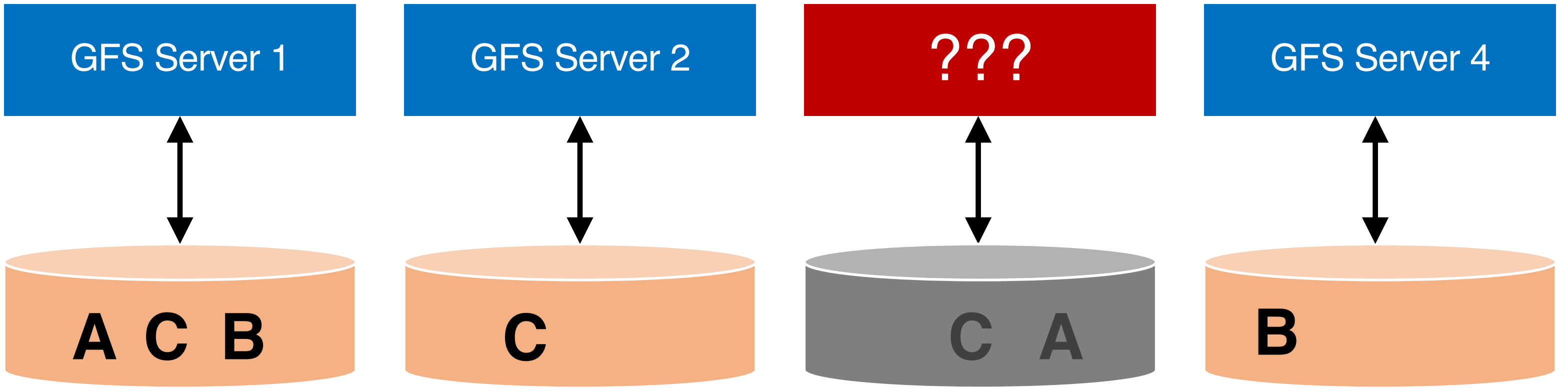
Replication



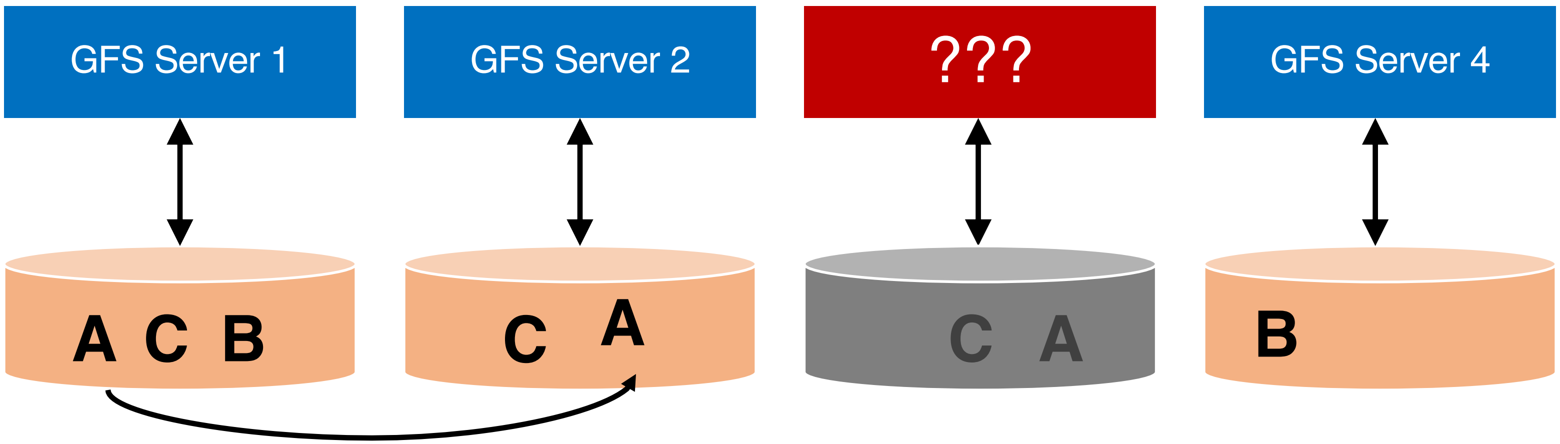
Resilience against failures



Resilience against failures

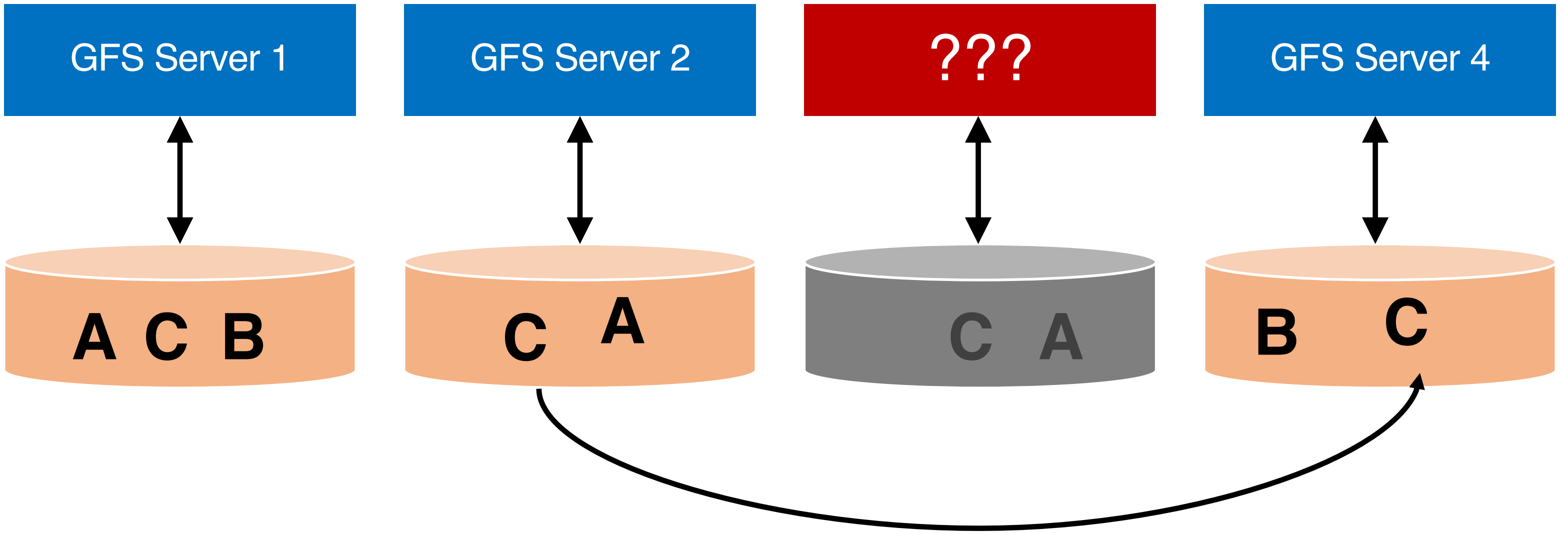


Data Recovery



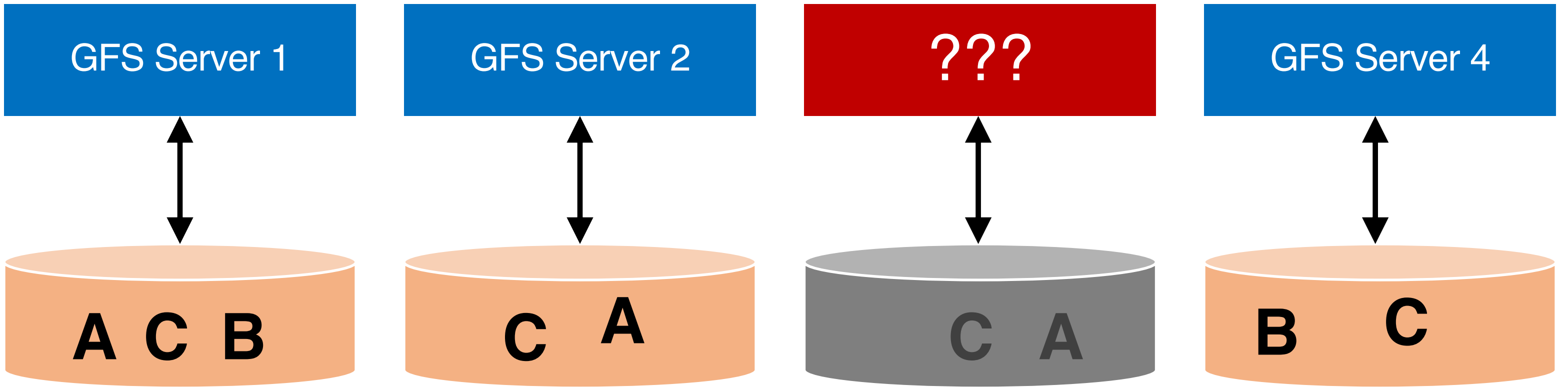
Replicating A to maintain a replication factor of 2

Data Recovery



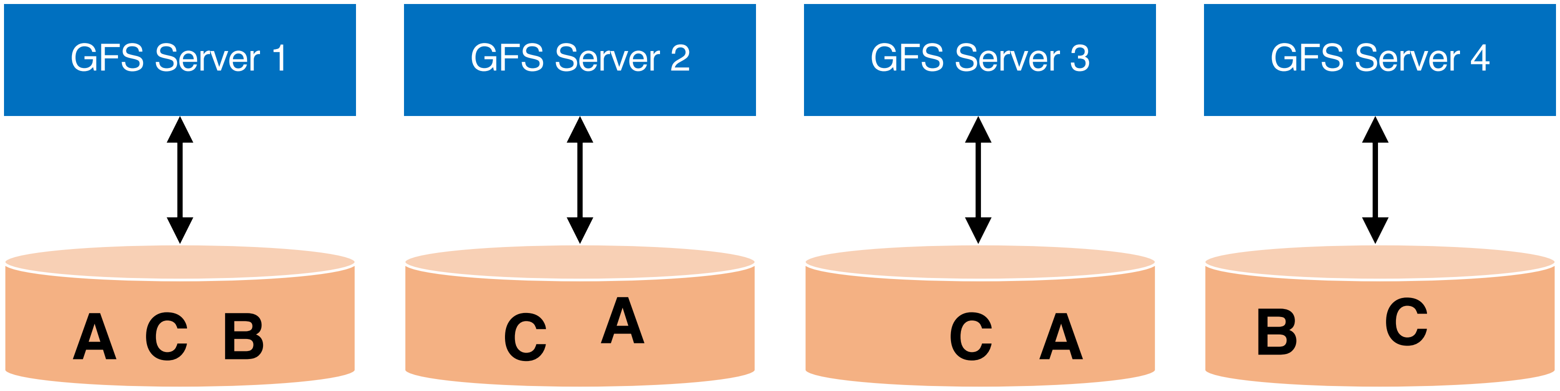
Replicating C to maintain a replication factor of 3

Data Recovery



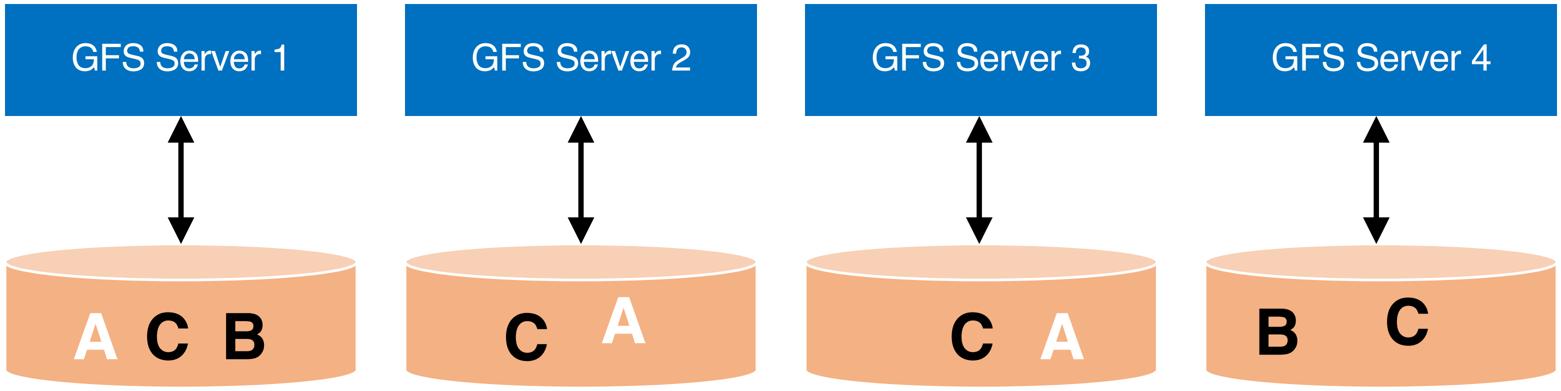
Machine may be dead forever, or it may come back

Data Recovery

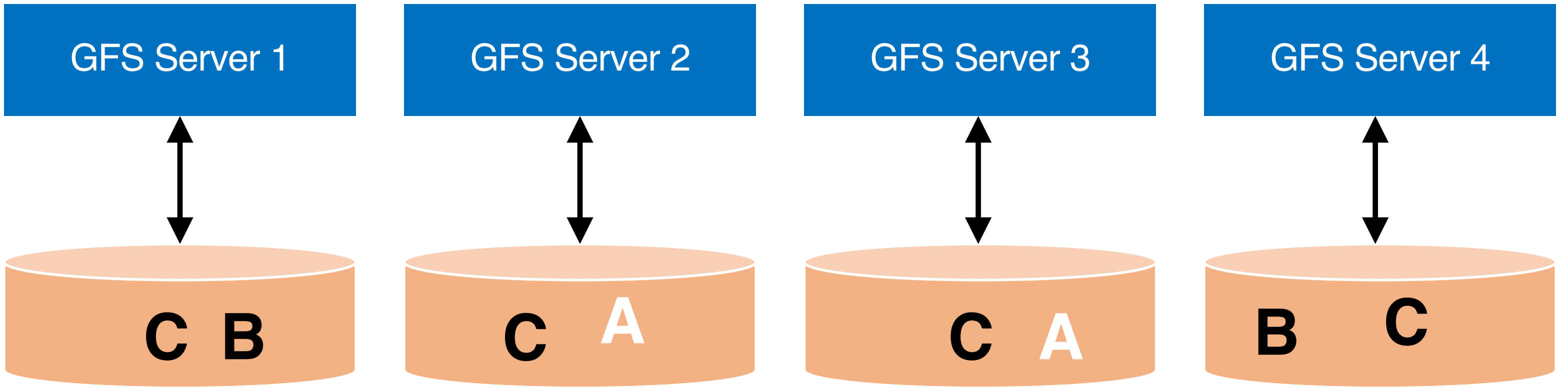


Machine may be dead forever, or it may come back

Data Recovery



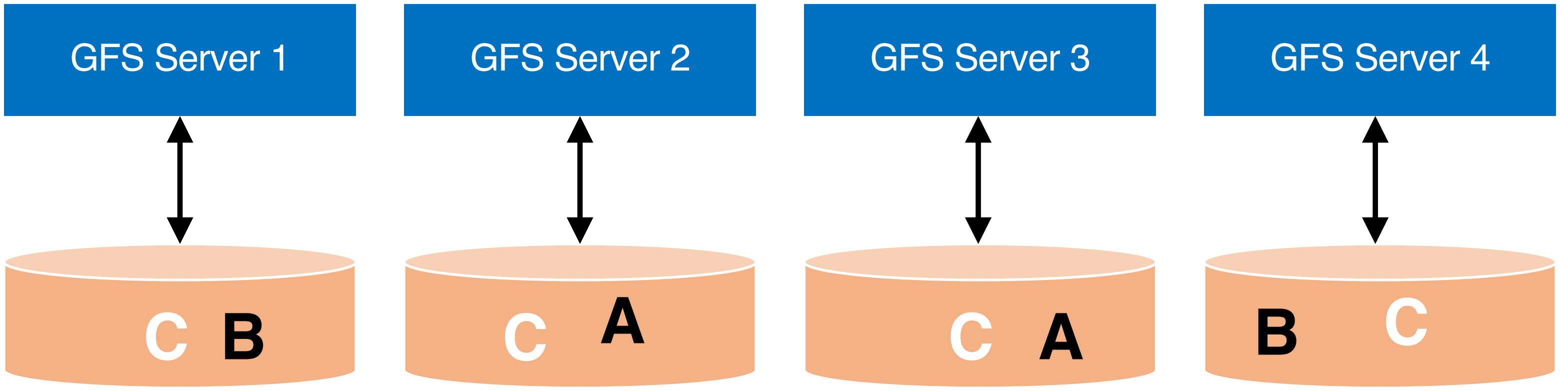
Data Recovery



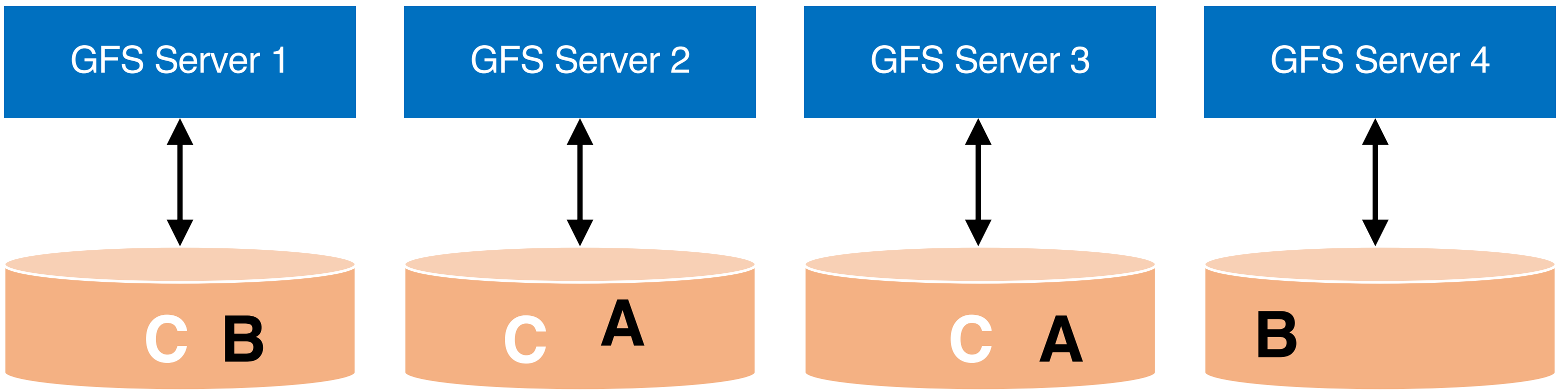
Data Rebalancing

Deleting one A to maintain a replication factor of 2

Data Recovery



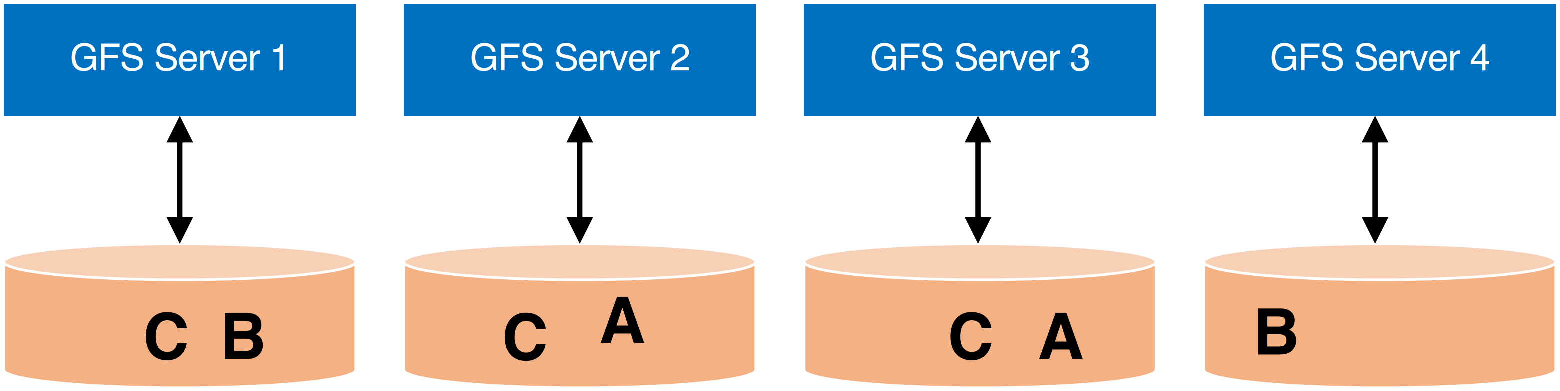
Data Recovery



Data Rebalancing

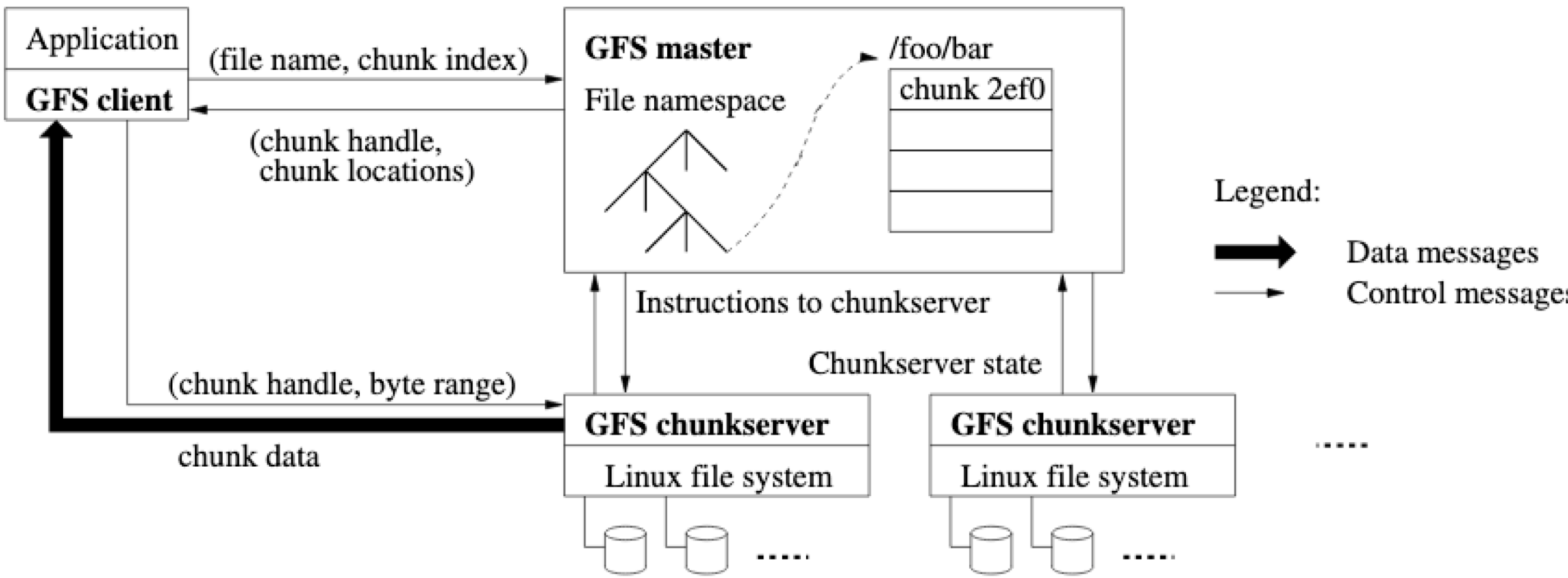
Deleting one C to maintain a replication factor of 3

Data Recovery

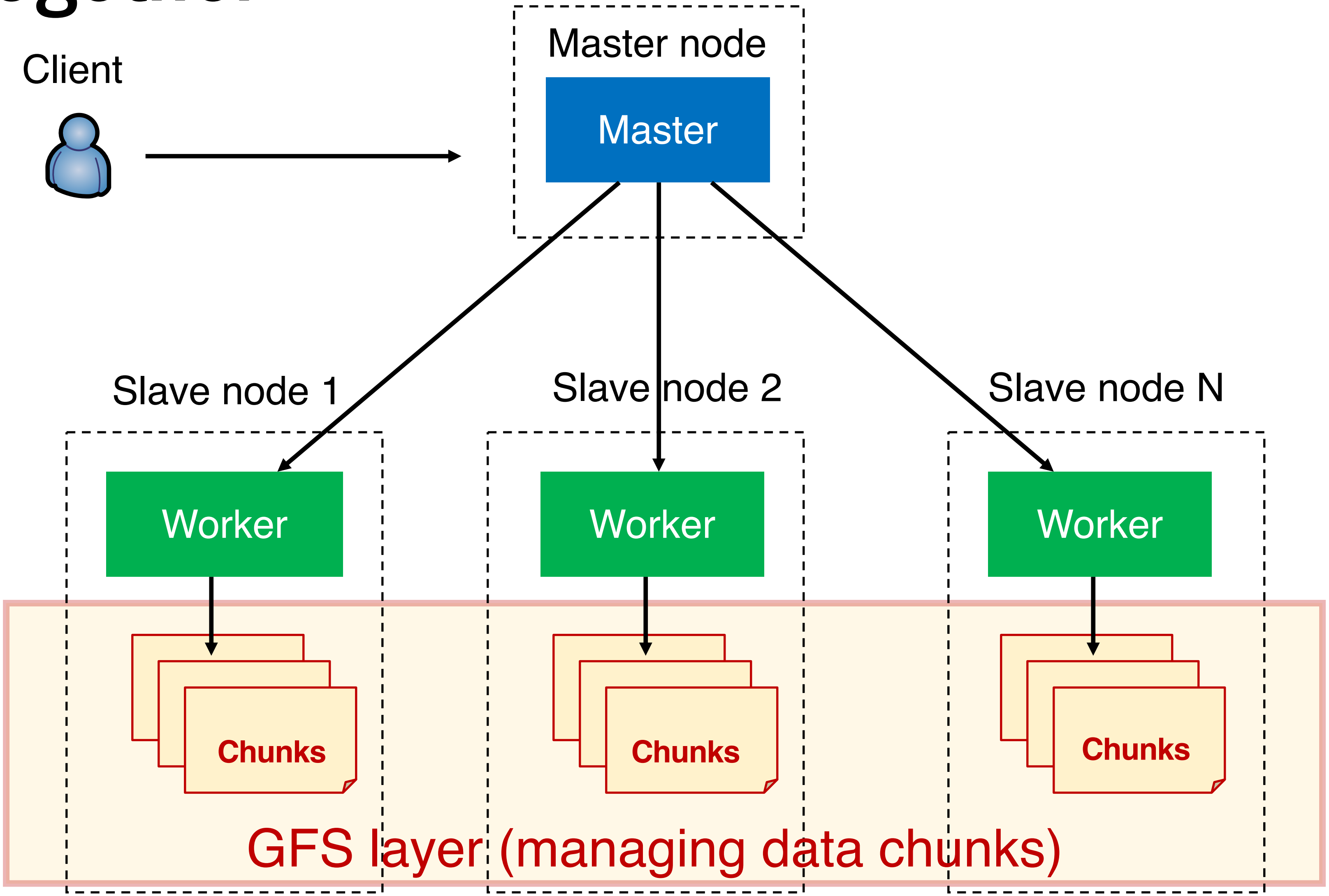


Question: how to maintain a global view of all data distributed across machines?

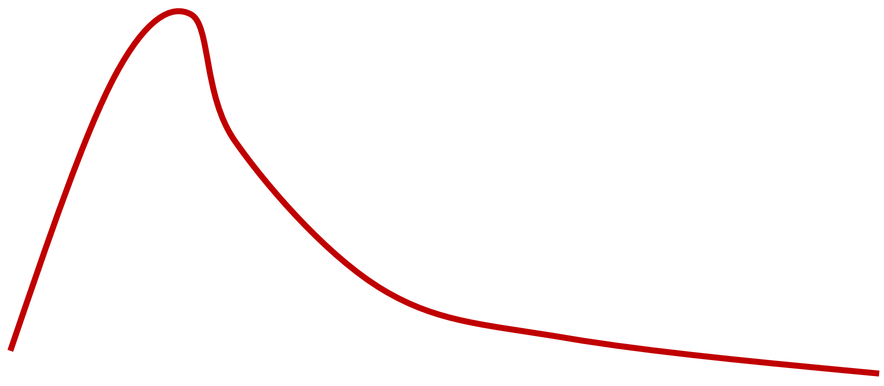
GFS architecture



MapReduce+GFS: Put everything together

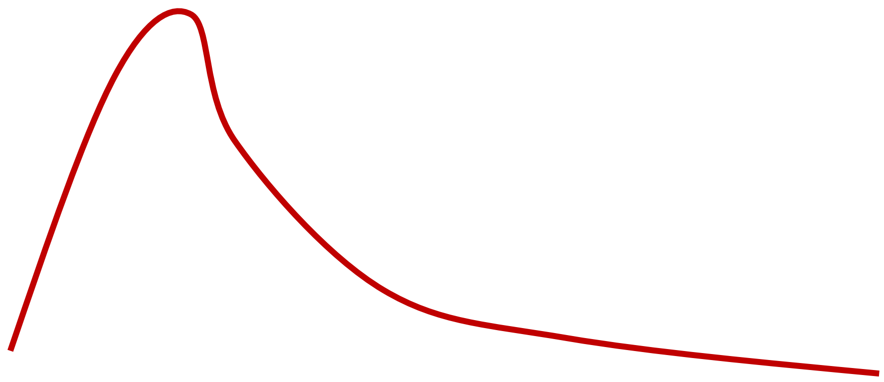


Stragglers



Map task completion time distribution

Stragglers



Map task completion time distribution

- Tail latency means some workers (always) finish late

- Q: How can MR work around this?
  - Hint: its approach to **fault-tolerance** provides the right tool

Resilience against stragglers

- If a task is going slowly (i.e., straggler):
  - Launch second copy of task on another node
  - Take the output of whichever finishes first

More design

- Master failure
- Locality
- Task granularity

GFS usage at Google

- 200+ clusters
- Many clusters of 1000s of machines
- Pools of 1000s of clients
- 4+ PB filesystems
- 40 GB/s read/write load
    - In the presence of frequent hardware failures

* Jeff Dean, LADIS 2009

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

MapReduce discussion

https://piazza.com/class/k5shuiyl7ur79q?cid=14

MapReduce discussion

- What will likely serve as a performance bottleneck for Google's MapReduce used back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

MapReduce discussion

- What will likely serve as a performance bottleneck for Google's MapReduce used back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

- How does MapReduce reduce the effect of slow network?

MapReduce discussion

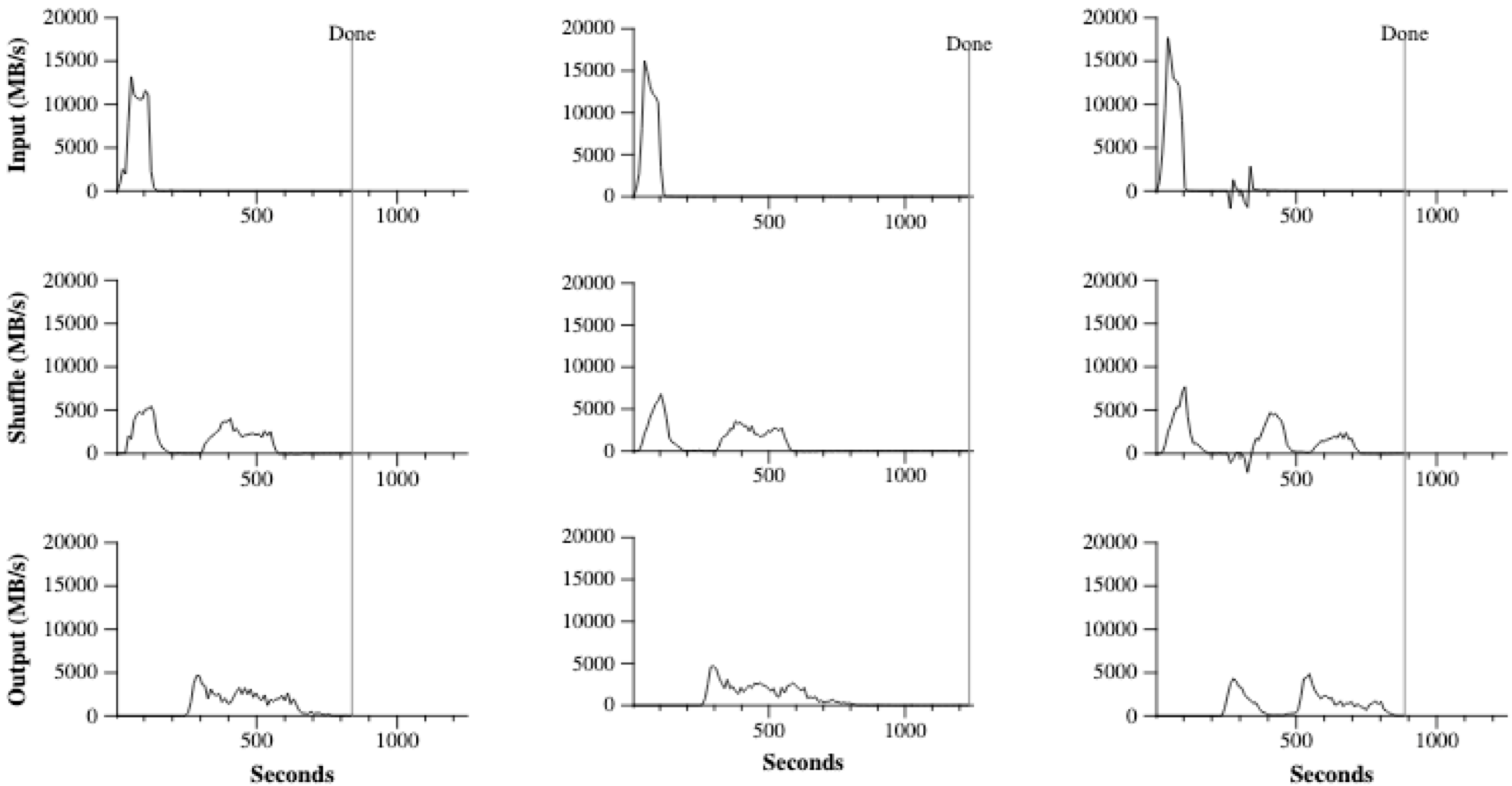
- How does MapReduce jobs get good load balance across worker machines?

MapReduce discussion

- Consider the indexing pipeline where you start with HTML documents. You want to index the documents after removing the most commonly occurring words:
    1. Compute the most common words;
    2. Remove them and build the index

What are the main shortcomings of using MapReduce to support such pipeline-like applications?

MapReduce discussion



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

Today's outline

1. Google MapReduce

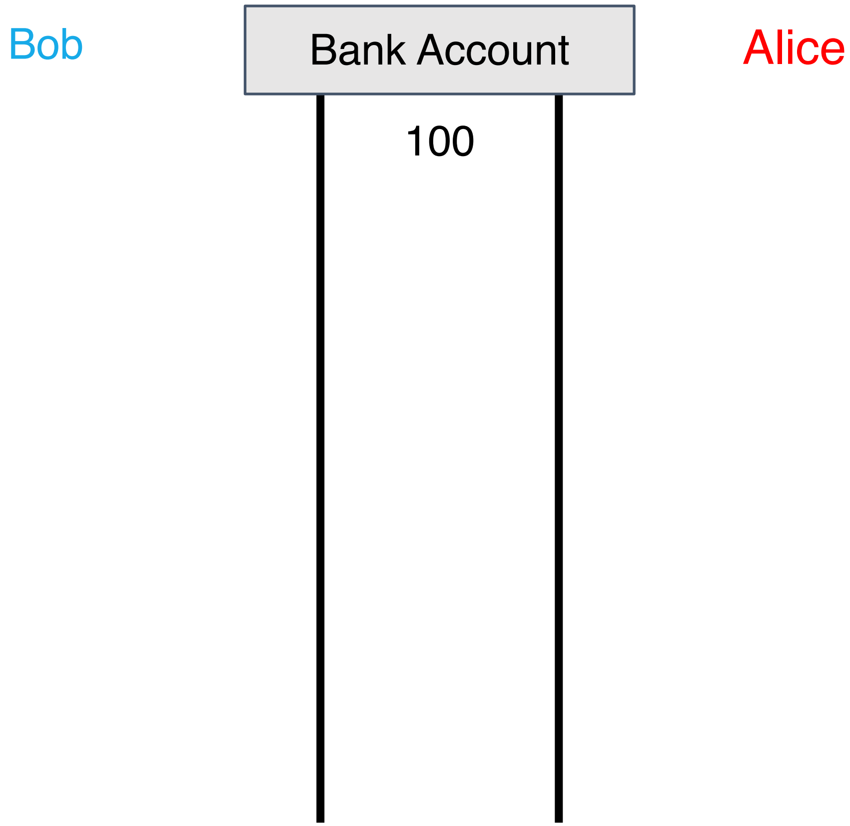
2. Concurrency in Go
    - Two synchronization mechanisms
        - Locks
        - Channels

Two synchronization mechanisms in Go

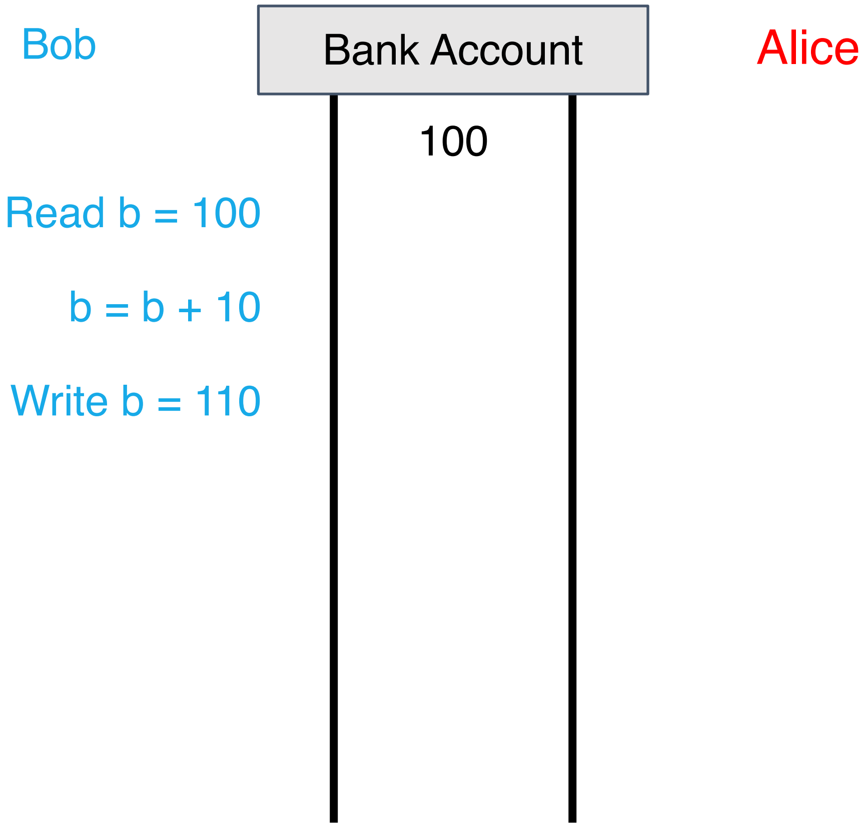
- **Locks:** limit access to a critical section
  - Access to a critical section (e.g., shared variables) must be mutually exclusive

- **Channels:** pass information across threads using a queue

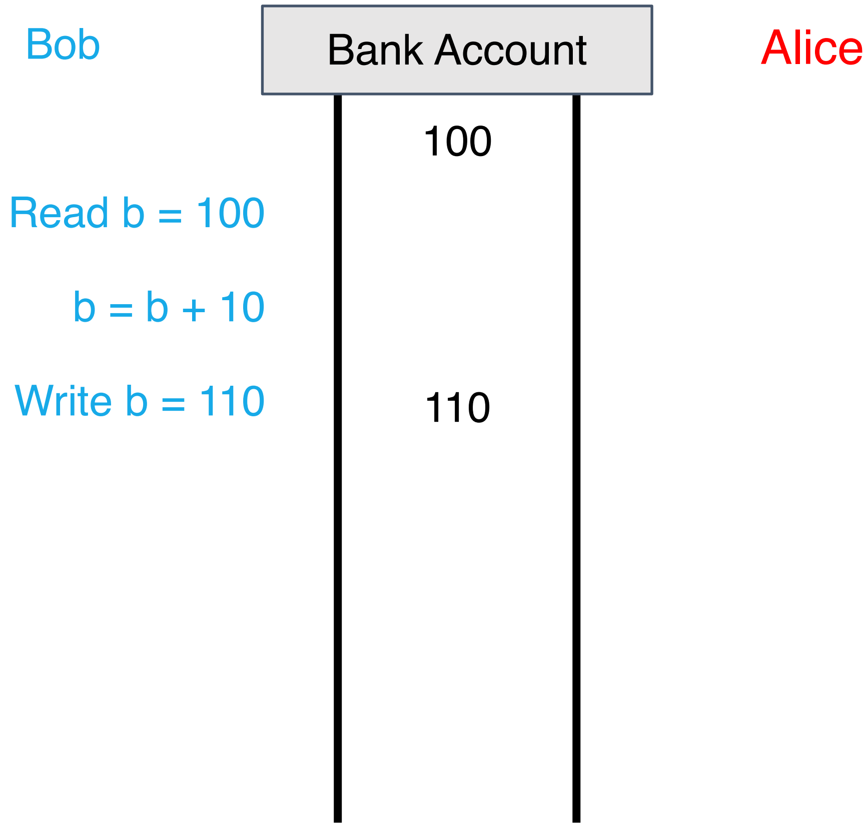
Example: Bank account



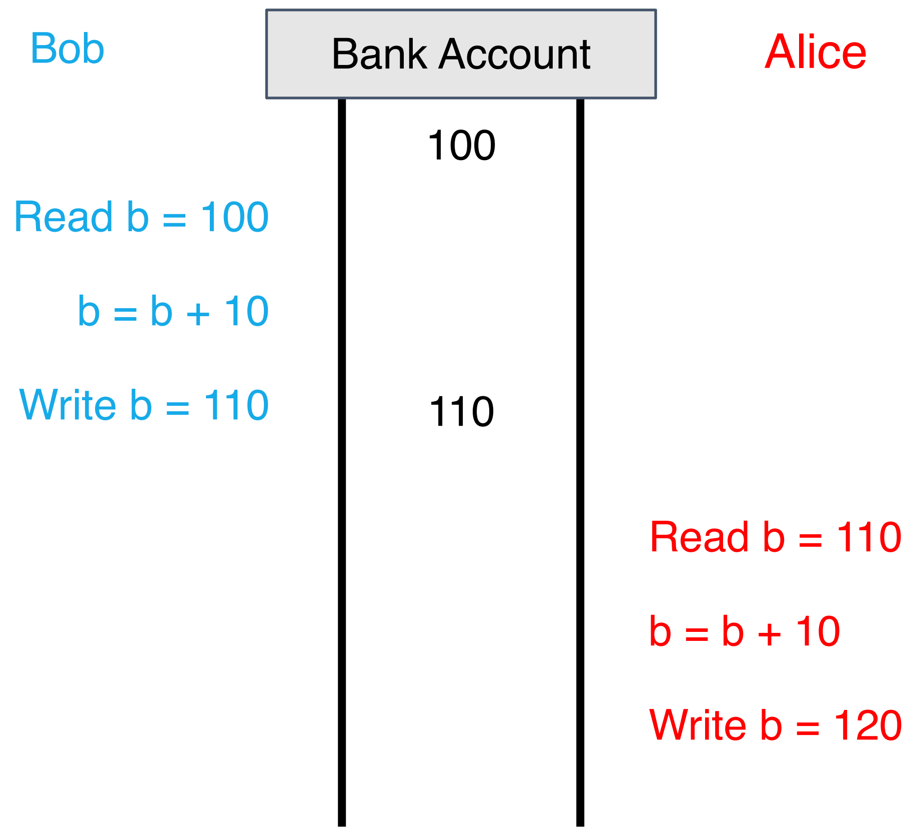
Example: Bank account



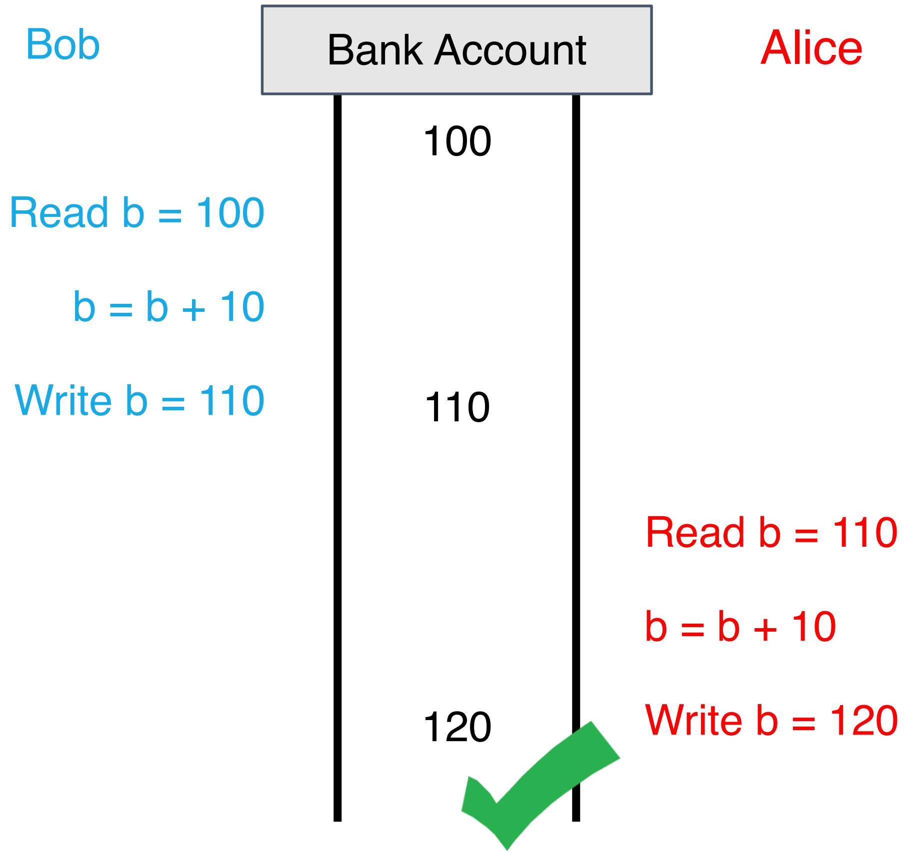
Example: Bank account



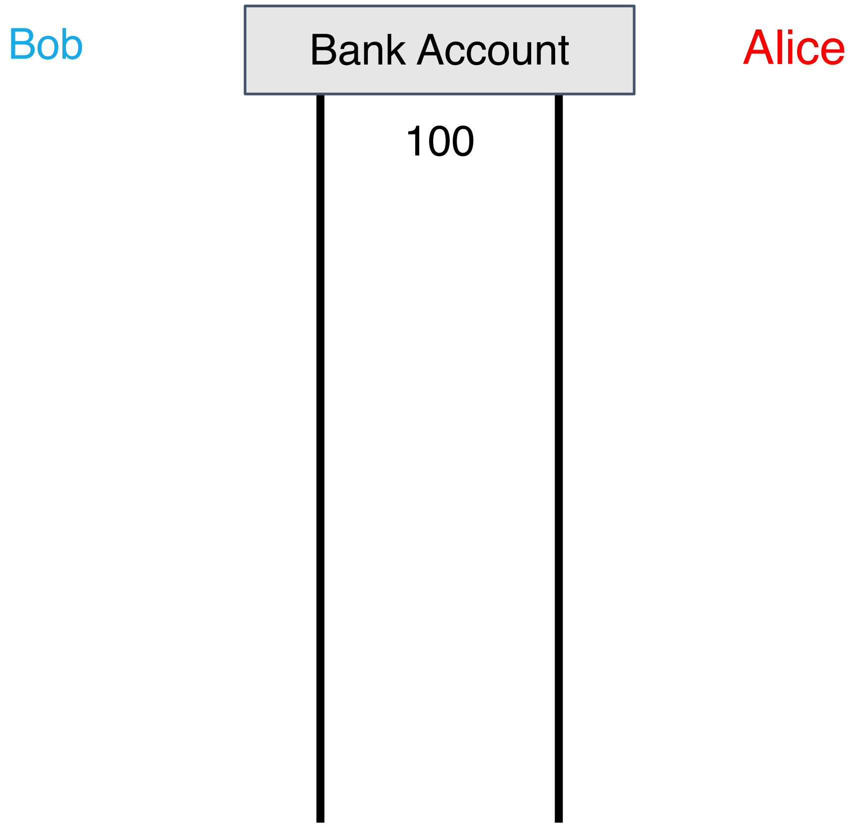
Example: Bank account



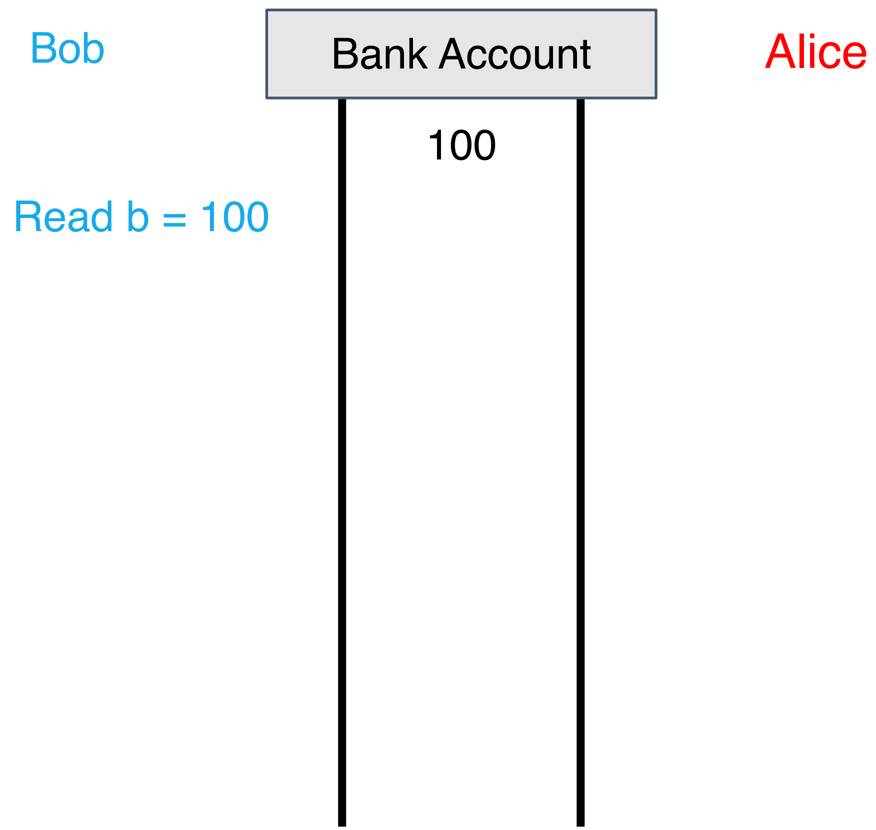
Example: Bank account



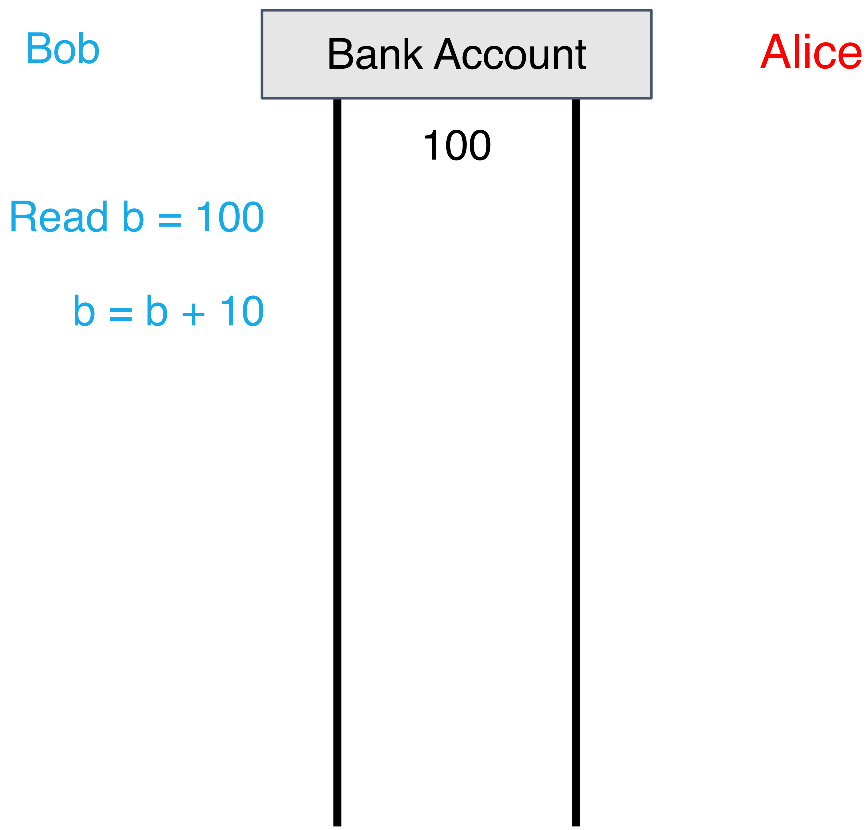
Example: Bank account



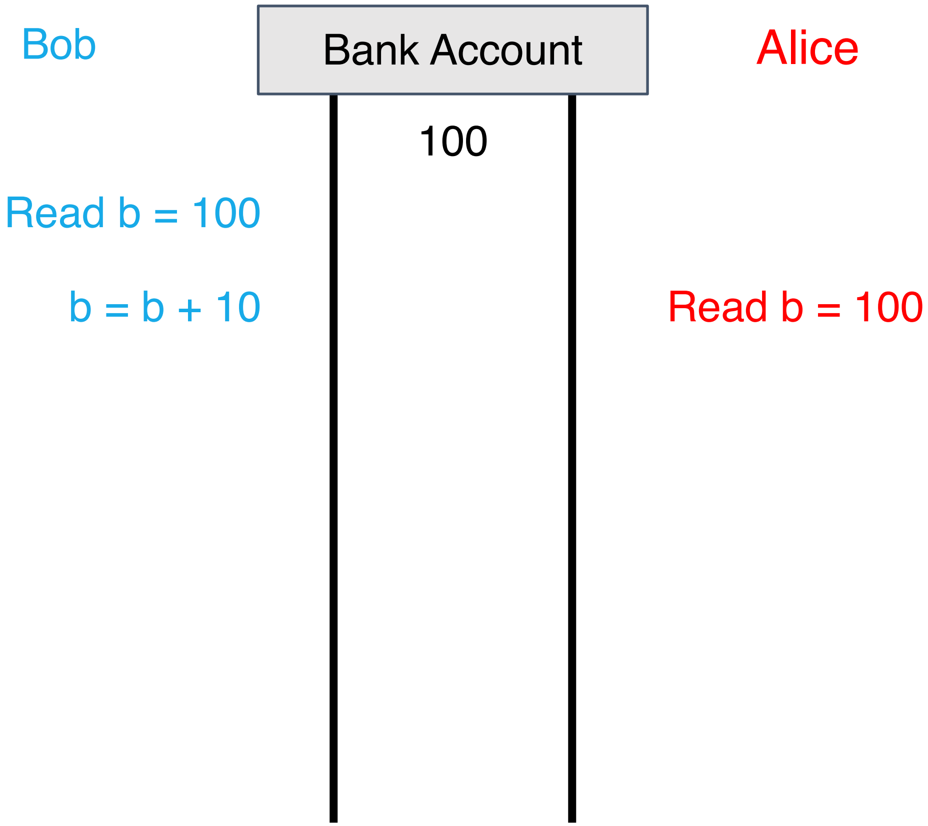
Example: Bank account



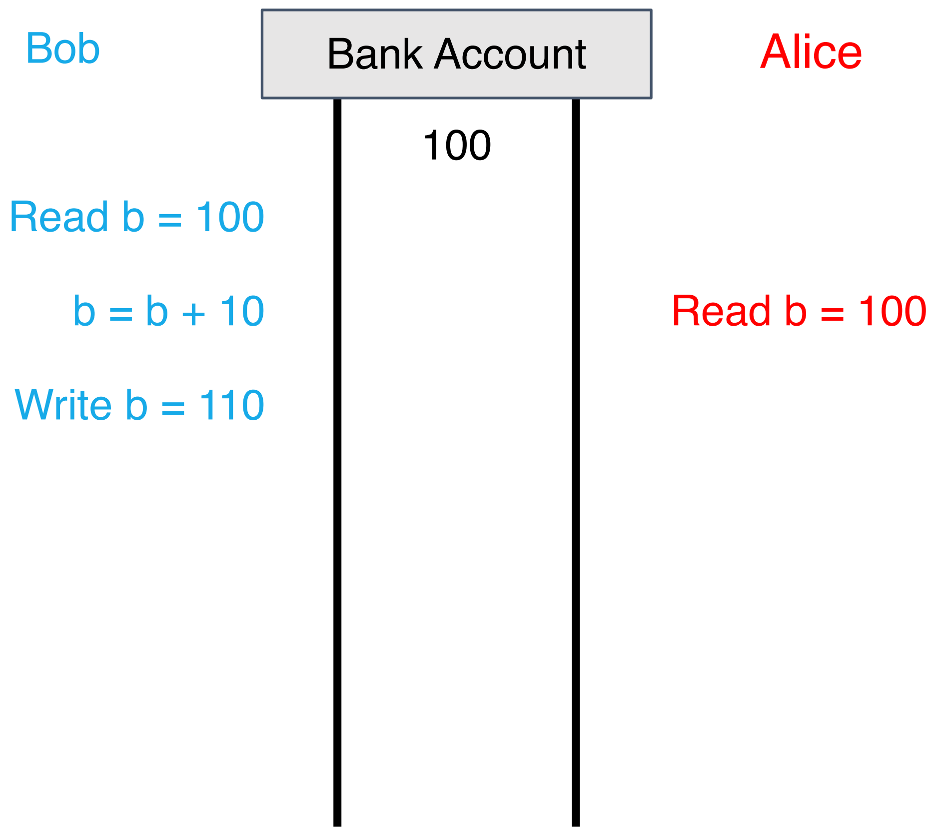
Example: Bank account



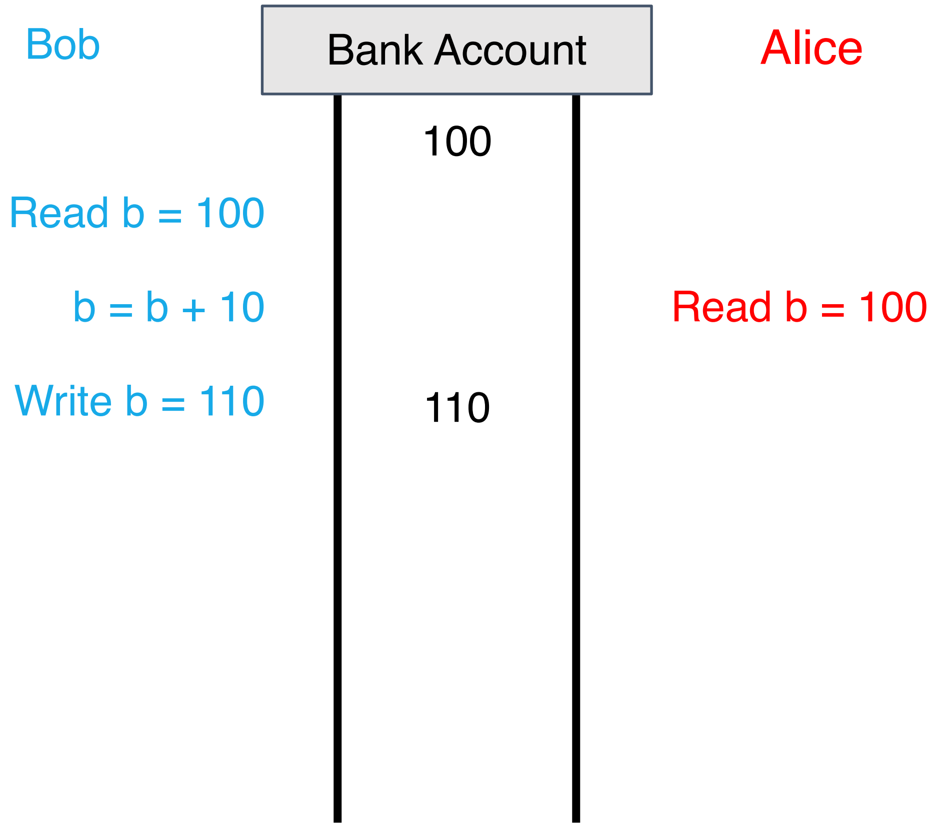
Example: Bank account



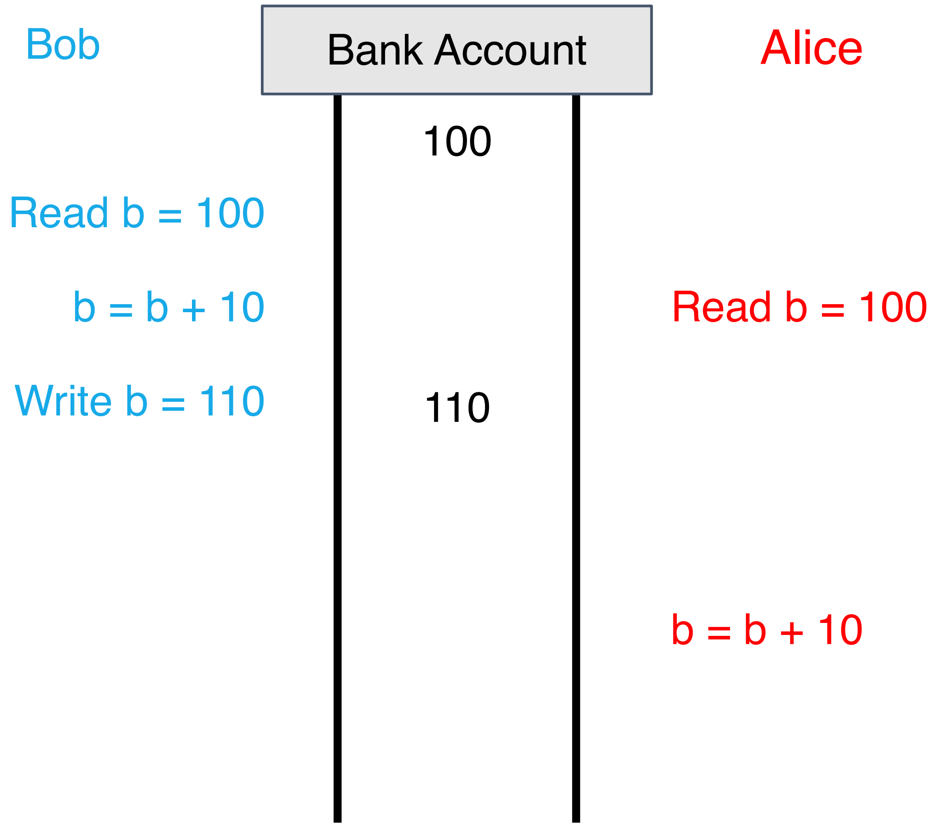
Example: Bank account



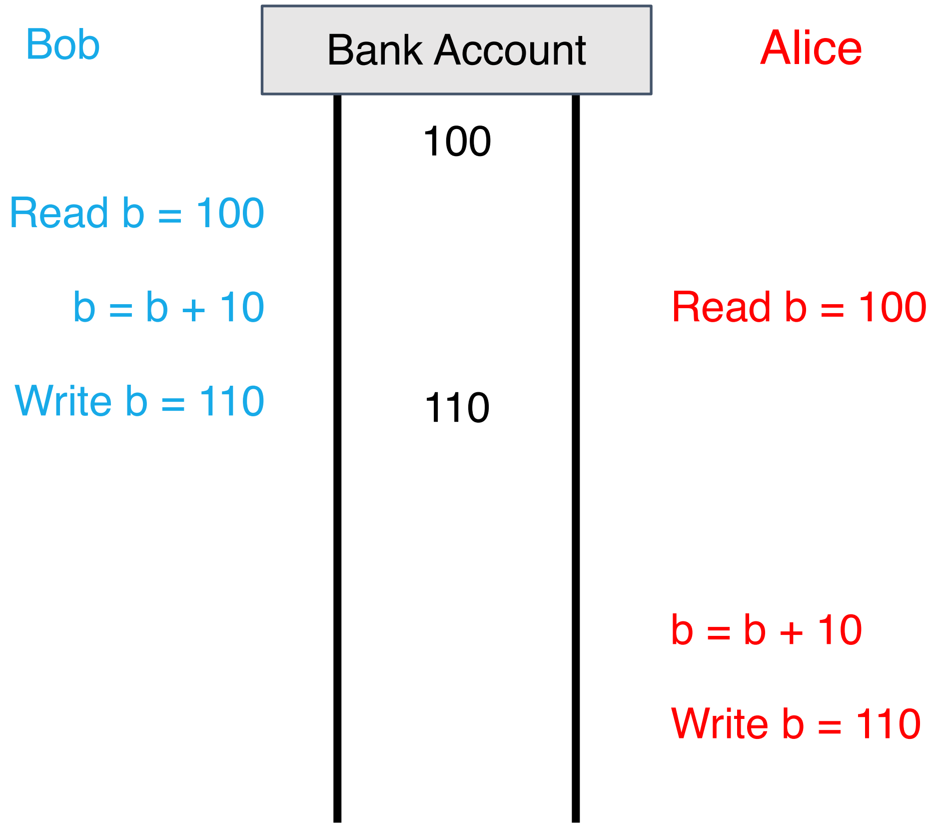
Example: Bank account



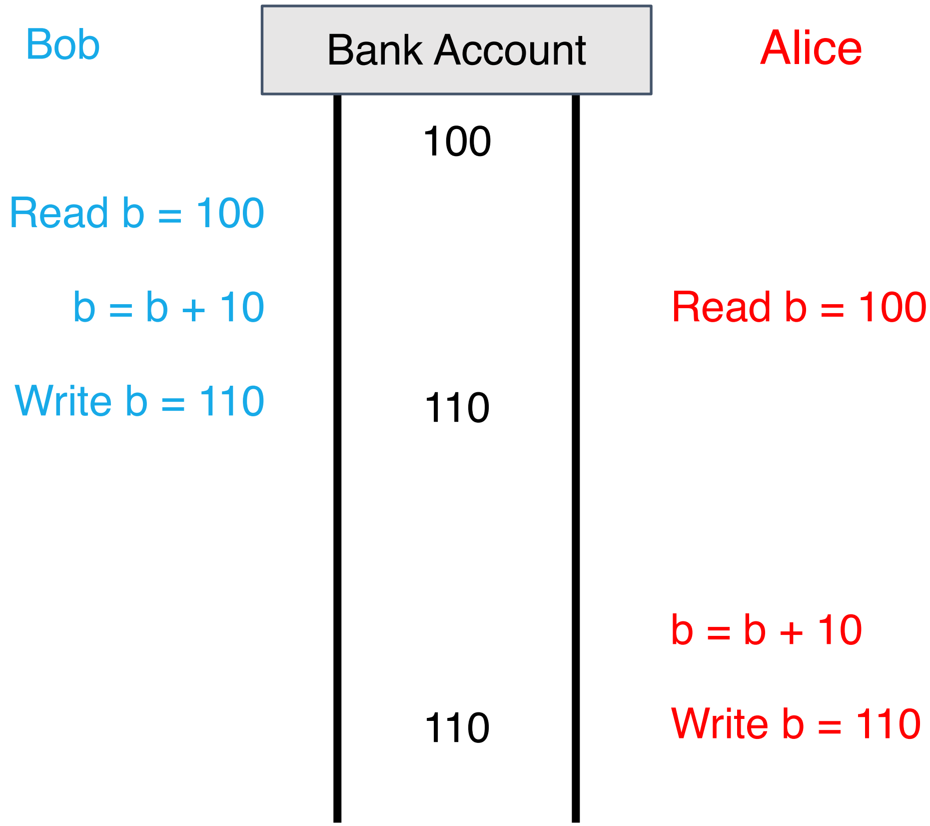
Example: Bank account



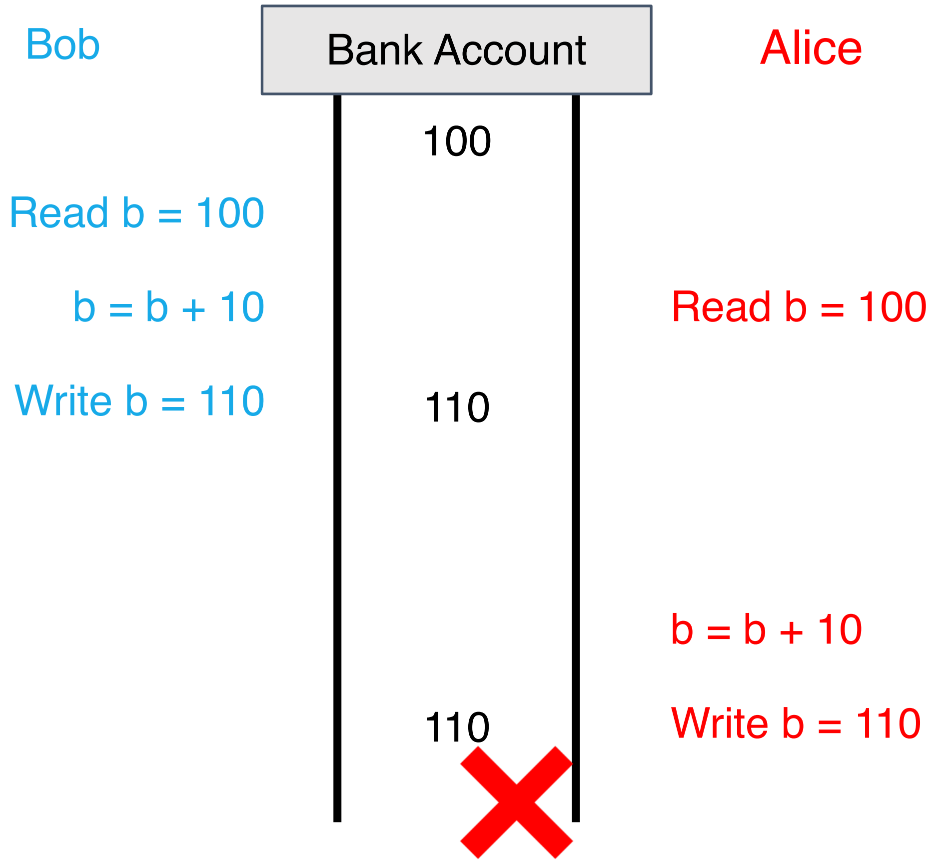
Example: Bank account



Example: Bank account



Example: Bank account



What went wrong?

- Changes to `balance` are not *atomic*

What went wrong?

- Changes to `balance` are not *atomic*

```
func Deposit(amount) {
        lock balanceLock
        read balance
        balance = balance + amount
        write balance
        unlock balanceLock
}
```

What went wrong?

- Changes to `balance` are not *atomic*

```
func Deposit(amount) {
      lock balanceLock
      read balance
      balance = balance + amount
      write balance
      unlock balanceLock
}
```

Critical section

Mutex locks in Go

```go
package account

import "sync"

type Account struct {
	balance int
	lock sync.Mutex
}
```

Mutex locks in Go

```go
package account

import "sync"

type Account struct {
        balance int
        lock sync.Mutex
}


func NewAccount(init int) Account
        return Account{balance: init}
}
```

Mutex locks in Go

```go
package account

import "sync"

type Account struct {
        balance int
        lock sync.Mutex
}


func NewAccount(init int) Account
        return Account{balance: init}
}
```

```go
func (a *Account) CheckBalance() int {
      a.lock.Lock()
      defer a.lock.Unlock()
      return a.balance
}
```

Mutex locks in Go

```go
package account

import "sync"

type Account struct {
        balance int
        lock sync.Mutex
}


func NewAccount(init int) Account
        return Account{balance: init}
}
```

```go
func (a *Account) CheckBalance() int {
     a.lock.Lock()
     defer a.lock.Unlock()
     return a.balance
}


func (a *Account) Withdraw(v int) {
     a.lock.Lock()
     defer a.lock.Unlock()
     a.balance -= v
}


func (a *Account) Deposit(v int) {
     a.lock.Lock()
     defer a.lock.Unlock()
     a.balance += v
}
```

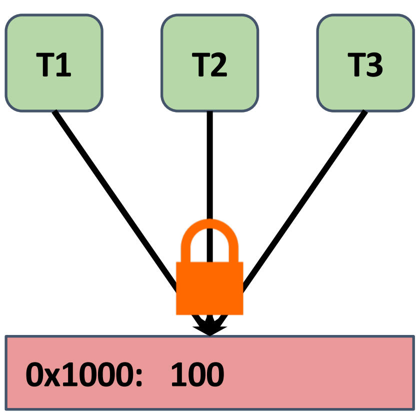
Read write locks in Go

```go
package account

import "sync"

type Account struct {
	balance int
	lock sync.RWMutex
}


func NewAccount(init int) Account
	return Account{balance: init}
}
```

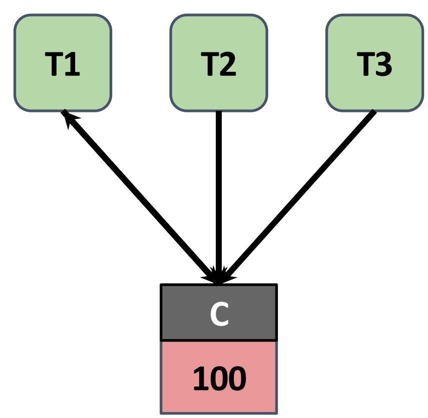
```go
func (a *Account) CheckBalance() int {
	a.lock.RLock()
	defer a.lock.RUnlock()
	return a.balance
}


func (a *Account) Withdraw(v int) {
	a.lock.Lock()
	defer a.lock.Unlock()
	a.balance -= v
}


func (a *Account) Deposit(v int) {
	a.lock.Lock()
	defer a.lock.Unlock()
	a.balance += v
}
```

Two solutions to the same problem

- Locks:
  - Multiple threads can reference same memory location
  - Use lock to ensure only one thread is updating it at any time



- Channels:
  - Data item initially stored in channel
  - Threads must request item from channel, make updates, and return item to channel



Go channels

```
// Launch workers
for i := 0; i < numWorkers; i++ {
        go func() {
                // ... do some work

        }()
}
```

- In Go, *channels* and *goroutines* are more idiomatic than locks

Go channels

```
result := make(chan int, numWorkers)

// Launch workers
for i := 0; i < numWorkers; i++ {
        go func() {
                // ... do some work
                result <- i
        }()
}
```

- In Go, ***channels*** and ***goroutines*** are more idiomatic than locks

Go channels

```go
result := make(chan int, numWorkers)

// Launch workers
for i := 0; i < numWorkers; i++ {
        go func() {
                // ... do some work
                result <- i
        }()
}


// Wait until all worker threads have finished
for i := 0; i < numWorkers; i++ {
        handleResult(<-result)
}
fmt.Println("Done!")
```

- In Go, *channels* and *goroutines* are more idiomatic than locks

Bank account code (using channels)

```
package account

type Account struct {
	// Fill in Here
}

func NewAccount(init int) Account {
	// Fill in Here
}
```

```
func (a *Account) CheckBalance() int {
	// What goes Here?
}

func (a *Account) Withdraw(v int) {
	// ???
}

func (a *Account) Deposit(v int) {
	// ???
}
```

Bank account code (using channels)

```
package account

type Account struct {
	balance chan int
}

func NewAccount(init int) Account {
	a := Account make(chan int, 1)}
	a.balance <- init
	return a
}
```

```
func (a *Account) CheckBalance() int {
	bal := <-a.balance
	a.balance <- bal
	return bal
}

func (a *Account) Withdraw(v int) {
	bal := <-a.balance
	a.balance <- (bal - v)
}

func (a *Account) Deposit(v int) {
	bal := <-a.balance
	a.balance <- (bal + v)
}
```

`select` statement in Go

- `select` allows a goroutine to wait on multiple channels at once

```
for {
    select {
      case money := <-dad:
            buySnacks(money)
      case money := <-mom:
            buySnacks(money)
      case default:
            starve()
            time.Sleep(5 * time.Second)
    }
}
```

Handle timeouts using `select`

```go
result := make(chan int)
timeout := make(chan bool)

// Asynchronously request an
// answer from server, timing
// out after X seconds
askServer(result, timeout)

// Wait on both channels
select {
	case res := <-result:
		handleResult(res)
	case <-timeout:
		fmt.Println("Timeout!")
}
```

```go
func askServer(
	result chan int,
	timeout chan bool) {

	// Start timer
	go func() {
		time.Sleep(5 * time.Second)
		timeout <- true
	}()

	// Ask server
	go func() {
		response := // ... send RPC
		result <- response
	}()
}
```

Handle timeouts using `select`

```go
result := make(chan int)
timeout := make(chan bool)

// Asynchronously request an
// answer from server, timing
// out after X seconds
askServer(result, timeout)

// Wait on both channels
select {
	case res := <-result:
		handleResult(res)
	case <-timeout:
		fmt.Println("Timeout!")
}
```

```go
func askServer(
	result chan int,
	timeout chan bool) {

	// Start timer
	go func() {
		time.Sleep(5 * time.Second)
		timeout <- true
	}()

	// Ask server
	go func() {
		response := // ... send RPC
		result <- response
	}()
}
```