

Concurrency in Go

September 2026

Go Resources

<https://tour.golang.org/list>

<https://play.golang.org>

<https://gobyexample.com>

https://go.dev/doc/effective_go

<https://go.dev/wiki/All>

Today's Precept...

Two synchronization mechanisms

Locks

Channels

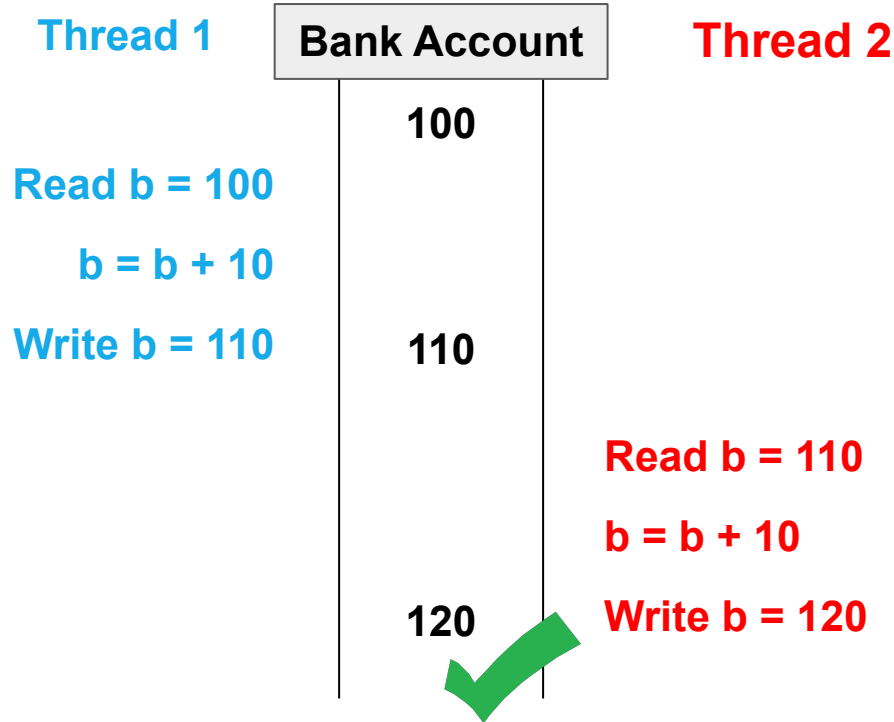
Mapreduce

Two synchronization mechanisms

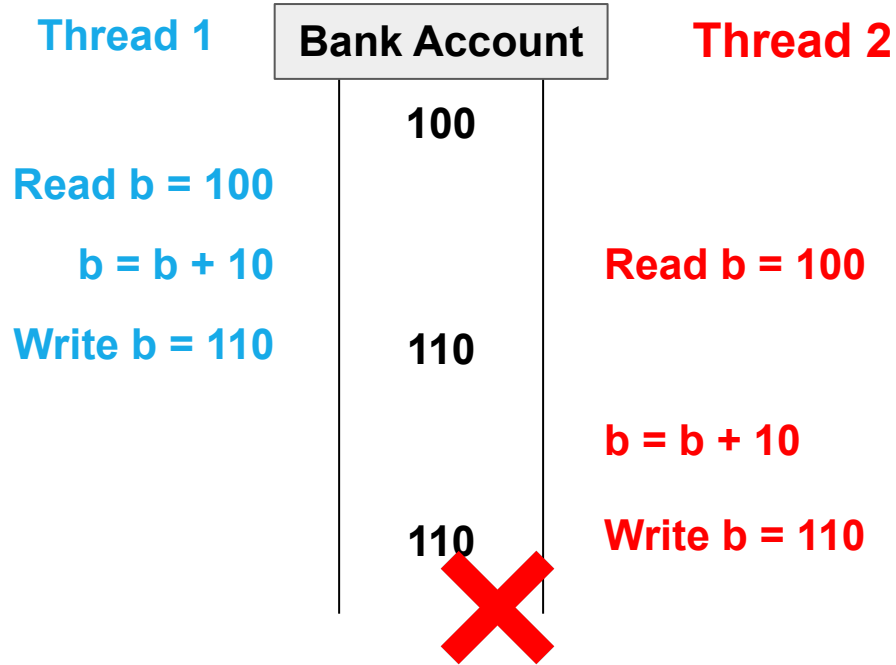
Locks - limit access to a critical section

Channels - pass information across processes using a queue

Example: Bank account



Example: Bank account



What went wrong?

Changes to balance are not *atomic*

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

What went wrong?

Suppose the function is called in two threads, with the **Thread 1** chosen to run first.

Thread 1

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

Thread 2

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

What went wrong?

Suppose the function is called in two threads, with the **Thread 1** chosen to run first.

Thread 1

```
func Deposit(amount) {  
    read balance  
    ↓  
    balance = balance+amount  
    write balance  
}
```

Thread 2

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

What went wrong?

Then, an interrupt happens, and the OS scheduler selects **Thread 2** to run.

Thread 1

```
func Deposit(amount) {  
    read balance  
    ↓  
    balance = balance+amount  
    write balance  
}
```

Thread 2

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



What went wrong?

Thread 1 did not write new balance to shared storage, so **Thread 2** reads the old value.

Thread 1

```
func Deposit(amount) {  
    read balance  
    ↓  
    balance = balance+amount  
    write balance  
}
```

Thread 2

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



What went wrong?

This is called a **race condition**.

Thread 1

```
func Deposit(amount) {  
    read balance  
    ↓  
    balance = balance+amount  
    write balance  
}
```

Thread 2

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



Solution - Locks

Changes to balance are now *atomic*.

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

Critical section

Good Video Explanations

<https://www.youtube.com/watch?v=FY9livorrJI>

<https://www.youtube.com/watch?v=LjWug2tvSBU>

Locks in Go

```
package account
```

```
import "sync"
```

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

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

```
func (a *Account) CheckBalance() int  
{
```

```
    defer a.mu.Unlock()  
    return a.balance  
}
```

```
func (a *Account) Withdraw(v int) {
```

```
    a.balance -= v
```

```
}
```

```
func (a *Account) Deposit(v int) {
```

```
    a.balance += v
```

```
}
```

Locks in Go

```
package account
```

```
import "sync"
```

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

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

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

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

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

Read Write Locks in Go

```
package account

import "sync"

type Account struct {
    balance int
    // ...
}

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

```
func (a *Account) CheckBalance() int
{
    // ...
    return a.balance
}

func (a *Account) Withdraw(v int) {
    // ...
    a.balance -= v
}

func (a *Account) Deposit(v int) {
    // ...
    a.balance += v
}
```

Read Write Locks in Go

```
package account
```

```
import "sync"
```

```
type Account struct {  
    balance int  
    rwLock sync.RWMutex  
}
```

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

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

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

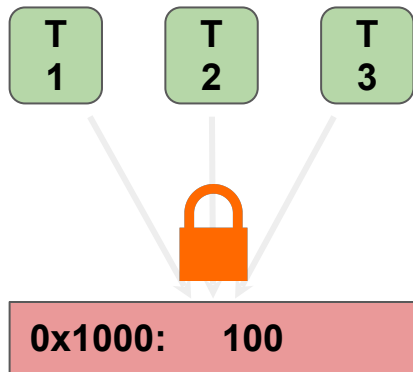
```
func (a *Account) Deposit(v int) {  
    a.rwLock.Lock()  
    defer a.rwLock.Unlock()  
    a.balance += v  
}
```

Two Solutions to the Same Problem

Locks:

Multiple threads can reference same memory location

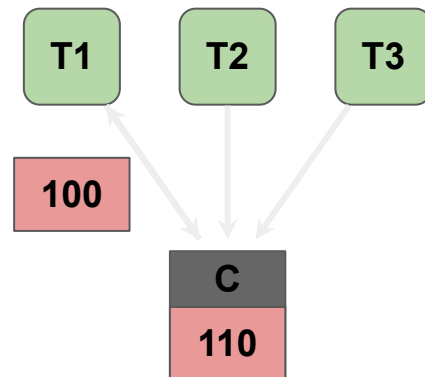
Use lock to ensure only one thread is updating it at any given time



Channels:

Data item initially stored in channel

Threads must request item from channel, make updates, and return item to channel



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

```
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{
        balance: 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) {
    // ???
}

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{  
        balance: 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) {  
    //???  
}
```

Bank Account Code (using channels)

```
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{
        balance: 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

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

```
for {  
    select {  
        case money := <-dad:  
            buySnacks(money)  
        case money := <-mom:  
            buySnacks(money)  
    }  
}
```

Select statement

`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

```
// Asynchronously request an answer
// from server, timing out after X
// seconds
result := make(chan int)
timeout := make(chan bool)

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

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

// Wait on both channels
select {
    case res := <-result:
        handleResult(res)

    case <-timeout:
        fmt.Println("Timeout!")
}
```

Exercise: Implementing a mutex using channels

```
type Lock struct {  
    // ???  
}  
  
func NewLock() Lock {  
    // ???  
}  
  
func (l *Lock) Lock() {  
    // ???  
}  
  
func (l *Lock) Unlock() {  
    // ???  
}
```

Exercise: Implementing a mutex using channels

```
type Lock struct {  
    ch chan bool  
}  
  
func NewLock() Lock {  
    // ???  
}  
  
func (l *Lock) Lock() {  
    // ???  
}  
  
func (l *Lock) Unlock() {  
    // ???  
}
```

Exercise: Implementing a mutex using channels

```
type Lock struct {  
    ch chan bool  
}  
  
func NewLock() Lock {  
    lock := Lock{make(chan bool, 1)}  
    lock.ch <- true  
    return lock  
}  
  
func (l *Lock) Lock() {  
    // ???  
}  
  
func (l *Lock) Unlock() {  
    // ???  
}
```

Exercise: Implementing a mutex using channels

```
type Lock struct {  
    ch chan bool  
}  
  
func NewLock() Lock {  
    lock := Lock{make(chan bool, 1)}  
    lock.ch <- true  
    return lock  
}  
  
func (l *Lock) Lock() {  
    <-lock.ch  
}  
  
func (l *Lock) Unlock() {  
    // ???  
}
```

Exercise: Implementing a mutex using channels

```
type Lock struct {  
    ch chan bool  
}  
  
func NewLock() Lock {  
    lock := Lock{make(chan bool,  
1)}  
    lock.ch <- true  
    return lock  
}  
  
func (l *Lock) Lock() {  
    <-lock.ch  
}  
  
func (l *Lock) Unlock() {  
    lock.ch <- true  
}
```

Outline

Two synchronization mechanisms

Locks

Channels

MapReduce

Application: Word count

How much wood would a woodchuck chuck if a woodchuck could chuck wood?



how: 1, much: 1, wood: 2, would: 1, a: 2, woodchuck: 2, chuck: 2, if: 1, could: 1

Application: Word count

Locally: tokenize and put words in a hash map

How do you parallelize this?

Partition the document into n partitions.

Build n hash maps, one for each partition

Merge the n hash maps (by key)

How do you do this in a distributed environment?



When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

Input document



When in the Course of human events, it becomes necessary for one people to

~~dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.~~



Partition

Shard 1

Shard 2

Shard 3

Shard 4

Shard 5

When in the Course of human events, it becomes necessary for one people to

~~dissolve the political bands which have connected them with another, and to assume,~~

~~among the Powers of the earth, the separate and equal station to which the Laws of~~

~~Nature and of Nature's God entitle them, a decent respect to the opinions of mankind~~

~~requires that they should declare the causes which impel them to the separation.~~

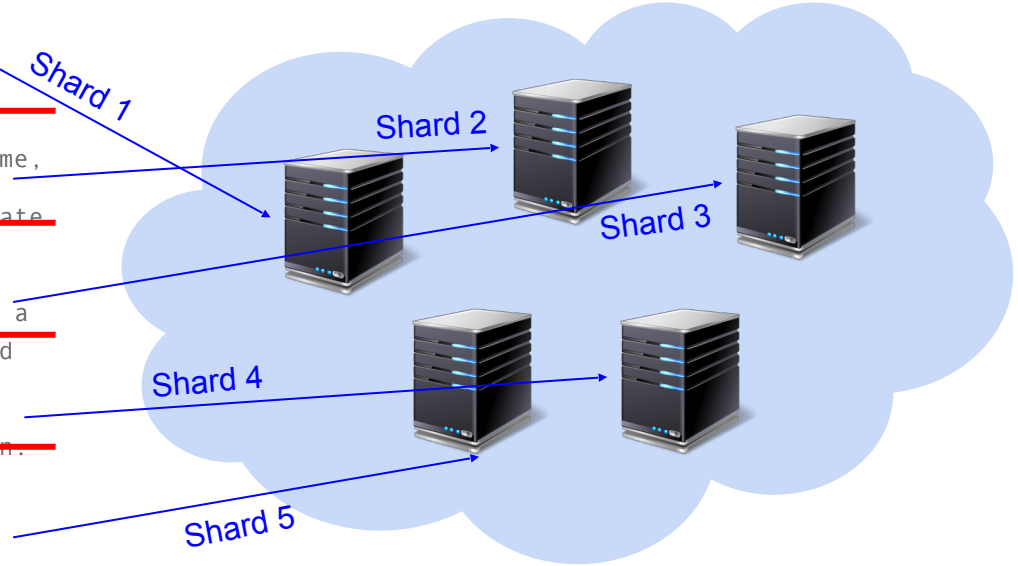
Shard 1

Shard 2

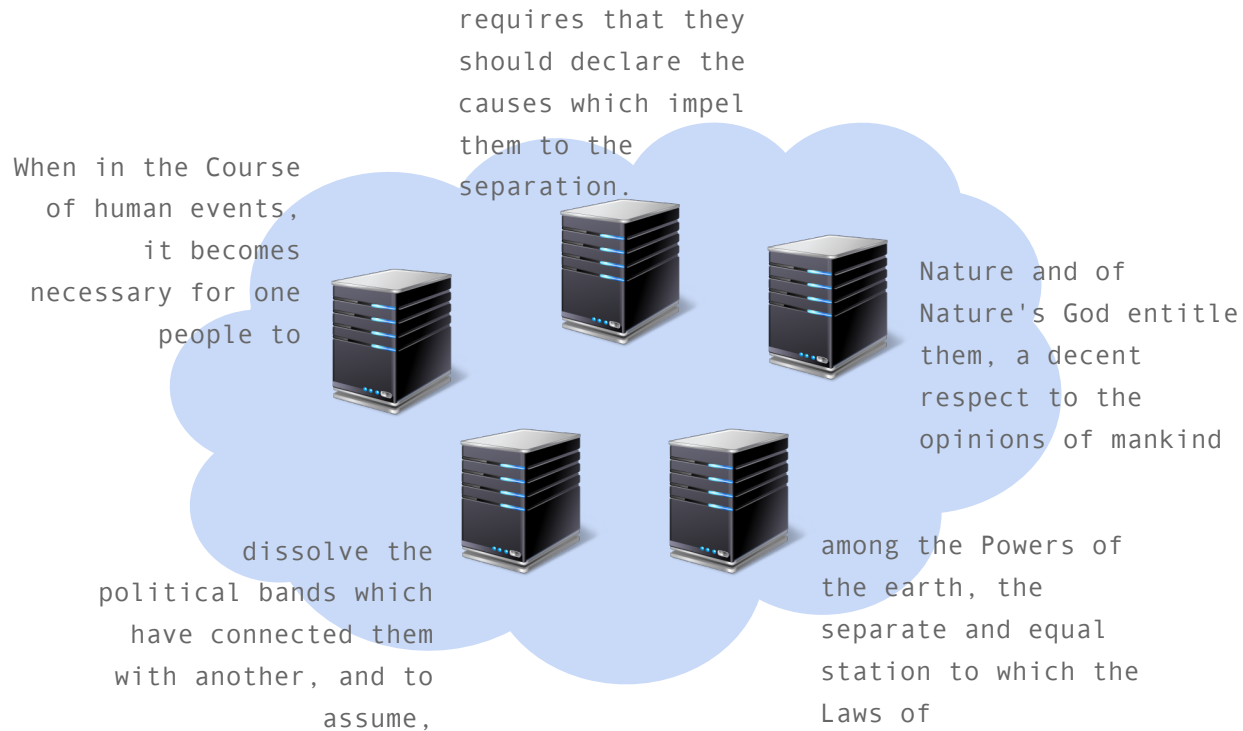
Shard 3

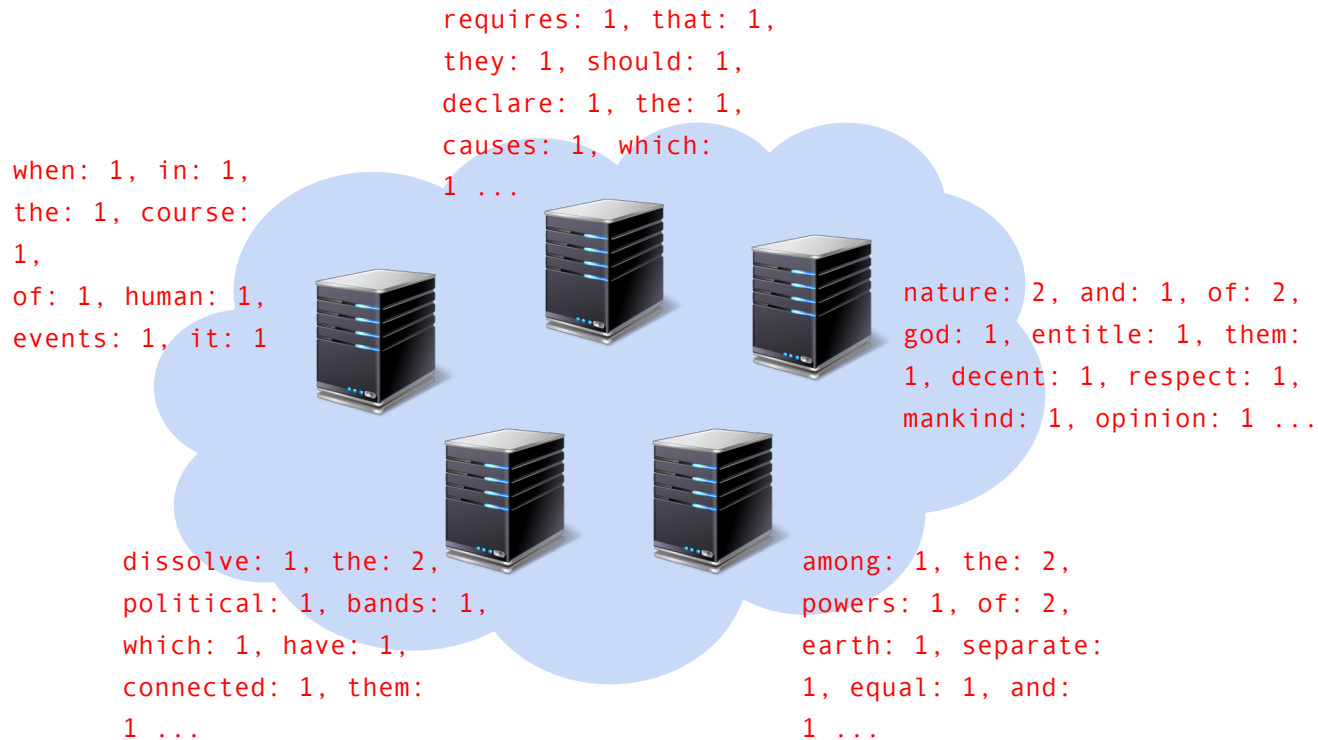
Shard 4

Shard 5



Partition





Compute word counts locally

when: 1, in: 1,
the: 1, course:
1,
of: 1, human: 1,
events: 1, it: 1

requires: 1, that: 1,
they: 1, should: 1,
declare: 1, the: 1,
causes: 1, which:
1 ...

nature: 2, and: 1, of: 2,
god: 1, entitle: 1, them:
1, decent: 1, respect: 1,
mankind: 1, opinion: 1 ...

Now what... How to merge results?

dissolve: 1, the: 2,
political: 1, bands: 1,
which: 1, have: 1,
connected: 1, them:
1 ...

among: 1, the: 2,
powers: 1, of: 2,
earth: 1, separate:
1, equal: 1, and:
1 ...

Compute word counts locally

Merging results computed locally

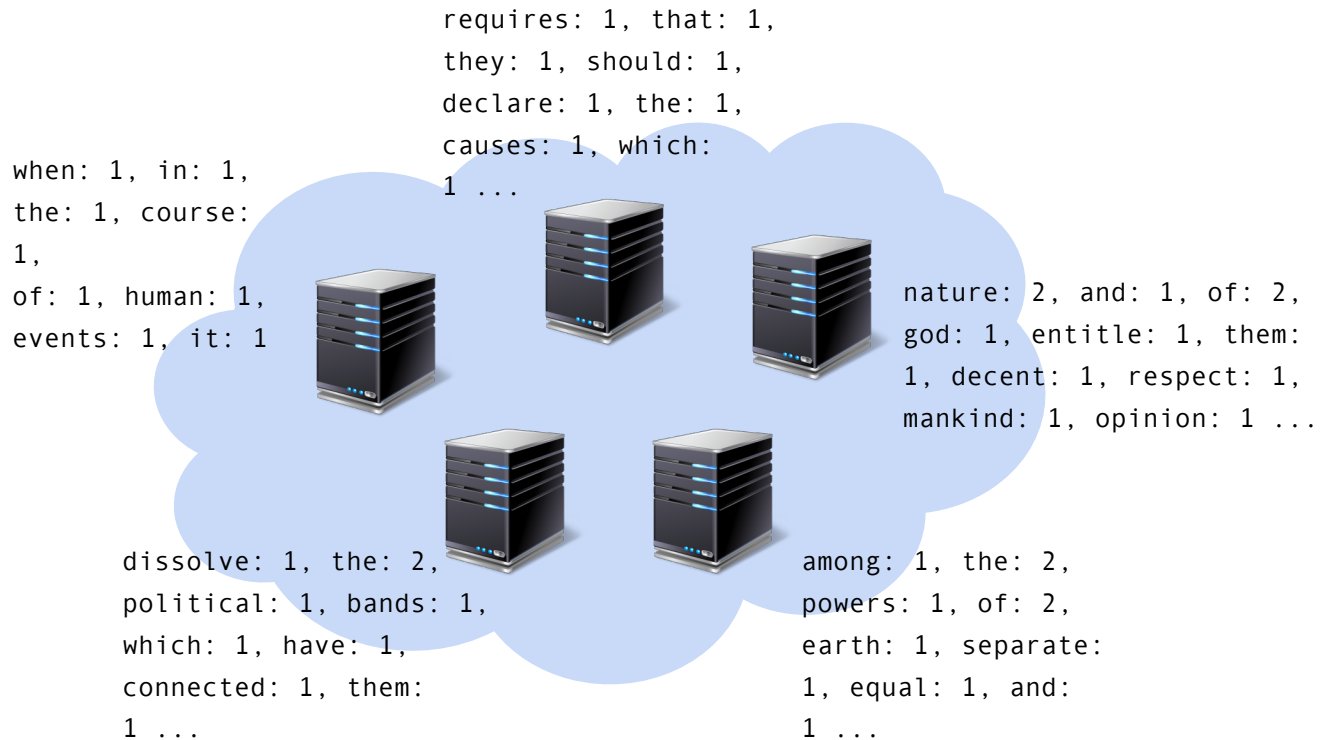
Several options

Don't merge — requires additional computation for correct results

Send everything to one node — what if data is too big? Too slow...

Partition key space among nodes in cluster (e.g. [a-e], [f-j], [k-p] ...)

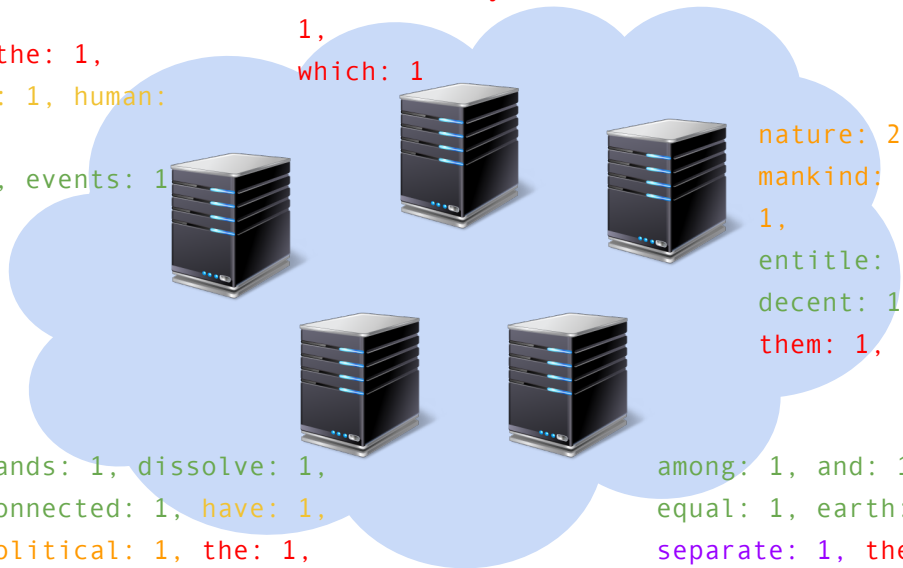
1. Assign a key space to each node
2. Split local results by the key spaces
3. Fetch and merge results that correspond to the node's key space



[a-e]
[f-j]
[k-p]
[q-s]
[t-z]

when: 1, the: 1,
in: 1, it: 1, human:
1,
course: 1, events: 1
of: 1

causes: 1, declare: 1,
requires: 1, should: 1,
that: 1, they: 1, the:
1,
which: 1

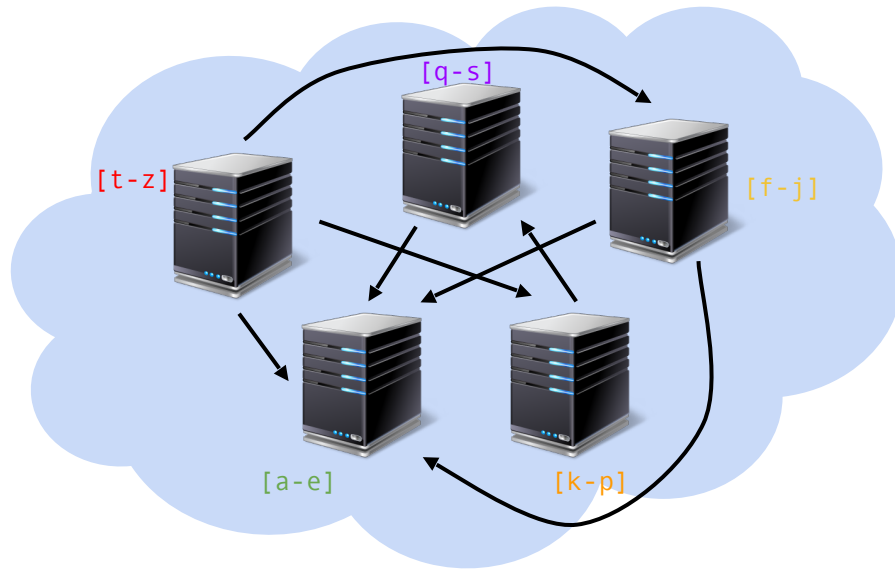


nature: 2, of: 2,
mankind: 1, opinion:
1,
entitle: 1, and: 1,
decent: 1, god: 1,
them: 1, respect: 1,

bands: 1, dissolve: 1,
connected: 1, have: 1,
political: 1, the: 1,
them: 1, which: 1

among: 1, and: 1,
equal: 1, earth: 1,
separate: 1, the: 2,
powers: 1, of: 2

Split local results by key space



All-to-all shuffle

[a-e]
[f-j]
[k-p]
[q-s]
[t-z]

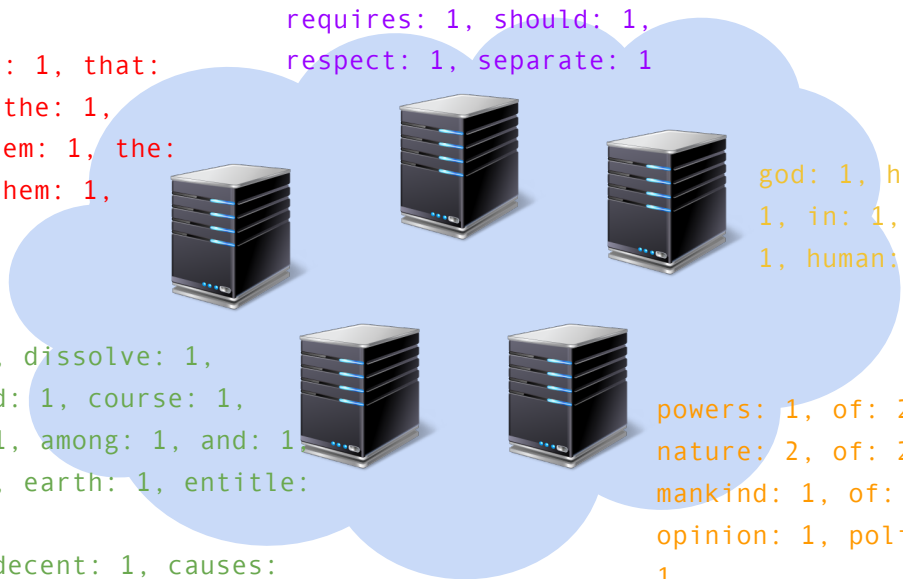
when: 1, the: 1, that:
1, they: 1, the: 1,
which: 1, them: 1, the:
2, the: 1, them: 1,
which: 1

requires: 1, should: 1,
respect: 1, separate: 1

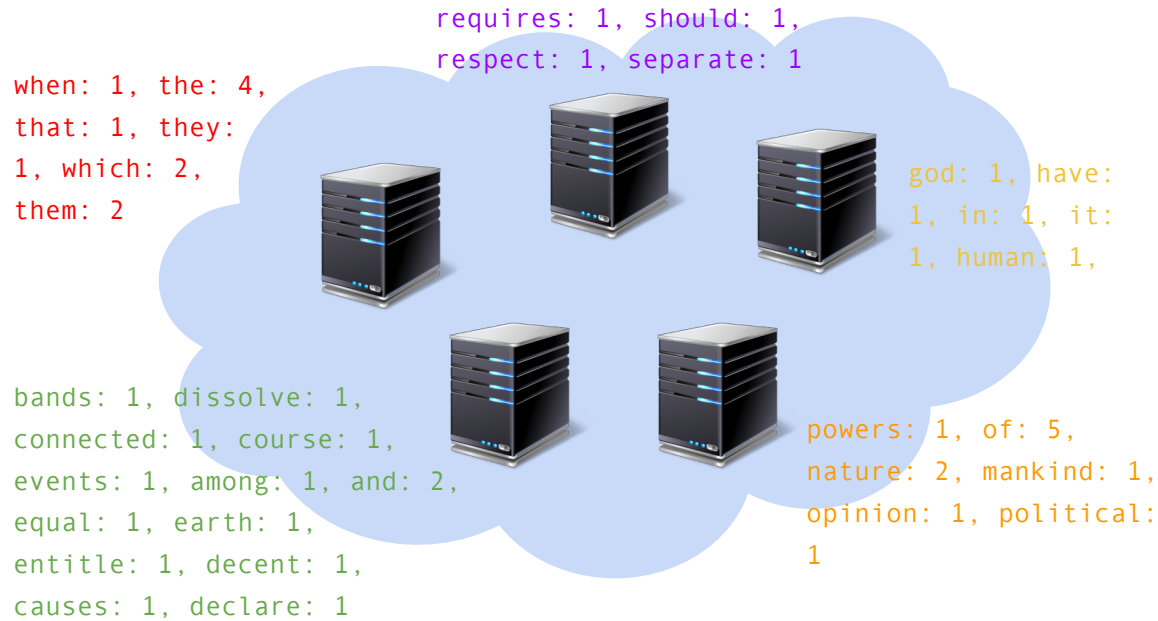
bands: 1, dissolve: 1,
connected: 1, course: 1,
events: 1, among: 1, and: 1
equal: 1, earth: 1, entitle:
1,
and: 1, decent: 1, causes:
1,
declare: 1

god: 1, have:
1, in: 1, it:
1, human: 1,

powers: 1, of: 2,
nature: 2, of: 2,
mankind: 1, of: 1,
opinion: 1, political:
1



Note the duplicates...



Merge results received from other nodes

Mapreduce

Partition dataset into many chunks

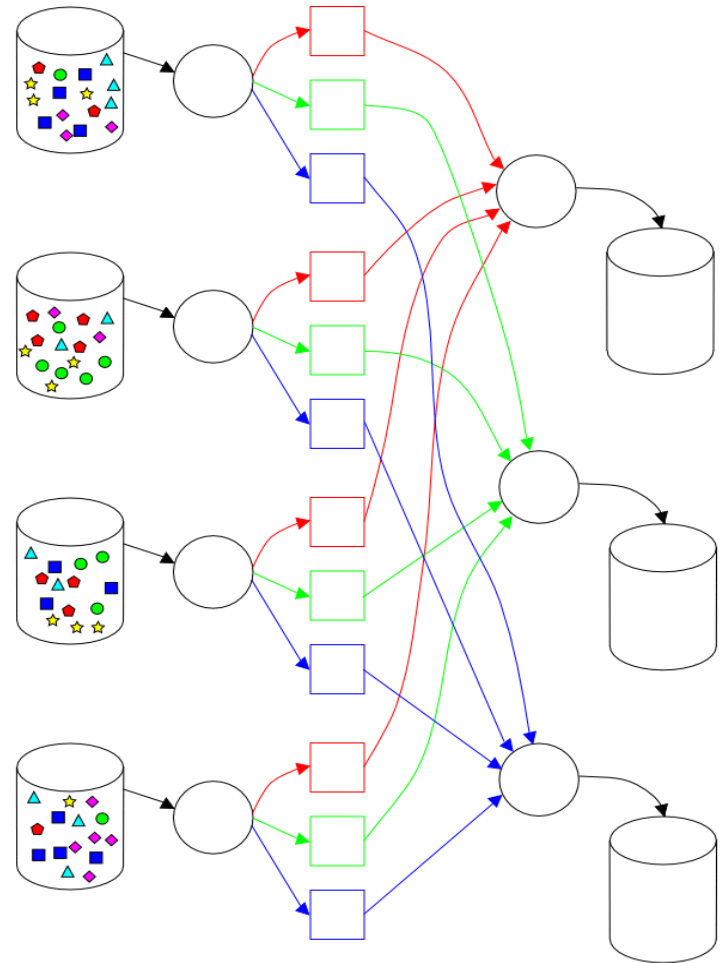
Map stage: Each node processes one or more chunks locally

Reduce stage: Each node fetches and merges partial results from all other nodes

Visualizing MapReduce

Note the 3 stages:

- Map
- Reduce
- Shuffle



Mapreduce Interface

map(key, value) -> list(<k', v'>)

Apply function to (key, value) pair

Outputs list of intermediate pairs

Mapreduce: Word count

map(key, value):

```
// key = document name  
// value = document contents  
for each word w in value:  
    emit (w, 1)
```

Mapreduce Interface

map(key, value) -> list(<k', v'>)

Apply function to (key, value) pair

Outputs list of intermediate pairs

reduce(key, list<value>) -> <k', v'>

Applies aggregation function to values

Outputs result

Mapreduce: Word count

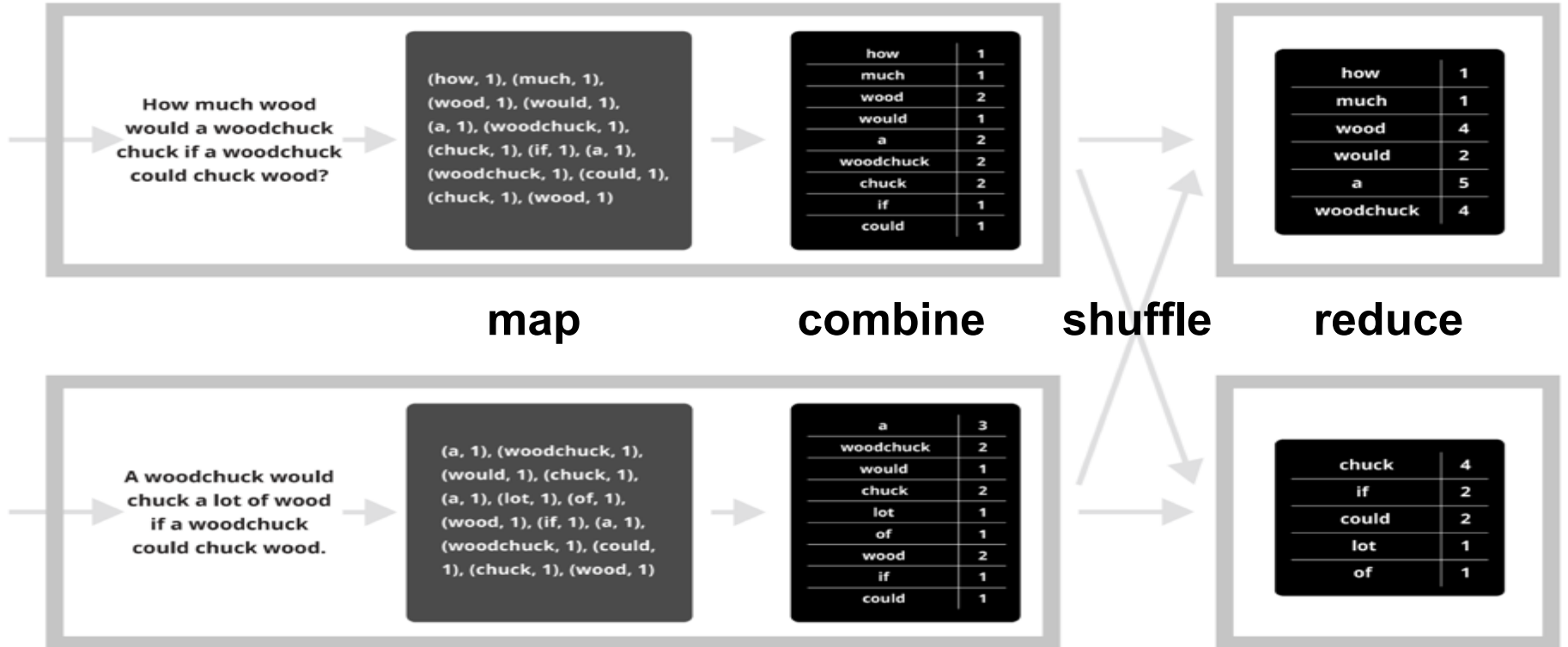
map(key, value) :

```
    // key = document name
    // value = document contents
    for each word w in value:
        emit (w, 1)
```

reduce(key, values) :

```
    // key = the word
    // values = number of occurrences of that word
    count = sum(values)
    emit (key, count)
```

Mapreduce: Word count



Why is implementing MapReduce hard?

- Failure is common
 - Even if each machine is available $p = 99.999\%$ of the time, a datacenter with $n = 100,000$ machines still encounters failures $(1 - p^n) = 63\%$ of the time
 - Data skew causes unbalanced performance across cluster
- Problems occur at scale.
- Hard to debug!

MapReduce



2004

2007

2011

2012

2015

Dryad



Cloud Dataflow