Big Data Processing

Georgios Gousios

2018-01-26

Contents

List of Tables

List of Figures

Chapter 1

Preface

TI2736-B: Big Data Processing is a second year BSc course at TU Delft that, as its title says, aims is to teach students how to Hi there, this is my great book. process big data. It is part of the "Data Processing" *variantblok*, that also includes courses like Data Mining and Computational Intelligence.

In Jan 2017, I took over the course; my collegue who was teaching the course before me had already done a fantastic job with it. I decided to follow a different route; instead of presenting specific systems or technologies, I would focus mostly on how big data systems are *programmed*.

Structure of the book

Chapters **??** introduces a new topic, and …

Acknowledgments

A lot of people helped me when I was writing this book. Wouter Zorgdrager

Georgios Gousios

Copyright information

Chapter 2

Big and Fast Data

What is big data?

An overloaded, fuzzy term

> "Data too large to be efficiently processed on a single computer"
>
> "Massive amounts of diverse, unstructured data prodiced by high-performance applications"

2.1 How big is "Big"?

Typical numbers associated with Big Data

- 2.5 Exabytes (10^3 TB) produced daily
- IoT: 8.4 Billion devices with internet access
- Amazon: 600 orders per second up (2016) from 35 in 2012
- Alibaba in 2015: 400 orders per second
- Google in 2016: 2+ trillion searches or 65k per second
  - Each query involves > 1000 machines
  - Each search touches 200+ services

How big is "Big"? – Instagram

- 250M daily users, clicking around the app
- 50M photos daily

Figure 2.1: Instagram

- Most followed user: 113M followers

How big is "Big"? – FaceBook

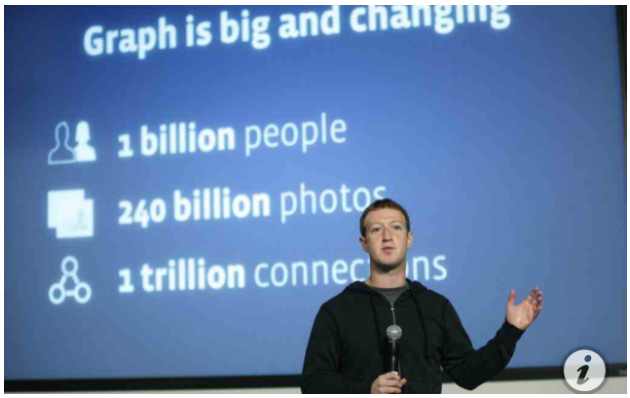


Figure 2.2: FaceBook

Warning: numbers (⇑) from 2014! Today on FB:

- 2 Billion users
- 1.32 Billion active users per day
- 300 million photos per day (136k/min)
- Every min: 510k comments, 293k status updates

2.2 The many Vs of Big data

Main Vs, by Doug Laney

- **Volume**: large amounts of data
- **Variety**: data comes in many different forms from diverse sources
- **Velocity**: the content is changing quickly

More Vs

- **Value**: data alone is not enough; how can value be derived from it?
- **Veracity**: can we trust the data? How accurate is it?
- **Validity**: ensure that the interpreted data is sound
- **Visibility**: data from diverse sources need to be stitched together

Volume

We call Big Data big because it is *really big*:

- 90% of all the data ever was created in the last 2 years
- By 2020, each person will generate 1.7MB per sec
- The Big data / data analytics industry will be worth $203 Billion in 2020

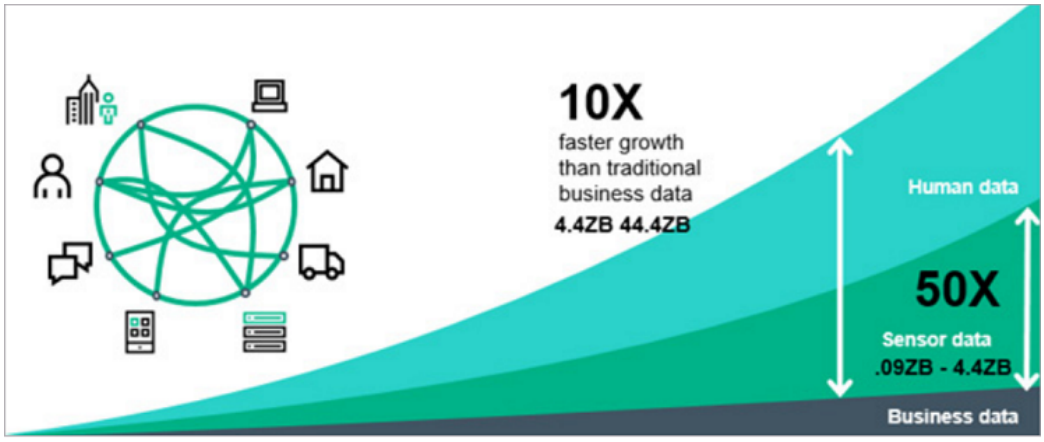


Figure 2.3: Data growth rate

Variety

- Structured data: SQL tables, images, format is known

- Semi-structured data: JSON, XML
- Unstructured data: Text, mostly

We often need to combine various data sources of different types to come up with a result

Velocity

Data is not just big; it is generated and needs to be processed fast. Think of:

- Datacenters writing to log files
- IoT sensors reporting temperatures around the globe
- Twitter: 500 million tweets a day (or 6k/sec)
- Stock Markets: high-frequency trading (latency costs money)
- Online advertising

Data needs to be processed with soft or hard real-time guarantees

Big Data processing

- The **ETL** cycle
  - **E**xtract: Convert raw or semi-structured data into structured data
  - **T**ransform: Convert units, join data sources, cleanup etc
  - **L**oad: Load the data into another system for further processing
- Big data *engineering* is concerned with building *pipelines*
- Big data *analytics* is concerned with *discovering patters*

How to process all this data?

- **Batch processing**: All data exists in some data store, a program processes the whole dataset at once
- **Stream processing**: Processing of data as they arrive to the system

2 basic approaches to distribute data processing operations on lots of machines

- Divide the **data** in chunks, apply the same algorithm on all chunks (concurrency)
- Divide the **problem** in chunks, run it on a cluster of machines (parallelism)

Large-scale computing

Not a new discipline:

- Cray-1 appeared in the late '70s
- Physicists used super computers for simulations in the '80s
- Shared-memory designs still in large scale use (e.g. TOP500 supercomputers)

What is new?

Large scale processing on **distributed**, **commodity** computers, enabled by advanced **software** using **elastic** resource allocation.

Software (not HW!) is what drives the Big Data industry

2.3 A brief history of Big Data tech

- 2003: Google publishes the Google Filesystem paper, a large-scale distributed file system
- 2004: Google publishes the Map/Reduce paper, a distributed data processing abstraction
- 2006: Yahoo creates and open sources Hadoop, inspired by the Google papers
- 2006: Amazon lunches its Elastic Compute Cloud, offering cheap, elastic resources
- 2007: Amazon publishes the DynamoDB paper, sketches the blueprints of a cloud-native database
- 2009 – onwards: The NoSQL movement. Schema-less, distributed databases defy the SQL way of storing data
- 2010: Matei Zaharia et al. publish the Spark paper, brings FP to in-memory computations
- 2012: Both Spark Streaming and Apache Flink appear, able to handle really high volume stream processing
- 2012: Alex Krizhevsky et al. publish their deep learning image classification paper re-igniting interest in neural networks and solidifying the value of big data

The Big Data Tech Landscape 2017

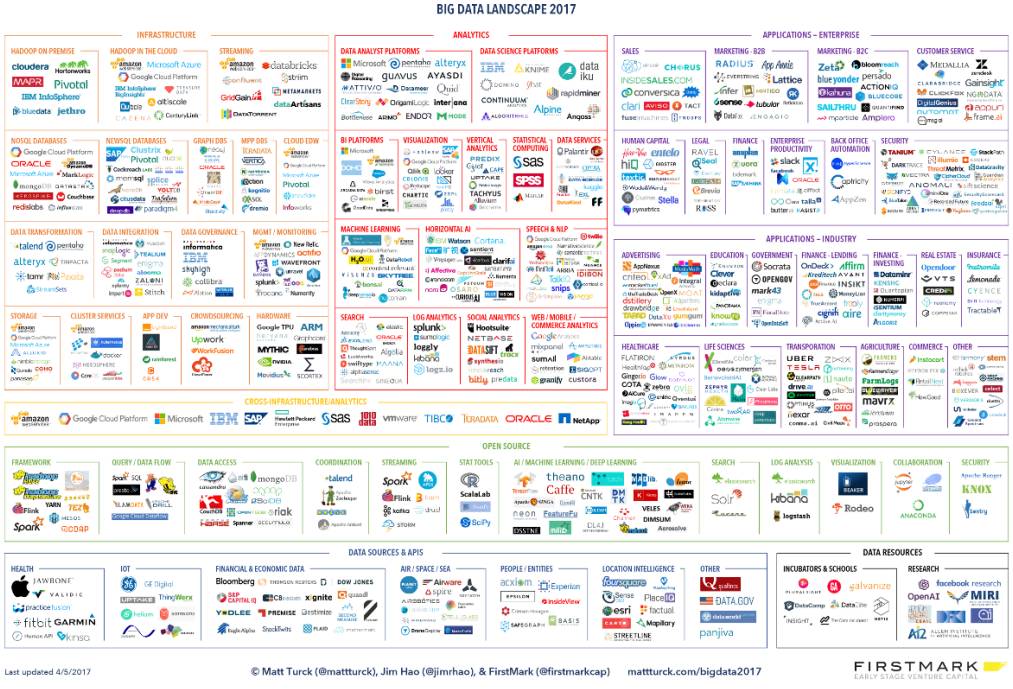


Figure 2.4: The big data landscape

Progress is mostly industry-driven

D:: Most advancement in Big Data technologies came from the industry. The universities only started contributing late. Why?

. . .

Data is the new oil

2.4 The importance of data

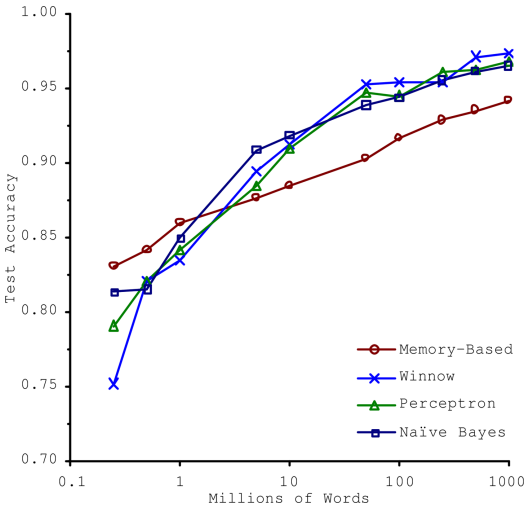


Figure 2.5: Importance of data

Figure by Banko and Brill, 2001. They showed that simple algorithms perform better than complex ones when the data is big enough.

2.5 Typical problems solved with Big Data

- **Modeling**: What factors influence particular outcomes/behaviours?
- **Information retrieval**: Finding needles in haystacks, aka search engines

- **Collaborative filtering**: Recommending items based on items other users with similar tastes have chosen
- **Outlier detection**: Discovering outstanding transactions

Group exercise: Big data in practice

D: Identify a big data system you interact with every day. Try to answer the following questions:

- What types of data does it need to integrate?
- How much data does it process per day?
- How do the 3 (or 5) Vs apply to it?

Use this URL: http://bit.ly/big-data-course-1

Work with the people around you, for 5 mins

Chapter 3

Languages for Big Data processing

3.1 Scala and Python

The *de facto* languages of Big Data and data science are

- **Scala** Mostly used for data intensive systems
- **Python** Mostly used for data analytics tasks

Other languages include

- **Java** The "assembly" of big data systems; the language that most big data infrastructure is written into.
- **R** The statistician's tool of choice. Great selection of libraries for serious data analytics, great plotting tools.

In our course, we will be using Scala and Python. R will be used in the minor part.

Scala and Python from 10k feet

- Both support object orientation, functional programming and imperative programming
  - Scala's strong point is the combination of FP and OO
  - Python's strong point is the combination of OO and IP

- Python is *interpreted*, Scala is *compiled*

3.2 Hello world

Scala

```
object Hello extends App {
    println("Hello, world")
    for (i <- 1 to 10) {
      System.out.println("Hello")
    }
}
```

- Scala is compiled to JVM bytecode
- Can interoperate with JVM libraries
- Scala is not sensitive to spaces/tabs. *Blocks* are denoted by `{` and `}`

Python

```
#!/usr/bin/env python

for i in range(1, 10):
  print "Hello, world"
```

- Python is interpreted
- Python is indentation sensitive: *blocks* are denoted by a TAB or 2 spaces.

3.3 Declarations

Scala

```
val a: Int = 5
val b = 5
b = 6 // re-assignment to val

// Type of foo is infered
```

```
val foo = new ImportantClass(...)

var a = "Foo"
a = "Bar"
a = 4 // type mismatch
```

- *Type inference* used extensively
- Two types of variables: `val`s are single-assignment, `var`s are multiple assignment

Python

```
a = 5
a = "Foo"

a = ImportantClass(...)
```

- No restrictions on types or assignment rules

Declaring functions

Scala

```
def max(x: Int, y: Int): Int = {
  if (x >= y)
    x
  else
    y
}
```

- Scala is statically typed
- The return value depends on the evaluation of *expressions*. The last evaluated expression determines the result (also the function return type)

Python

```
def max(x,y):
  if x >= y:
    return x
  else:
```

```
    return y
```

- Python is dynamically typed
- Mostly based on *statements*, thus we need to return explicitly

Scala

```
def bigger(x: Int, y: Int,
  f: (Int,Int) => Boolean) = {

  f(x, y)
}

bigger (1, 2, (x, y) => (x < y))
bigger (1, 2, (x, y) => (x > y))
// Compile error
bigger (1, 2, x => x)
```

Python

```
def bigger(x, y, f):
  return f(x, y)

bigger(1,2, lambda x,y: x > y)
bigger(1,2, lambda x,y: x < y)
# Runtime error
bigger(1,2, lambda x: x)
```

In both cases, `bigger` is a higher-order function, i.e. a function whose behaviour is parametrised by another function. `f` a function parameter. To call a HO function, we need to construct a function with the appropriate arguments. The compiler checks this in the case of Scala.

Declaring classes

Scala

```
class Foo(val x: Int,
          var y: Double = 0.0)

// Type of a is infered
```

```
val a = new Foo(1, 4.0)
println(a.x) //x is read-only
println(a.y) //y is read-write
a.y = 10.0
println(a.y) //y is read-write
a.y = "Foo"   // Type mismatch, y is double
```

- `val` means a read-only attribute. `var` is read-write
- A default constructor is created automatically

Python

```
class Foo():
  def __init__(self, x, y):
    self.x = x
    self.y = y

a = Foo(3,2)
print a.x
a.x = "foo"
print a.x
```

- Attributes are declared in the *constructor*
- Attributes are by default read-write
- No types are enforced

3.4 Object-Oriented programming

Scala

```
class Foo(val x: Int,
          var y: Double = 0.0)

class Bar(x: Int, y: Int, z: Int)
  extends Foo(x, y)

trait Printable {
```

```
  val s: String
  def asString() : String
}

class Baz(x: Int, y: Int, private z: Int)
  extends Foo(x, y)
  with Printable
```

Python

```
class Foo():
  def __init__(self, x, y):
    self.x = x
    self.y = y

class Bar(Foo):
  def __init__(self, x, y, z):
    Foo.__init__(self, x, y)
    self.z = z
```

In both cases, the traditional rules of method overriding apply. Traits in Scala are similar to interfaces in Java; in addition, declared methods may be implemented and they can include attributes (state).

Case classes in Scala

```
case class Address(street: String, number: Int)
case class Person(name: String, address: Address)

val a1 = new Address("Mekelweg", 4)
val p1 = new Person("Georgios", a1)

val p2 = new Person("Georgios", a1)

p1 == p2 // True
```

Case classes are blueprints for immutable objects. We use them to represent data records. Scala automatically implements `hashCode` and `equals` for them, so we can compare them directly.

Pattern matching in Scala

Pattern matching is `if..else` on steroids

```
// Code for demo only, won't complile

value match {
  // Match on a value, like if
  case 1 => "One"
  // Match on the contens of a list
  case x :: xs => "The remaining contents are " + xs
  // Match on a case class, extract values
  case Email(addr, title, _) => s"New email: $title..."
  // Match on the type
  case xs : List[_] => "This is a list"
  // With a pattern guard
  case xs : List[Int] if x.head == 5 => "This is a list of integers"
  case _ => "This is the default case"
}
```

Reading ahead

This is by far not an introduction to either programming languages. Please read more here

- Scala documentation
- Python documentation

Pick one and become good at it!

- BSc student? Pick Scala
- Minor student? Pick Python

Chapter 4

Programming for Big Data

4.1 Basic Data Types

In this section, we review the basic data types we use when processing data.

Types of data

- **Unstructured**: Data whose format is not known
  - Raw text documents
  - HTML pages
- **Semi-Structured**: Data with a known format.
  - Pre-parsed data to standard formats: JSON, CSV, XML
- **Structured**: Data with known formats, linked together in graphs or tables
  - SQL or Graph databases
  - Images

D: *What types of data are more convenient when processing?*

Sequences / Lists

Sequences or **Lists** or **Arrays** represent consecutive items in memory

In Python:

```
a = [1, 2, 3, 4]
```

In Scala

```
val l = List(1,2,3,4)
```

Basic properties:

- Size is bounded by memory
- Items can be accessed by an index: `a[1]` or `l.get(3)`
- Items can only inserted at the end (*append*)
- Can be sorted

Sets

Sets store values, without any particular order, and no repeated values.

```
scala> val s = Set(1,2,3,4,4)
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
```

Basic properties:

- Size is bounded by memory
- Can be queried for containment
- Set operations: union, intersection, difference, subset

Maps or Dictionaries

Maps or **Dictionaries** or **Associative Arrays** is a collection of `(k,v)` pairs in such a way that each `k` appears only once.

Some languages have build-in support for Dictionaries

```
a = {'a' : 1, 'b' : 2}
```

Basic properties:

- One key always corresponds to one value.
- Accessing a value given a key is very fast ($\approx O(1)$)

Nested data types: Graphs

A graph data structure consists of a finite set of vertices or nodes, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph.

- Nodes can contain attributes

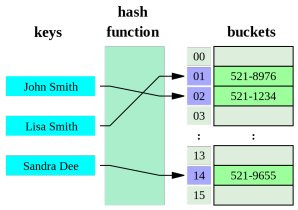


Figure 4.1: A HashTable

- Vertices can contain weights and directions

Graphs are usually represented as `Map[Node, List[Vertex]]`, where

```
case class Node(id: Int, attributes: Map[A, B])
case class Vertex(a: Node, b: Node, directed: Option[Boolean],
                  weight: Option[Double] )
```

Nested data types: Trees

Ordered graphs without loops

```
a = {"id": "5542101946", "type": "PushEvent",
    "actor": {
      "id": 801183,
      "login": "tvansteenburgh"
    },
    "repo": {
      "id": 42362423,
      "name": "juju-solutions/review-queue"
   }}
```

If we parse the above JSON in almost any language, we get a series of nested maps

```
Map(id -> 5542101946,
    type -> PushEvent,
    actor -> Map(id -> 801183.0, login -> tvansteenburgh),
    repo -> Map(id -> 4.2362423E7, name -> juju-solutions/review-queue)
)
```

Relations

An n-tuple is a sequence of n elements, whose types are known.

```
val record = Tuple4[Int, String, String, Int]
                    (1, 'Georgios', 'Mekelweg', '4')
```

A *relation* is a `Set` of n-tuples $(d1, d2, ..., dn)$.

Relations are very important for data processing, as they form the theoretical framework (Relational Algebra) for *relational (SQL) databases*.

Typical operations on relations are insert, remove and *join*. Join allows us to compute new relations by joining existing ones on common fields.

Relations example

```
val addr1 = Tuple4[Int, String, String, Int](1, "Gebouw 35",
                                                "Mekelweg", 5)
val addr2 = Tuple4[Int, String, String, Int](2, "Gebouw 36",
                                                "Drebbelweg", 4)

val addr = Set(addr1, addr2)

val georgios = Tuple3[Int, String, Int](1, "Georgios", 1)
val wouter = Tuple3[Int, String, Int](1, "Wouter", 2)
val joris = Tuple3[Int, String, Int](1, "Joris", 4)

val people = Set(georgios, wouter, joris)
```

Q: How can we get a list of buildings and the people that work there?

. . .

Key/Value pairs

A key/value pair (or KV) is a special type of a Map, where a key `k` does not have to appear once.

Key/Value pairs are usually implemented as a `Map` whose keys are of a sortable type `K` (e.g. `Int`) and values are a `Set` of elements of type `V`.

```
val kv = Map[K, Set[V]]()
```

K and V are flexible: that's why the Key/Value abstraction is key to NoSQL

databases, including MongoDB, DynamoDB, Redis etc. Those databases sacrifice, among others, type safety for scaling.

4.2 Functional programming in a nutshell

In this section, we discuss the basics of functional programming, as those apply to data processing with tools like Hadoop, Spark and Flink.

Functional programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data (Wikipedia).

Functional programming characteristics:

- Absence of **side-effects**: A function, given an argument, always returns the same results irrespective of and without modifying its environment.
- **Higher-order functions**: Functions can take functions as arguments to parametrise their behavior
- **Lazyness**: The art of waiting to compute till you can wait no more

Function signatures

$foo(x : [A], y : B) \rightarrow C$

- foo: function name
- x and y: Names of function arguments
- $[A]$ and B: Types of function arguments.
- $\rightarrow$: Denotes the return type
- C: Type of the returned result
- $[A]$: Denotes that type A can be traversed

We read this as: *Function foo takes as arguments an array/list of type A and an argument of type B and returns an argument of type C*

Q: *What does the following function signature mean?* $f(x : [A], y : (z : A) \rightarrow B) \rightarrow [B]$

Side effects

A function has a side effect if it **modifies** some state outside its scope or has an observable interaction with its calling functions or the outside world besides returning a value.

```
max = -1

def ge(a, b):
    global max
    if a >= b:
        max = a ## <- Side effect!
        return True
    else:
        max = b
        return False
```

As a general rule, any function that returns nothing (`void` or `Unit`) does a side effect!

Examples of side effects

- Setting a field on an object: OO is not FP!
- Modifying a data structure in place: In FP, data structures are always persistent.
- Throwing an exception or halting with an error: In FP, we use types that encapsulate and propagate erroneous behaviour
- Printing to the console or reading user input, reading writing to files or the screen: In FP, we encapsulate external resources into *Monads*.

How can we write code that does something useful given those restrictions?

From OO to FP

Buying coffee in OO

```
class Cafe {

  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    cup
  }
```

```
}
```

Can you spot any problems with this code?

. . .

- `charge()` performs a side-effect: we need to contact the credit card company!
- `buyCoffee()` is *not testable*

Breaking external dependencies

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

Slightly better option, *but*:

. . .

- `Payments` has to be an interface
- We still need to perform side effects
- We need to inspect state within the `Payments` mock

. . .

Q: How can we buy 10 coffees?

Buying 10 coffees

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = { ... }

  def buyCoffees(cc: CreditCard, p: Payments, num: Int) = {
    for (i <- 1 to num) {
      buyCoffee(cc, p)
```

```
    }
  }
}
```

Seems to be working, *but*:

. . .

- No way to batch payments
- No way to batch checkouts

Removing side effects

Idea: how about instead of charging in place, we decouple the action of *buying* coffee from that of *charging* for it?

```
class Charge(cc: CreditCard, amount: Double) {
  def combine(other: Charge) = new Charge(cc, amount + other.amount)
  def pay = cc.charge(amount)
}

class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    (cup, Charge(cc, cup.price))
  }
}
```

Nice! We can now:

- Test
- Combine multiple `Charge`s in one
- Maintain in flight accounts for all customers

Buying 10 functional coffees

```
class Charge(cc: CreditCard, amount: Double) {
  def combine(other: Charge) = new Charge(cc, amount + other.amount)
  def pay = cc.charge(amount)
```

```
}

class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = { ...   }
  def buyCoffees(cc: CreditCard, num: Int): Seq[(Coffee, Charge)] =
    (1 to num).map(buyCoffee(cc))

  def checkout(cc: CreditCard, charges: Seq[Charge]) : Seq[Charge] = {
    charges.
      filter(charge => charge.cc == cc).
      foldLeft(new Charge(cc, 0.0)){(acc, x) => acc.combine(x)}.
      pay  // <- side-effect, but once, in one place.

    charges.filter(charge => charge.cc != cc)
  }
}
```

. . .

This example was adapted from the (awesome) FP in Scala book, by Chiusano and Bjarnason

Higher-Order functions

A higher order function is a function that can take a function as an argument or return a function.

```
class Array[A] {
  // Return elements that satisfy f
  def filter(f: A => Boolean) : Array[A]
}
```

In the context of BDP, high-order functions capture common idioms of processing data as enumarated elements, e.g. going over all elements, selectively removing elements and aggregating them.

Common higher-order functions on Lists

`map(xs: List[A], f: A => B) : List[B]` Applies `f` to all elements and

returns a new list

`flatMap(xs: List[A], f: A => List[B]) : List[B]` Like `map`, but flattens the result to a single list

`fold(xs: List[A], f: (B, A) => B, init: B) : B` Takes `f` of 2 arguments and an init value and combines the elements by applying `f` on the result of each previous application

`scan(xs: List[A], f: (B, A) => B, init: B) : List[B]` Like `fold`, but returns a list of all intermediate results

`filter(xs: List[A], f: A => Boolean) : List[A]` Takes a function that returns a boolean and returns all elements that satisfy it

Laziness

Laziness is an evaluation strategy which delays the evaluation of an expression until its value is needed.

- Seperating a pipeline construction from its evaluation
- Not requiring to read datasets in memory: we can process them in lazy-loaded batches
- Generating *infinite* collections
- Optimising execution plans

```
def primes(limit):
  sieve = [False]*limit

  for j in xrange(2, limit):
    if sieve[j]: continue
    yield j
    if j*j > limit: continue
    for i in xrange(j, limit, j):
      sieve[i] = True

print [x for x in primes(100)][1:10]
```

```
## [3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Lazy data pipelines

In tools like Spark and Flink, we always express computations in a lazy manner. This allows for optimizations before the actual computation is executed

```
# Word count in PySpark
text_file = sc.textFile("words.txt")
counts = text_file \
              .flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("results.txt")
```

Q: In the code above, when will Spark compute the result?

. . .

A: When `saveAsTextFile` is called!

Monads

Monads are a design pattern that defines how functions can be used together to build generic types. Practically, a monad is a value-wrapping type that:

- Has an `identity` function
- Has a `flatMap` function, that allows data to be converted between monad types

```
trait Monad[M[_]] {
  def unit[S](a: S) : M[S]
  def flatMap[S, T] (m: M[S])(f: S => M[T]) : Monad[T]
}
```

Monads are the tool FP uses to deal with (side-)effects

- Null points: `Option[T]`
- Exceptions: `Try[T]`
- Latency in asynchronous actions: `Future[T]`

Combining monads `flatMap`

`flatMap` enables us to join sequences of arbitrary types.

Example: `Futures` wrap values that will be eventually available.

```
def callWebService(): Future[R]
def heavyComputation(r: R): Future[V]

val r = callWebService().flatMap(r => heavyComputation2(r))
```

Example: `Options` wrap values that may be null.

```
def getArgument(k: String): Option[Arg]
def process(a: Arg): Option[Result]

getArgument("foo").
  flatMap(processArgument).
  getOrElse(new Result("default"))
```

Dealing with exceptions: Scala

```
object Converter extends App {
  def toInt(a: String): Try[Int] = Try{Integer.parseInt(a)}
  def toString(b: Int): Try[String] = Try{b.toString}

  val a = toInt("4").flatMap(x => toString(x))
  println(a)

  val b = toInt("foo").flatMap(x => toString(x))
  println(b)
}
```

`Try` is a type that can have 2 instances:

- `Success[T]`, where `T` represents the type of the result
- `Failure[E]`, where `E` represents the type of error, usually an exception

Dealing with exceptions: Java

. . .

Figure 4.2: Warning: Java Code Ahead!

. . .

```
public class Converter {
  public Integer toInt(String a) throws NumberFormatException {
    return Integer.parseInt(a)
  }
  public String toString(Integer a) throws NullPointerException {
    return b.toString();
  }

  public static void main(String[] args) {
    try {
      Integer five = toInt('5');
        try {
          return toString(five);
        } catch(NullPointerException e) {
          //baaah
        }
    } catch(NumberFormatException r) {
      //oooffff
    }
  }
}
```

Mapping effects to the type system

```
class Amazon {
  def login(login: String, passwd: String): Future[Amazon]
  def search(s: String): Future[Seq[String]]
}

class Main extends App {
  val amazon = new Amazon()
  val result =
    amazon.login("uname", "passwd") // Might fail
          .flatMap(a => a.search("foo"))
}
```

We now need to encode error handling, so that the compiler checks that we

are doing it propertly.

D: Which of the following should our new `search` return?

- `Seq[Future[Try]]`
- `Future[Seq[Try]]`
- `Future[Try[Seq]]`

4.3 Enumerating datasets

Before starting to process datasets, we need to be able to go over their contents in a systematic way. The process of visiting all items in a dataset is called *traversal*.

Big data sets

In a big data system:

- **Client** code processes data
- A **data source** is a container of data (e.g. array, database, web service)

There are two fundamental techniques for the client to process all available data in the data source

- **Iteration**: The client *asks* the data source whether there are items left and then *pulls* the next item.
- **Observation**: The data source *pushes* the next available item to a client end point.

D: *What are the relative merits of each technique?*

Iteration

In the context of BDP, iteration allows us to process finite-sized data sets without loading them in memory at once.

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}
```

Typical usage

```
val it = Array(1,2,3,4).iterator
while(it.hasNext) {
  val i = it.next + 1
  println(i)
}
```

The `Iterator` pattern is supported in all programming languages.

Iteration example

Reading from a file, first in Scala

```
val data = scala.io.Source.fromFile("/big/data").getLines
while (data.hasNext) {
  println(data.next)
}
// Equivalently...
for (line <- data) {
  println(line)
}
```

and then in Python

```
with open("/big/data","r") as data:
  # readlines() returns an object that implements __iter__()
  for line in data.readlines():
      print line
```

Observation

Observation allows us to process (almost) unbounded size data sets, where the data source controls the processing rate

```
// Consumer
trait Observer[A] {
  def onNext(a: A): Unit
  def onError(t: Throwable): Unit
  def onComplete(): Unit
}
```

```
// Producer
trait Observable[A] {
  def subscribe(obs: Observer[A]): Unit
}
```

Typical usage

```
Observable.from(1,2,3,4,5).
  map(x => x + 1).
  subscribe(x => println(x))
```

Iteration and Observation

- Iteration and Observation are *dual*
- The same set of higher-order functions can be used to process data in both cases

Iterator-based `map`. "Pulls" data out of array.

```
Array(1,2,3,4).map(x => x + 1) // Scala
```

```
map([1,2,3,4], lambda x: x + 1) # Python
```

Observation-based (reactive) `map`. Data is "pushed" to it asynchronously, when new data is available.

```
Observable.from(1,2,3,4,5).map(x => x + 1) // Scala
```

```
Observable.from_([1,2,3,4]).map(lambda x: x + 1) # Python
```

D: (How) Can we convert between the two types of enumeration?

Traversal

We apply a strategy to visit all individual items in a collection.

```
for i in [1,2,3]:
  print i

for k,v in {"x": 1, "y": 2}:
  print k
```

In case of nested data types (e.g. trees/graphs), we need to decide how to

traverse. Common strategies include:

- **Breadth-first traversal**: From any node A, visit its neighbours first, then its children.
- **Depth-first traversal**: From any node A, visit its children first, then its neighbours.

Traversal through iteration

In most programming environments, traversal is implemented by *iterators*.

```
class Tree(object):

  def __init__(self, title, children=None):
    self.title = title
    self.children = children or []

  def __iter__(self): ## Depth first
    yield self
    for child in self.children:
      for node in child:
        yield node
```

Then, we can iterate using standard language constructs

```
t = Tree("a", [Tree("b", [Tree("c"), Tree("d")])])
for node in iter(t):
  print node
```

4.4 Operations

Operations are transformations, aggregations or cross-referenceing of data stored in data types. All of container data types can be iterated.

4.5 Basic transformations

- **Conversion**: Convert values of type A to type B

  - Celcius to Kelvin
  - € to $
- **Filtering**: Only present data items that match a condition
  - All adults from a list of people
  - Remove duplicates
- **Projection**: Only present parts of each data item
  - From a list of cars, only display their brand

Our running example Suppose we have a list of people; each person is identified by a unique identifier, their age, their height in cm and their gender.

```
p = {id: 10, age: 50, height: 180, weight: 75, gender: 'Male'}
```

Now, let's create 1000 random people!

```
from random import randint, seed
seed(42)
```

```
from random import randint, seed
seed(42)

people = []
genders = ['Male', 'Female', 'Other']
for i in range(1000):
  p = {'id': i,
    'age': randint(10,80),
    'height': randint(60, 200),
    'weight': randint(40, 120),
    'gender': genders[randint(0,2)] }
  people.append(p)
```

```
## people[0]: {'gender': 'Male', 'age': 55, 'id': 0, 'weight': 62, 'height': 63}
```

Conversion

To convert values, we traverse a collection and apply a convertion function to each individual element. This is generalized to the *map* function:

$map(xs : [A], f : (A) \to B) \to [B]$

Let's convert the persons' heights to meters

```
def to_m(person):
  person['height'] = person['height'] * 1.0 / 100
  return person

people = map(lambda x: to_m(x), people)
```

```
## people[0]: {'gender': 'Male', 'age': 55, 'id': 0, 'weight': 62, 'height': 0.63}
```

Projection

Projection allows us to select parts of a Tuple, Relation or other nested data type for further processing. To implement this, we need to iterate over all items of a collection and apply a *conversion* function.

Filtering

To filter values from a list, we traverse a collection and apply a predicate function to each individual element.

$filter(xs : [A], f : (A) \to Boolean) : [A]$

Q: How can we implement filter?

. . .

```
def filter(xs, pred):
  result = []
  for i in xs:
    if pred(i):
      result.append(i)

  return result
```

Aggregation via Reduction Reductions (or *folds*) apply a *combining operator* to a traversable sequence to aggregate the individual items into a single result. Two variants exist: *left reduction* and *right reduction*.

Our task is to calculate the total weight for our people list. To do so, we need to iterate over the list and add the individual weights. First, we do it with an imperative approach:

```
total_weight = 0
for p in people:
```

```
  total_weight = total_weight + p['weight']

print total_weight
```

We notice that iteration and reduction are independent. What if we abstract iteration away?

Functional Reduction: Left

$reduceL(xs : [A], init : B, f : (acc : B, x : A) \to B) \to B$

Left reduce takes a function `f` with 2 arguments and an initial value and applies `f` on all items, starting from the **left** most. `f` combines the result of its previous application with the current element.

```
def total_weight(acc, p):  # acc is for accumulator
  return acc + p['weight']
```

Then we can calculate the total weight of all people in our collection as follows

```
total = reduce(total_weight, people, 0)
# or equivalently, using an anonymous function
total = reduce(lambda x,y: x + y['weight'], people, 0)
```

Functional Reduction: Right

$reduceR(xs : [A], init : B, f : (acc : A, x : B) \to B) \to B$

Left right takes a function `f` with 2 arguments and an initial value and applies `f` on all items, starting from the **right** most. `f` combines the result of its previous application with the current element.

```
def reduceR(func, xs, init):
  return reduce(lambda x,y: func(y,x), reversed(xs), init)
```

`reduceR` and `reduceL`: differences

To see how `reduceR` and `reduceL` are evaluated, we define a reduction function that prints the intermediate steps.

```
def reduce_pp(acc, x):
  return "(%s + %s)" % (acc, x)
```

How does `reduceL` work?

```
print reduce(reduce_pp, range(1,10), 0)
```

```
## (((((((((0 + 1) + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9)
```

How does `reduceR` work?

```
print reduceR(reduce_pp, range(1,10), 0)
```

```
## (1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + 0))))))))))
```

Q: *Can we always apply* ***`reduceL`*** *instead of* ***`reduceR`****?*

`reduceR != reduceL`

The answer to the previous question is: *it depends on whether the reduction operation is commutative.*

An op $\circ$ is commutative iff $x \circ y = y \circ x$.

We can see that if we choose a non-commutative operator, `reduceL` and `reduceR` produce different results.

```
print reduce(lambda x, y: x * 1.0 / y, range(1,10), 1)
```

```
## 2.7557319224e-06
```

```
print reduceR(lambda x, y: x * 1.0 / y, range(1,10), 1)
```

```
## 2.4609375
```

Counting elements The simplest possible aggregation is counting the number of elements, possiblibly matching a condition

$count(xs : [A], pred) : Integer$

```
def count(xs, pred):
  len(filter(xs, pred))
```

Distinct elements Given a sequence of items, produce a new sequence with no duplicates.

$distinct(xs : [A]) : [A]$

Distinct operations are usually implemented by copying the initial sequence elements to a `set` data structure.

```
a = [1,2,2,3]
set(a)
```

Distinct assumes that items have unique identities; in Python, we use the `id()` method (same as `hashCode()` in Java).

Aggregation functions Aggregation functions have the following generic signature:

$f : [A] \rightarrow Number$

Their job is to reduce sequences of elements to a single measurement. Some examples are:

- Mathematical functions: `min`, `max`, `count`
- Statistical functions: `mean`, `median`, `stdev`

Grouping

Grouping splits a sequence of items to groups given a classification function.

$groupBy(xs : [A], f : A \rightarrow K) : Map[K, [A]]$

```
def group_by(classifier, xs):
  result = dict()
  for x in xs:
    k = classifier(x)
    if k in result.keys():
        result[k].append(x)
    else:
        result[k] = [x]
  return result
```

```
def number_classifier(x):
  if x % 2 == 0:
    return "even"
  else:
    return "odd"
```

```
a = [1,2,3,4,5,6,7]
print group_by(number_classifier, a)
```

```
## {'even': [2, 4, 6], 'odd': [1, 3, 5, 7]}
```

Aggregation example

How can we get the average height per gender (adults only) in our list of people?

```
from numpy import mean

adults = filter(lambda x: x['age'] > 18, people)
adults_per_gender = group_by(lambda x: x['gender'], adults)
avg_age_per_gender = \
  map(lambda (k, v): {k, mean(map(lambda y: y['age'], v))},
      adults_per_gender.items())

print avg_age_per_gender
```

```
## [set([50.288808664259925, 'Male']), set([49.821428571428569, 'Other']), set([49.4448
```

The above is equivalent to the following SQL expression

```
select gender, mean(age)
from people
where age > 18
group by gender
```

D: What are the relative strengths and weaknesses of each representation?

4.6 Key-value pairs

KV databases / systems

KV stores is the most common form of distributed databases.

What KV systems enable us to do effectively is processing data locally (e.g. by key) before re-distributing them for further processing. Keys are

naturally used to aggregate data before distribution. They also enable (distributed) data joins.

Typical examples of distributed KV stores are Dynamo, MongoDB and Cassandra

Keys and Values

The most common data structure in big data processing is *key-value pairs*.

- A **key** is something that identifies a data record.
- A **value** is the data record. Can be a complex data structure.
- The KV pairs are usually represented as sequences

```python
[ # Python
  ['EWI': ["Mekelweg", 4]],
  ['TPM': ["Jafaalaan", 5]],
  ['AE': ["Kluyverweg", 1]]
]
```

```scala
List( // Scala
  List("EWI", Tuple2("Mekelweg", 4)),
  List("TPM", Tuple2("Jafaalaan", 5)),
  List("AE",  Tuple2("Kluyverweg", 1))
)
```

Typical operations for KV collections

`mapValues`: Transform the values part $mapVal(kv : [(K, V)], f : V \rightarrow U) : [(K, U)]$

. . .

`groupByKey`: Group the values for each key into a single sequence. $groupByKey(kv : [(K, V)]) : [(K, [V])]$

. . .

`reduceByKey`: Combine all elements mapped by the same key into one $reduceByKey(kv : [(K, V)], f : (V, V) \rightarrow V) : [(K, V)]$

. . .

`join`: Return a sequence containing all pairs of elements with matching keys
$join(kv1 : [(K, V)], kv2 : [(K, W)]) : [(K, (V, W))]$

Common operations on KVs

`mapValues`: With `mapValues` we can apply a transformation and keep data local.

```
def mapValues[U](f: (V) => U): RDD[(K, U)]
```

`reduceByKey`: Reducing values by key allows us to avoid moving data among nodes. This is because we can reduce locally and only distribute the *results* of the reduction for further aggregation. This is also why `f` does not allow us to change the reduction type.

```
def reduceByKey(f: (V, V) => V): RDD[(K, V)]
```

Joining datasets

Supose we have a dataset of addresses

```
case class Addr(k: String, street: String, num: Int)
val addr = List(
  Addr("EWI", "Mekelweg", 4),
  Addr("EWI", "Van Mourik Broekmanweg", 6),
  Addr("TPM", "Jafaalaan", 5),
  Addr("AE",  "Kluyverweg", 1)
)
```

and a dataset of deans

```
case class Dean(k: String, name: String, surname: String)
val deans = List(
  Dean("EWI", "John", "Schmitz"),
  Dean("TPM", "Hans", "Wamelink")
)
```

Q: Define a method `deanAddresses` to retrieve a list of address of each dean.

First attempt

```
def deanAddresses : List[Dean, List[Addr]] =
  deans.map {d => (d, addr.filter(a => a.k == d.k))}
```

In practice, we get the following results

```
// EEEWWWW
List(
  (Dean(EWI,John,Schmitz), List(
    Addr(EWI,Mekelweg,4),
    Addr(EWI,Van Mourik Broekmanweg,6))
  ),
  (Dean(TPM,Hans,Wamelink),List(
    Addr(TPM,Jafaalaan,5))
  )
)
```

This is OK, but a more practical result would be a `List[(Dean, Addr)]`. For this, we need to *flatten* the internal sequence.

`flatMap`: Map and flatten in one step

$flatMap(xs : [A], f : A \to [B]) : [B]$

`flatMap` enables us to combine two data collections and return a new collection with flattened values.

```
def deanAddresses2: Seq[(Dean, Addr)] =
  deans.flatMap(d =>
    addr.filter(a => a.k.==(d.k)).
      map(v => (d, v)))
```

. . .

In Scala, `flatMap` is special

```
def deanAddresses2: Seq[(Dean, Addr)] = {
  for (
    d <- deans;
    v <- addr.filter(a => a.k == d.k)
  ) yield (d, v)
}
```

KV pairs and SQL tables

A KV pair is an alternative form of a relation, indexed by a key. We can always convert between the two

```
val relation : Set[Tuple3[Int, Int, String]] = ???
```

Convert the above relation to a KV pair

```
val kvpair : Set[Tuple2[Int, Tuple2[Int, String]]] =
  relation.map(x => (x._1, (x._2, x._3)))
```

Convert the KV pair back to a relation

```
val relation2: Set[Tuple3[Int, Int, String]] =
  kvpair.map(x=> (x._1, x._2._1, x._2._2))
```

This means that any operation we can do on relations, we can also do on KV pairs.

Types of joins

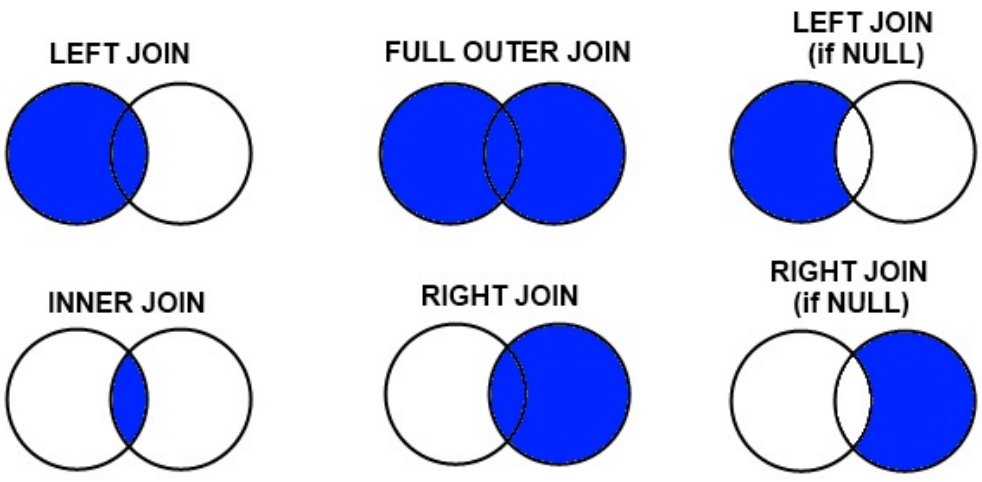


Figure 4.3: Types of joins

4.7 Immutability

Big data is immutable

One of the key characteristics of data processing is that data is never modified in place. Instead, we apply operations that create new versions of the data, without modifying the original version.

Immutability is a general concept that expands in much of the data processing stack.

Immutability accross the stack

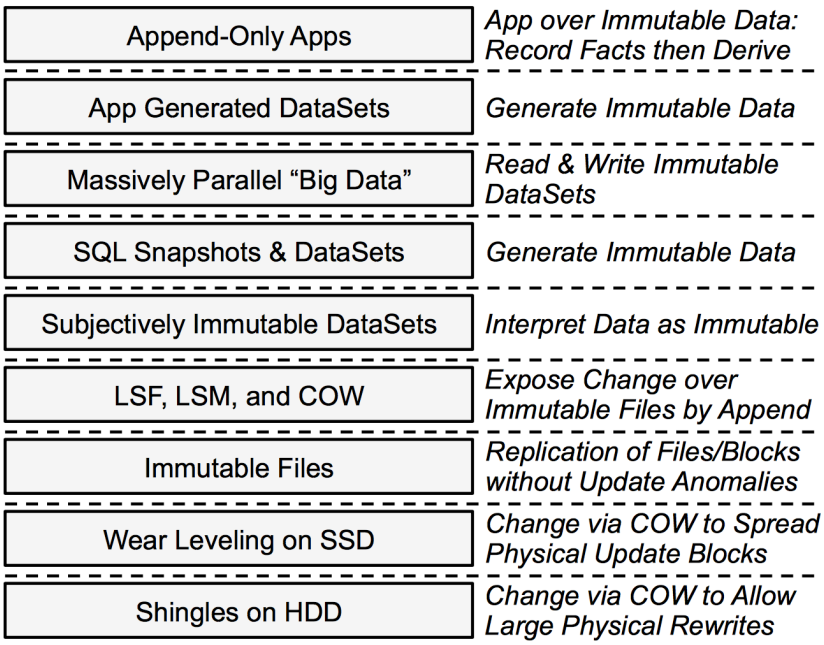


Figure 4.4: Immutability

Image from Helland in ref [Helland, 2015]

Copy-On-Write (COW)

Copy-On-Write is a general technique that allows us to share memory for read-only access accross processes and deal with writes only if/when they are performed by copying the modified resource in a private version.

COW is the basis for many operating system mechanisms, such as process creation (forking), while many new filesystems (e.g. BTRFS, ZFS) use it as their storage format.

COW enables systems with multiple readers and few writers to efficiently share resources.

Immutable data structures

Immutable or *persistent* data structures always preserve the previous version of themselves when they are modified [Okasaki, 1999].

With immutable data structures, we can:

- Avoid locking while processing them, so we can process items in parallel
- Maintain and share old versions of data
- Seamlesly persist the data on disk
- Reason about changes in the data

They come at a cost of increased memory usage (data is never deleted).

Scala has both mutable and immutable versions of many common data structures. If in doubt, use immutable.

Example: immutable tree

```
val xs = TreeSet[Char]('a', 'b',
                       'c', 'd',
                       'g', 'f',
                       'h')
```

`xs` is looks like a regular mutable tree here. We can share it between 2 threads, , without worrying about updates.

The difference of immutability is apparent only when we try to insert a new node `e`.

```
val ys = xs ++ TreeSet[Char]('e')
```

Data structures in Scala

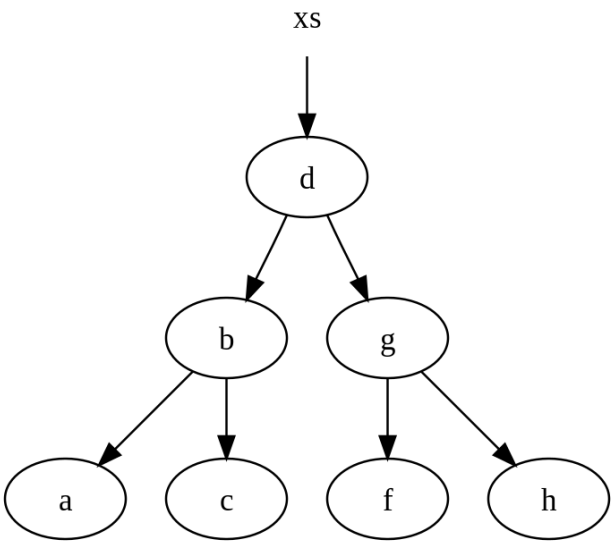


Figure 4.5: A tree

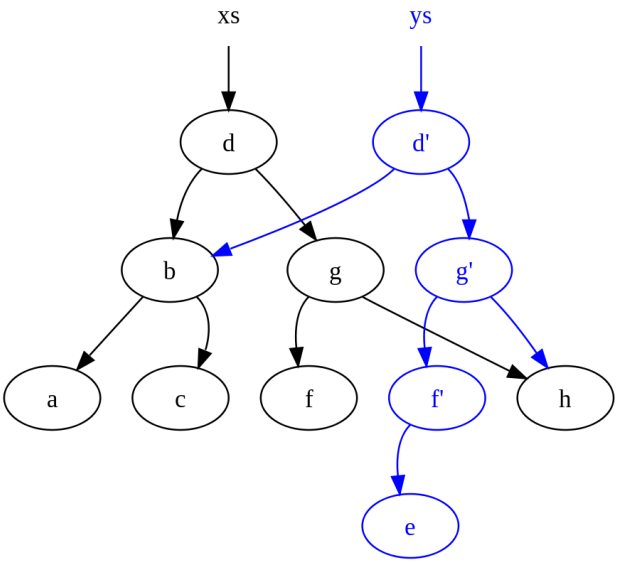


Figure 4.6: The tree after addition

ADT	`collection.mutable`	`collection.immutable`
Array	ArrayBuffer	Vector
List	LinkedList	List
Map	HashMap	HashMap
Set	HashSet	HashSet
Queue	SynchronizedQueue	Queue
Tree	—	TreeSet

Image credits

- Transformation image
- Hashtable image
- Join types

Chapter 5

Introduction to distributed systems

What is a distributed system?

According to Wikipedia: *A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages.*

Parallel and distributed systems

- **Parallel systems** use shared memory
  - *Distributed* parallel systems still use the notion of shared memory, but this is co-ordinated with special HW/SW that unifies memory accesses accross multiple computers

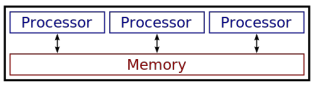


Figure 5.1: Parallel system

- **Distributed systems** use no shared components

Distributed system characteristics

- Computational entities each with own memory
  - Need to synchronize distributed state

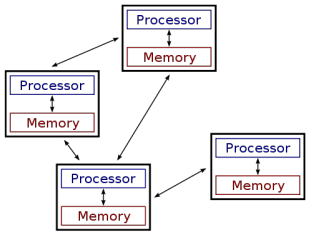


Figure 5.2: Distributed system

- Entities communicate with message passing
- Each entity maintains parts of the complete picture
- Need to tolerate failure

Building distributed systems is hard

- They fail often (and failure is difficult to spot!)
  - Split-brain scenarios
- Maintaining order/consistency is hard
- Coordination is hard
- Partial operation must be possible
- Testing is hard
- Profiling is hard: "it's slow" might be due to 1000s of factors

Fallacies of distributed systems

By Peter Deutsch

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology does not change
- Transport cost is zero
- The network is homogeneous

Four main problems

We will focus on the following four issues with distributed systems

- **Unreliable networks:** As distributed systems need to share data, they have to co-ordinate over unreliable networks.

- **Unreliable time:** Time is a universal co-ordination principle. Distributed systems need time to determine *order*
- **No single source of truth:** Distributed systems need to co-ordinate and agree upon a (version of) truth.
- **Different opinions:** Distributed systems need to provide guarantees of consistency in query answers.

5.1 Unreliable networks

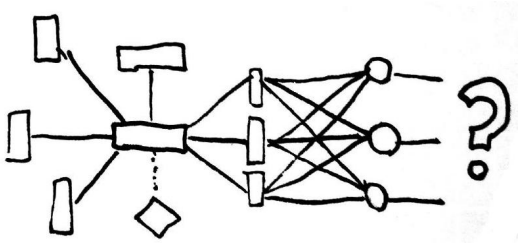


Figure 5.3: Unreliable networks

How can a network fail?

Most distributed systems use some form of asynchronous networking to communicate. Upon waiting for a response to a requests, it is not possible to distinguish whether:

1. the request was lost
2. the remote node is down
3. the response was lost

The usual remedy is to set *timeouts* and retry the request until it succeeds.

Network failures in practice

In a study of network failures by Microsoft Research[Gill et al., 2011], they found that:

- 5 devices per day fail
- 41 links per day fail
- Load balancers fail with a probability >20% once per year

- MMTR 5 mins
- Redundancy is not very effective
- Most failures are due to misconfiguration

The data is for a professionally managed data centre by a single company.

On the public cloud, failures may affect thousands of systems in parallel.

Timeouts

Timeouts is a fundamental design choice in asynchronous networks: Ethernet, TCP and most application protocols work with timeouts.

The problem with timeouts is that *delays in asynchronous systems are unbounded*. This can be due to:

- Queueing of packets at the network level, due to high or spiking traffic
- Queueing of requests at the application level, e.g. because the application is busy processing other requests

Queues also experience a snowball effect; queues are getting bigger on busy systems.

Timeouts usually follow the *exponential back-off* rule; we double the time we check for an answer up to an upper bound. More fine-grained approaches use successful request response times to calibrate appropriate timeouts.

5.2 Unreliable time

Figure 5.4: Soft Watch At The Moment Of First Explosion, by Salvador Dali

Time is essential In a distributed system, time is the only global constant nodes can rely on to make distributed decisions on ordering problems.

Time in computers is kept in two ways:

- "Real" time clocks (RTCs): Capture our intuition about time and are kept in sync with the NTP protocol with centralised servers. (e.g. `System.getCurrentTimeMillis`).
- Monotonic clocks: Those clocks only move forward. (e.g. `System.nanoTime`)

Q: Which clock would you use to time a task? Why?

The trouble with computer clocks

Monotonic clocks are maintained by the OS and rely on HW counters exposed by CPUs. They are (usually!) good for determining order within a node, but each node only has its own notion of time.

NTP can synchronize time across nodes with an accuracy of *ms*. A modern CPU can execute 10^6 instructions ($\times$ number of cores) in an ms!

Moreover, *leap seconds* are introduced every now and then; minutes may last for 61 or 59 seconds on occasion

μs accuracy is possible with GPS clocks, but expensive

Order

What is order? A way of arranging items in a way that the following properties are maintained.

- **Reflexivity**: $\forall a.a \leq a$
- **Transitivity**: if $a \leq b$ and $b \leq c$ then $a \leq c$
- **Antisymetry**: $a \leq b$ and $b \leq a <=> b = a$

When do we need to order?

- Sequencing items in memory (e.g. with a mutex)
- Encoding history ("happens before" relationships)
- Mutual exclusion of access to a shared resource
- Transactions in a database

Order and time

- FIFO is enough to maintain order with a single sender

- Time at the receiver end is enough to maintain order at the receiver end
- When multiple senders/receivers are involved, we need external ordering scheme
  - **Total order:** If our message rate is *globally* bounded (e.g. 1 msg/sec/receiver), and less fine-grained than our clock accuracy (e.g. ms range), then synchronized RTCs are enough to guarantee order.
  - **Causal order:** Otherwise, we need to rely on *happens before* ($\rightarrow$) relationships.

Encoding *happens before*

Lamport introduced the eponymus timestamps in 1978[Lamport, 1978]. Lamport timestamps define a **partial causal ordering** of events.

Invariant: For two events a and b, if $a \rightarrow b$, then $LT(a) < LT(b)$.

Q: The reverse is **not** true: If $LT(a) < LT(b)$, then **it does not mean** that $a \rightarrow b$!! Why?

Lamport timestamps: How they work

- Each individual process p maintains a counter: $LT(p)$.
- When a process p performs an action, it increments $LT(p)$.
- When a process p sends a message, it includes $LT(p)$ in the message.
- When a process p receives a message from a process q, that message includes the value of $LT(q)$; p updates its $LT(p)$ to the $\max(LT(p), LT(q)) + 1$

Why is the LT invariant not symmetric? 4 nodes exchange events.

Initial state of timestamps: $[A(0), B(0), C(0), D(0)]$

E1. A sends to C: $[A(1), B(0), C(0), D(0)]$

. . .

E2. C receives from A: $[A(1), B(0), C(2), D(0)]$

. . .

E3. C sends to A: $[A(1), B(0), C(3), D(0)]$

. . .

E4. A receives to C: $[A(4), B(0), C(3), D(0)]$

. . .

E5. B receives to D: $[A(4), B(1), C(3), D(0)]$

. . .

E6. D receives from B: $[A(4), B(1), C(3), D(2)]$

At this point, $LT(E6) < LT(E4)$, but it does not mean that $E6 \rightarrow E4$! Events 4 and 6 are independent.

Vector clocks Vector clocks[Mattern, 1988] can maintain **total causal order**.

On a system with N nodes, each node i maintains a vector V_i of size N.

- $V_i[i]$ is the number of events tht occurred at node i
- $V_i[j]$ is the number of events that node i knows occurred at node j

They are updated as follows:

- Local events increment $V_i[i]$
- When i sends a message to j, it includes V_i
- When j receives V_i, it updates all elements of V_j to $V_j[a] = \max(V_i[a], V_j[a])$

Vector clocks guarantees

- if $a \rightarrow b$, then $VC(a) < VC(b)$
- if $VC(a) < VC(b)$, then $a \rightarrow b$
- if $VC(a) < VC(b)$ and VC(b) < VC(c) then $a \rightarrow c$
- if $VC(a) < VC(b)$, then $RT(a) < RT(b)$, where RT is the clock time of events a and b

Vector clocks are expensive to maintain: they require $O(n)$ timestamps to be exchanged with each communication.

However, it has been shown[Charron-Bost, 1991] that we cannot do better than vector clocks!

5.3 Distributed decision making

What is true in a distributed setting?

- Nodes in distributed systems cannot know anything for sure
  - Only make guesses
- Individual nodes cannot rely on their own information
  - Clocks can be unsynchronized
  - Other nodes may be unresponsive when updating state
- "Split-brain" scenarios: Parts of the system know a different version of the truth than the other part(s)

In distributed settings, the truth is determined by *concensus*, which is reached through *voting* mechanisms.

Consensus Achieving a decision in the presence of faulty processes.

- Committing a transaction
- Synchronizing state machines
- Leader election
- Atomic broadcasts

The 2 generals problem

- 2 armies camped in opposing hills (A1 and A2)
- The are only able to communicate with messengers

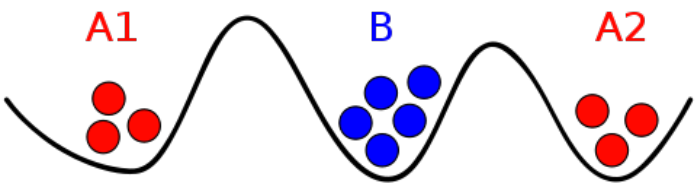


Figure 5.5: The two generals problem setting

- They need to decided on a time to attack
- Enemy (B) is camped between the two hills and can at any time intercept the messengers
- How can the generals decide when to attack?

. . .

It turns out that it is impossible to make a reliable decision.

The 2 generals problem solution

- The problem cannot be solved without loss of information
- Approximately:
  - Pre-agree on timeouts
  - Send n labeled messages
  - Receiver calculates received messages within time window, then decides how many messages to send for ack.

Consequences: we can only make distributed decisions using either reliable communication or more than 2 parties.

The Byzantine generals problem Formulated by Lamport et al.[Lamport et al., 1982], the Byzantine generals problem shaped distributed systems research for the next 40 years.

The formulation is simple:

- n generals need to make unanimous decision
- they communicate with unreliable messages
- m ($n > m$) generals vote suboptimally (i.e. lie) or do not vote

Consensus algorithms

Figure 5.6: Byzantine Eagle

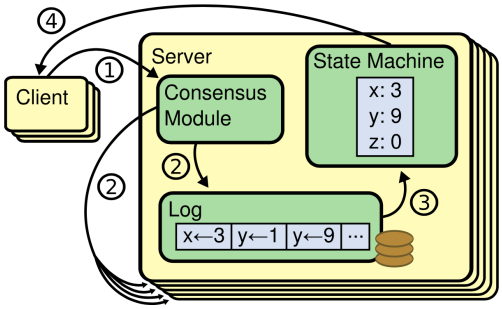


Figure 5.7: Roles in a Raft cluster

Most consensus algorithms (e.g. Paxos[Lamport, 1998] or Raft[Ongaro and Ousterhout, 2014]), attempt to keep the log module of *replicated state machines* in sync. Basic properties include:

- **Safety:** Never returning an incorrect result, in the presence of *non*-Byzantine conditions.
- **Availability:** Able to provide an answer if $n/2 + 1$ servers are operational
- **No clocks:** They do not depend on RTCs to work
- **Immune to stranglers:** If $n/2 + 1$ servers vote, then the result is considered safe

The Raft consensus algorithm

Raft defines the following server roles:

- Client: Creates log entries, asks queries
- Leader: Accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries

- Followers: Replicate the leader's state machine.

Raft cluster states

1. Leader election
   - Select one server to act as leader
   - Detect crashes, choose new leader
2. Log replication (normal operation)
   - Leader accepts commands from clients, appends to its log
   - Leader replicates its log to other servers (overwrites inconsistencies)

Raft ensures that: logs are always consistent and that only servers with up-to-date logs can become leader

Raft terms

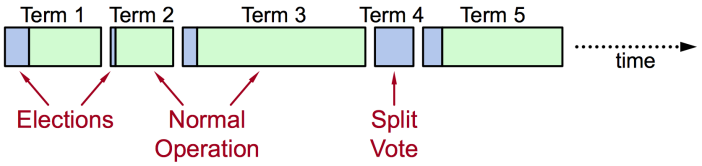


Figure 5.8: Raft terms

- One leader per term only
- Some terms have no leader (failed election)
- Each server maintains current term value (no global view)
  - Exchanged in every RPC
  - Peer has later term? Update term, revert to follower

Terms are used to identify obsolete information

Leader election process

- Raft elects only one leader per election
- Raft ensures that one candidate will eventually win

See more details in this visualization

The FLP impossibility Fischer, Linch and Patterson (FLP) theorem[Fischer et al., 1985]: *In an asynchronous network, consensus cannot be reached if at least one node fails in asynchronous networks*

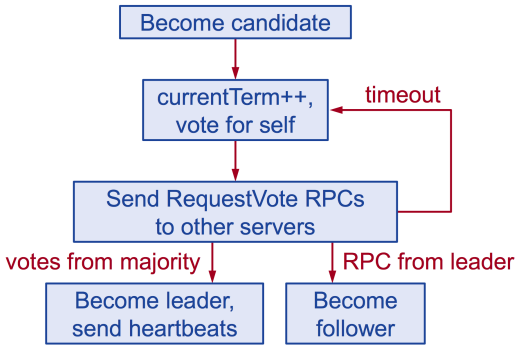


Figure 5.9: Raft leader election

A foundational result, proving the impossibility of distributed consensus. The system model the authors assume is fairly restrictive:

- Asynchronous communication
- No clocks or timeouts
- No random number generators

In practice however, we can mitigate the consequences, as we are indeed allowed to use both clocks and/or random numbers.

Byzantine fault tolerance

Raft and Paxos work by assuming that the exchanged messages are valid and true (i.e. non-Byzantine). In open distributed systems (e.g. BitCoin) this assumption is not necessarily valid.

Most open distributed systems use public-key cryptography and node registration before they start and sign messages to avoid MITM attacks.

Still, decisions require majority votes, so the $n/2 + 1$ rule applies.

5.4 Consistency

The consistency guarantee Consistency refers to the requirement that any given transaction must change affected data only in allowed ways.

- **strong**: at any time, concurrent reads from any node return the same values
- **eventual**: if writes stop, all reads will return the same value after a while

Consensus is the basis upon which we build consistency

The CAP conjecture

By Erik Brewer[Brewer, 2012]: A distributed system can only provide 2 of the following 3 guarantees

- **Consistency**: all nodes see the same data at the same time
- **Availability**: every request receives a response about whether it succeeded or failed
- **Partition tolerance**: the system continues to operate despite arbitrary partitioning due to network failures

While widely cited, it is only indicative; when the network is working, systems can offer all 3 guarantees. So it can be reduced to *either consistent or available when partitioned*.

Linearisability

At any time, concurrent reads from any node return the same values. As soon as writes complete successfully, the result is *immediately* replicated to all nodes in an *atomic* manner an is made available to reads. In that sense, linearisability is a *timing constraint*[Herlihy and Wing, 1990].

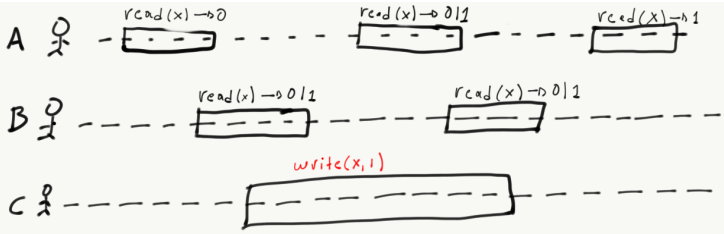


Figure 5.10: A non-linearisable system

Q: Is the above system linearisable?

. . .

It is not, as while the write operation is in flight, the system cannot return a consistent answer.

Linearisability primitives

All operations last for a time block called a transaction; this involves setting up a connection to a remote system and executing the command.

- *reads*: must always return the latest value from the storage.
- *writes*: change the value of the shared memory; last-one-wins is the most common strategy to deal with multiple writers
- *compare and swap:* or atomic writes. *TODO: Fix this* It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. All further writes fail.

Linearisability example

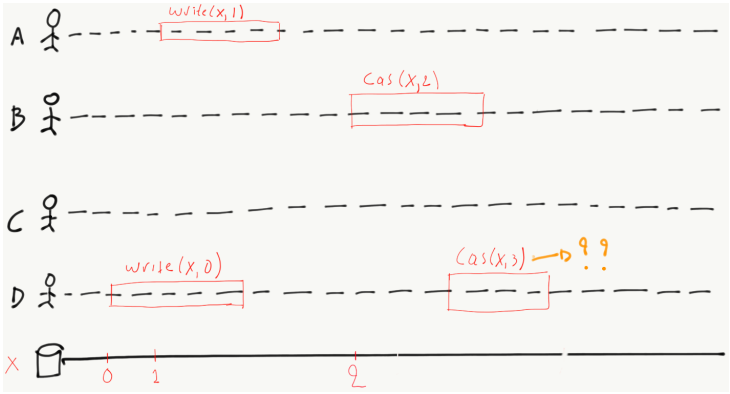


Figure 5.11: Linearisability Compare and Swap

- TODO: 2 must be set to 2 at the end of the cas command*

Q: What does the $cas(x, 3)$ instruction do?

. . .

Return an error! Compare and swap instructions should run atomically across the cluster.

Q: What should $read(x)$ return for client B?

. . .

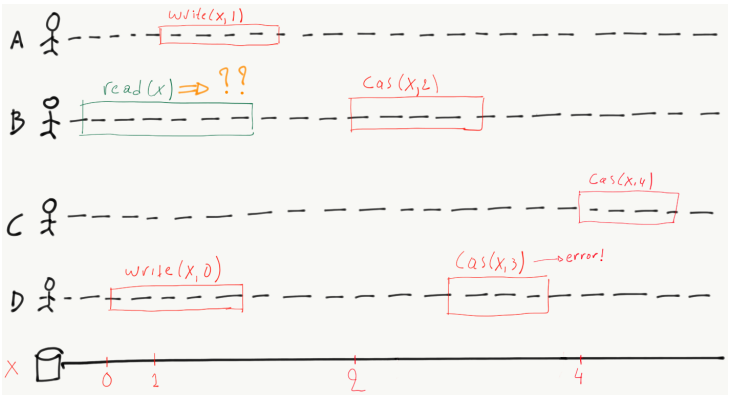


Figure 5.12: Linearisability Read after Write

It should return 1. We assume that write transactions get committed imediately when they are executed, so A's write overwrites D's earlier write.

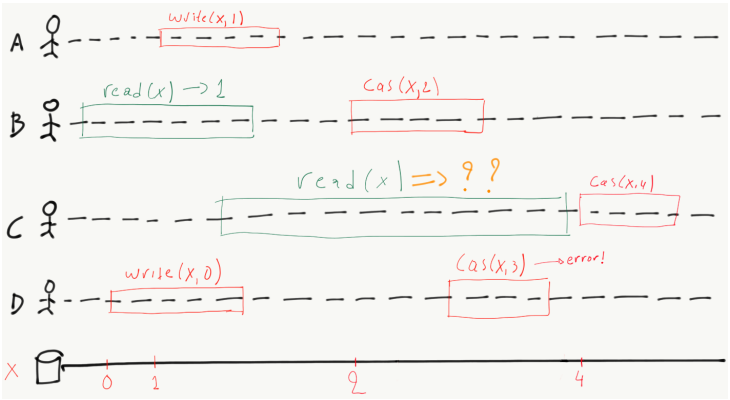
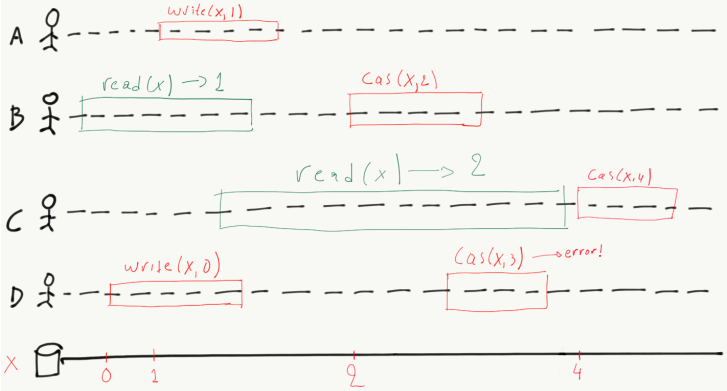


Figure 5.13: Linearisability read during conflict

Q: What should $read(x)$ return for client C?

. . .

It should return 2. The compare-and-swap transaction started by B may conflict with the one started by D, but the system always maintains a consistent state.



General advice

- Avoid implementing a distributed system
- Avoid relying on a distributed system
- If the above fail, only use available solutions

Content credits

- Distributed systems, by Miym
- Lamport clock example
- 2 generals problem picture, by Jens Erat
- Images in Raft explanation, by Diego Ongaro
- Linearisability examples are from Kleppmann[Kleppmann, 2017]

Chapter 6

Distributed databases

Distributed databases in a nutshell

. . .

Figure 6.1: Hey I just met you, the network's laggy

. . .

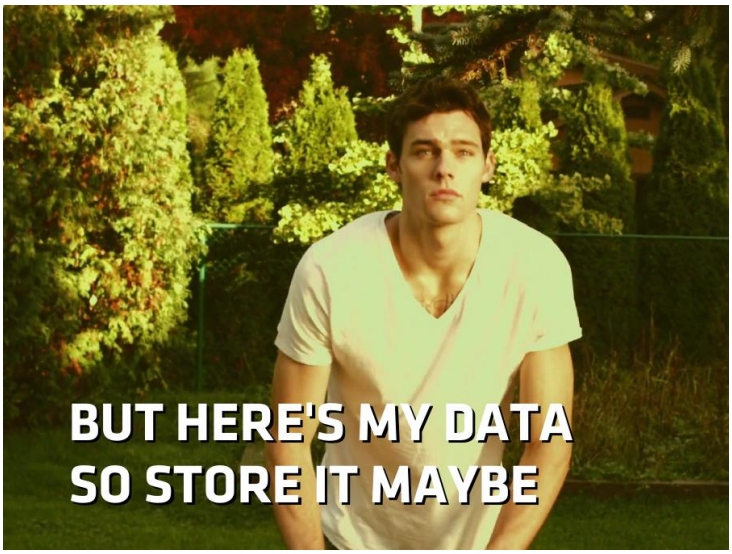


Figure 6.2: but here's my data, so store it maybe

. . .

A distributed database is a distributed system designed to provide read/write access to data, using the relational or other format.

Splitting the data among nodes

We will examine 3 topics in distributed databases:

- **Replication:** Keep a copy of the same data on several different nodes.
- **Partitioning:** Split the database into smaller subsets and distributed the partitions to different nodes.
- **Transactions:** Mechanisms to ensure that data is kept consistent in the database

Q: Usually, distributed databases employ both replication and partitioning at the same time. Why?

6.1 Replication

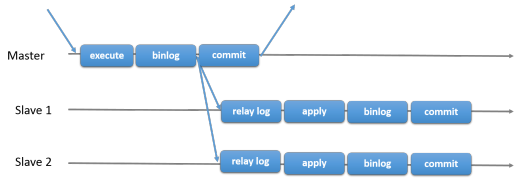


Figure 6.3: MySQL async replication

Why replicate?

With replication, we keep identical copies of the data on different nodes.

D: Why do we need to replicate data?

. . .

- To allow the system to work, even if parts of it are down
- To have the data (geographically) close to the data clients
- To increase read throughput, by allowing more machines to serve read-only requests

Replication Architectures

In a replicated system, we have two node roles:

- **Leaders** or **Masters**: Nodes that accept writes from clients
- **Followers**, **Slaves** or **replicas**: Nodes that provide read-only access to data

Depending on how replication is configured, we can see the following architectures

- **Single leader** or **master-slave**: A single master accepts writes, which are distributed to slaves
- **Multi leader** or **master-master**: Multiple masters accept writes, keep themselves in sync, then update slaves
- **Leaderless replication** All nodes are peers in the replication network

How does replication work?

The general idea in replicated systems is that when a *write* occurs in node, this write is distributed to other nodes in either of the two following modes:

- **Synchronously:** Writes need to be confirmed by a configurable number of slaves before the master reports success.
- **Asynchronously:** The master reports success immediately after a write was committed to its own disk; slaves apply the changes in their own pace.

D: What are the advantages/disadvantages of each replication type?

Statement based replication in databases

The master ships all write statements to the slaves, e.g. all `INSERT` or `UPDATE` in tact. However, this is problematic:

```
UPDATE foo
SET updated_at=NOW()
WHERE id = 10
```

D: What is the issue with the code above in a replicated context?

. . .

Statement based replication is non-deterministic and mostly abandoned.

Write-ahead log based replication

Most databases write their data to data structures, such as B^+-trees. However, before actually modifying the data structure, they write the intended change to an append-only *write-ahead log* (WAL).

WAL-based replication writes all changes to the master WAL also to the slaves. The slaves apply the WAL entries to get a consistent data.

The main problem with WAL-based replication is that it is bound to the implementation of the underlying data structure; if this changes in the master, the slaves stop working.

Postgres and Oracle use WAL-based replication.

Logical-based replication

The database generates a stream of *logical* updates for each update to the WAL. Logical updates can be:

- For new records, the values that where inserted
- For deleted records, their unique id
- For updated records, their id and the updated values

MongoDB and MySQL use this replication mechanism.

Process for creating a replica

1. Take a consistent snapshot from the master
2. Ship it to the replica
3. Get an `id` to the state of the master's replication log at the time the snapshot was created
4. Initialize the replication function to the latest master `id`
5. The replica must retrieve and apply the replication log until it catches up with the master

A real example: MySQL

Master

```
> SHOW MASTER STATUS;
+--------------------+----------+
| File               | Position |
+--------------------+----------+
| mariadb-bin.004252 | 30591477 |
+--------------------+----------+
1 row in set (0.00 sec)
```

Slave

```
>CHANGE MASTER TO
  MASTER_HOST='10.0.0.7',
  MASTER_USER='replicator',
  MASTER_PORT=3306,
  MASTER_CONNECT_RETRY=20,
  MASTER_LOG_FILE='mariadb-bin.452',
  MASTER_LOG_POS= 30591477;
```

```
> SHOW SLAVE STATUS\G

Master_Host: 10.0.0.7
Master_User: replicator
Master_Port: 3306
Master_Log_File: mariadb-bin.452
Read_Master_Log_Pos: 34791477
Relay_Log_File: relay-bin.000032
Relay_Log_Pos: 1332
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
```

Complications with async replication

- **Read-after-write:** A client must always be able to read their own writes from any part of the system. Practically, the client can:
  - Read recent writes from the master
  - Read session data from the master
- **(non-) Monotonic reads:** When reading from multiple replicas concurrently, a stale replica might not return records that the client read from an up to date one.
- **Causality violations:** Violations of the *happened-before* property.

Async replication is fundamentally a *consensus problem*.

Multi-leader replication

The biggest problem with multi-leader replication are *write conflicts*. To demonstrate this, let's think in terms of `git`:

```
                                # Clock
# User 1
git clone git://....            # t+1
git add foo.c                   # t+2
##hack hack hack
git commit -a -m 'Hacked v1'    # t+3
git push                        # t+5
```

```
# User 2
git clone git://....            # t+2
git add foo.c                   # t+5
##hack hack hack
git commit -m 'Hacked new file' # t+6
git push # fails                # t+7
git pull # CONFLICT             # t+7
```

If we replace *user* with *master node*, we have exactly the same problem

How to avoid write conflicts?

- **One master per session** If session writes do not interfere (e.g., data are only stored per user), this will avoid issues altogether.
- **Converge to consistent state** Apply a *last-write-wins* policy, ordering writes by timestamp (may loose data) or report conflict to the application and let it resolve it (same as `git` or Google docs)
- **Use version vectors** Modelled after version clocks, they encode happens before relationships at an object level.

6.2 Partitioning

Why partitioning?

With partitioning, each host contains a fraction of the whole dataset.

The main reason is *scalability*:

- Queries can be run in parallel, on parts of the dataset
- Reads and writes are spread on multiple machines

Partitioning is always combined with replication. The reason is that with partitioning, a node failure will result in irreversible data corruption.

How to partition?

The following 3 strategies are

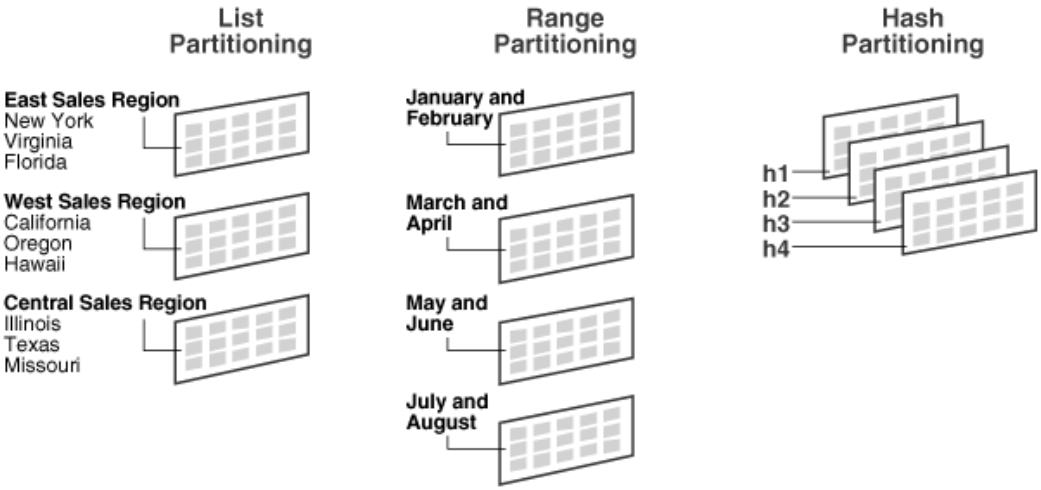


Figure 6.4: Types of partitioning

- **Range partitioning** Takes into account the natural order of keys to split the dataset in the required number of partitions. Requires keys to be naturally ordered and keys to be equally distributed across the value range.
- **Hash partitioning** Calculates a hash over the each item key and then produces the modulo of this hash to determine the new partition.
- **Custom partitioning** Exploits locality or uniqueness properties of the data to calculate the appropriate partition to store the data to. An example would be pre-hashed data (e.g. `git` commits) or location specific data (e.g. all records from Europe).

Request routing

On partitioned datasets, clients need to be aware of the partitioning scheme in order to direct queries and writes to the appropriate nodes.

To hide partitioning details from the client, most partitioned systems feature a *query router* component siting between the client and the partitions.

The query router knows the employed partitioning scheme and directs requests to the appropriate partitions.

Partition and replication example: MongoDB

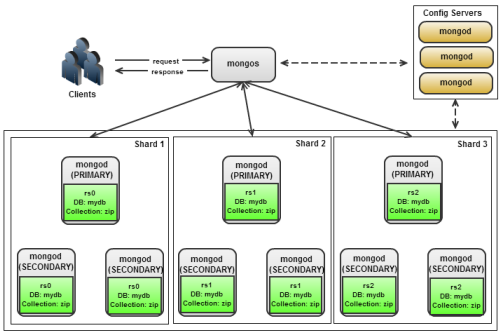


Figure 6.5: Sharding and replication in MongoDB

6.3 Transactions

Many clients on one database

What could potentially go wrong?

- Many clients try to update the same data store at the same time, when….

. . .

- the network fails, and then…

. . .

- the database master server cannot reach its network-mounted disk, so…

. . .

- the database tries to fail over to a slave, but it is unreachable, and then…

. . .

- the application writes timeout.

What is the state of the data after this scenario?

As programmers, we are mostly concerned about the code's *happy path*. Systems use **transactions** to guard against catastrophic scenarios.

What are transactions?

Transactions[Gray, 1981] are blocks of operations (reads and writes), that either succeed or fail, as a whole.

Transactions are defined in the context of an *application* that wants to modify some data and a *system* that guards data consistency.

- The application initiates a transaction, and when done, it *commits* it.
- If the system deems the transaction successful, it guarantees that the modifications are safely stored
- If the system deems the transaction unsuccessful, the transaction is *rollbacked*; the application can safely retry.

Transactions in relational databases

The concept of the transaction started with the first SQL database, System R; while the mechanics changed, the semantics are stable for the last 40 years.

Transactions provide the following guarantees[Haerder and Reuter, 1983]:

- **A**tomicity: The transaction either succeeds or fails; in case of failure, outstanding writes are ignored
- **C**onsistency: any transaction will bring the database from one valid state to another
- **I**solation: Concurrent execution of transactions results in a system state that would be obtained if transactions were executed sequentially
- **D**urability: once a transaction has been committed, it will remain so

Isolation Isolation guards data consistency in the face of concurrency.

Databases try to hide concurrency issues from applications by providing *transaction isolation*.

Serializability By default, isolation entails executing transactions as if they were serially executed. To implement serializability, databases:

- Only execute in a single core (e.g. Redis)
- Use **2 phase locking**: all database objects (e.g. rows) have a shared lock and an exclusive lock
  - All readers acquire a shared lock
  - A writer needs to acquire the exclusive lock
  - Deadlocks can happen when a transaction A waits for B to release their exclusive locks and vice versa

- Use MVCC and Copy-on-Write data structures

Multi-Version Concurency Control

With MVCC, the database works like Git:

- Each transaction A sees the most recent copy of the data (`git checkout -b A`)
- When A commits (`git commit -a`):
  - If A touched no object that was updated before by another transaction the database will create a new version (`git checkout master; git merge A`)
  - If A changed an object that was updated before, the database with report a conflict
- The current state is the result of the application of all intermediate transactions on an initial state
- Occasionally, we need to clean up ignored intermediate states (`git gc`)

Weaker isolation levels While serialiazable isolation works it may be slow. Consequently, historically, databases made compromises in how they implement isolation.

- *Dirty reads*: A transaction reads data written by a concurrent uncommitted transaction
- *Non Repeatable Reads*: A transaction re-reads data it has previously read and finds that data modified
- *Phantom reads*: Results to queries change due to other transactions being committed

Isolation	DR	NRR	PR
Read uncommitted	X	X	X
Read committed		X	X
Repeatable read			X
Serializable			

6.4 Distributed transactions

Atomic commits

In a distributed database, a transaction spanning multiple nodes must either succeed on all nodes or fail (to maintain atomicity).

Transactions may also span multiple systems; for example, we may try to remove a record from a database and add it to a queue service in an atomic way.

In contrast to the distributed systems consensus problem, all nodes must agree on whether a transaction has been successfully committed.

The most common mechanism used to dead with distributed atomic commits is the *two-phase commit* (2PC) protocol.

2-phase commits

In 2PC, we find the following roles:

- A coordinator or transaction manager
- Participants or cohorts

The 2PC protocol makes the following assumptions:

- There is stable, reliable storage on the cohorts
- No participant will crash forever
- Participants use (indestructible) write-ahead log
- Any two nodes can communicate with each other

A transaction in 2PC

1. A client starts a 2PC transaction by acquiring a globally unique transaction id (GTID)
2. The client attaches the GTID to each transaction it begins with individual nodes
3. When the client wants to commit, the coordinator sends a *PREPARE* message to all nodes
4. If at least one node replies with 'NO', the transaction is aborted
5. If all nodes reply with 'YES', the coordinator writes the decision to disk and tries forever to commit individual transactions to the cohort

D: What problems do you see with 2PC?

Problems with 2PC

- *Holding locks in nodes*: While a decision is being made, faster nodes must lock objects; consequently, a slow or crashing node will kill the performance of the whole system
- *Exaclty-once semantics*: All nodes participating in a distributed transaction must support transactions to guarantee exactly once semantics
- *Co-ordinator failures*: Even though the co-ordinator must be able to resist corruption in case of crashes, this is not always the case and semi-aborted transactions affect performance due to locking

Image credits

- Jespen, by K. Kingsbury
- MySQL async replication, MySQL documentation
- Database partitioning from the Oracle 11g documentation
- MongoDB sharding by StackExchange user Jerry

Chapter 7

Processing data with Spark

7.1 The world before Spark

Parallelism

Parallelism is about speeding up computations by utilising clusters of machines.

- **Task parallelism**: Distribute computations across different processors
- **Data parallelism**: Apply the same computation on different data sets

Distributed data parallelism involves splitting the data over several distributed nodes, where nodes work in parallel, and combine the individual results to come up with a final one.

Issues with data parallelism

- Latency: Operations are 1.000x (disk) or 1.000.000x (network) slower than accessing data in memory
- (Partial) failures: Computations on 100s of machines may fail at any time

This means that our programming model and execution should hide (but not forget!) those.

Hadoop: Pros and Cons

What is good about Hadoop: *Fault tolerance*

Hadoop was the first framework to enable computations to run on 1000s of commodity computers.

What is wrong: *Performance*

- Before each Map or Reduce step, Hadoop writes to HDFS
- Hard to express iterative problems in M/R
  - All machine learning problems are iterative

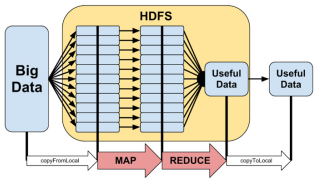


Figure 7.1: The Hadoop Workflow

DryadLINQ

Microsoft's DryadLINQ combined the Dryad distributed execution engine with the LINQ language for defining queries.

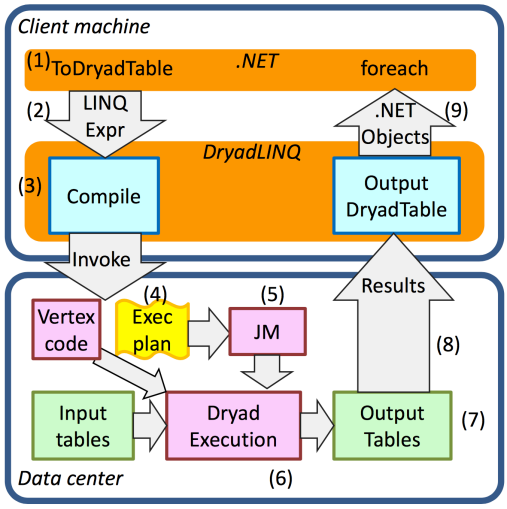


Figure 7.2: DryadLINQ architecture

FlumeJava

Google's FlumeJava attempted to provide a few simple abstractions for programming data-parallel computations. These abstractions are higher-level than those provided by MapReduce, and provide better support for pipelines.

```
PTable<String,Integer> wordsWithOnes =
  words.parallelDo(
      new DoFn<String, Pair<String,Integer>>() {
    void process(String word,
                 EmitFn<Pair<String,Integer>> emitFn) {
      emitFn.emit(Pair.of(word, 1));
    }
  }, tableOf(strings(), ints()));

PTable<String,Collection<Integer>>
  groupedWordsWithOnes = wordsWithOnes.groupByKey();

PTable<String,Integer> wordCounts =
  groupedWordsWithOnes.combineValues(SUM_INTS);
```

7.2 Spark and RDDs

What is Spark?

Spark is on open source cluster computing framework.

- automates distribution of data and computations on a cluster of computers
- provides a fault-tolerant abstraction to distributed datasets
- is based on functional programming primitives
- provides two abstractions to data, list-like (RDDs) and table-like (Datasets)

Figure 7.3: Spark Logo

Resilient Distributed Datasets (RDDs)

RDDs are the core abstraction that Spark uses.

RDDs make datasets distributed over a cluster of machines look like a Scala collection. RDDs:

- are immutable
- reside (mostly) in memory
- are transparently distributed
- feature all FP programming primitives
  - in addition, more to minimize shuffling

In practice, `RDD[A]` works like Scala's `List[A]`

Counting words with Spark

In Scala

```
// http://classics.mit.edu/Homer/odyssey.mb.txt
val rdd = sc.textFile("./datasets/odyssey.mb.txt")
rdd.
  flatMap(l => l.split(" ")).       // Split file in words
  map(x => (x, 1)).                 // Create key,1 pairs
  reduceByKey((acc, x) => acc + x). // Count occurences of same pairs
  sortBy(x => x._2, false).         // Sort by number of occurences
  take(50).                         // Take the first 50 results
  foreach(println)
```

In Python

```
text_file = sc.textFile("odyssey.mb.txt")
counts = text_file.flatMap(lambda line: line.split(" ")). \
             map(lambda word: (word, 1)). \
             reduceByKey(lambda a, b: a + b)
```

Wait, what? Python?

Yes. RDDs are implemented in Scala, Java, Python and R.

Spark itself is implemented in Scala, internally using the Akka actor framework to handle distributed state and Netty to handle networking.

For Python, Spark uses Py4J, which allows Python programs to access Java objects in a remote JVM. The PySpark API is designed to do most computations in the remote JVM; if processing needs to happen in Python, data must be copied; this incurs a performance penalty.

How to create an RDD?

RDDs can only be created in the following 3 ways

1. Reading data from external sources

```
val rdd1 = sc.textFile("hdfs://...")
val rdd2 = sc.textFile("file://odyssey.txt")
val rdd3 = sc.textFile("s3://...")
```

. . .

2. Convert a local memory dataset to a distributed one

```
val xs: Range[Int] = Range(1, 10000)
val rdd: RDD[Int] = sc.parallelize(xs)
```

. . .

3. Transform an existing RDD

```
rdd.map(x => x.toString) //returns an RDD[String]
```

RDDs are lazy!

There are two types of operations we can do on an RDD:

- **Transformation:** Applying a function that returns a new RDD. They are *lazy*.
- **Action:** Request the computation of a result. They are *eager*.

```
// This just sets up the pipeline
val result = rdd.
  flatMap(l => l.split(" ")).
  map(x => (x, 1))

// Side-effect, triggers computation
result.foreach(println)
```

Common transformations on RDDs

`map`: Apply f on all items in the RDD and return an RDD of the result. $RDD[A].map(f : A \to B) : RDD[B]$

. . .

`flatMap`: Apply f on all RDD contents, return an RDD with the contents of all i $RDD[A].flatMap(f : A \to Iterable[B]) : RDD[B]$

. . .

`filter`: Apply predicate p, return items that satisfy it. $RDD[A].filter(p : A \to Boolean) : RDD[A]$

Examples of RDD transformations

All uses of articles in the Odyssey

```
val odyssey = sc.textFile("datasets/odyssey.mb.txt").
                 flatMap(_.split(" "))

odyssey.map(_.toLowerCase).
        filter(Seq("a", "the").contains(_))
```

Q: How can we find uses of all regular verbs in past tense?

. . .

```
odyssey.filter(x => x.endsWith("ed"))
```

Q: How can we remove all punctuation marks?

. . .

```
odyssey.map(x => x.replaceAll("\\p{Punct}|\\d", ""))
```

Common actions on RDDs

`collect`: Return all elements of an RDD $RDD[A].collect() : Array[A]$

. . .

`take`: Return the first n elements of the RDD $RDD[A].take(n) : Array[T]$

. . .

`reduce`, `fold`: Combine all elements to a single result of the same time.
$RDD[A].reduce(f : (A, A) \to A) : A$

. . .

`aggregate`: Aggregate the elements of each partition, and then the results for all the part
$RDD[A].aggr(init : B)(seqOp : (B, A) \to B, combOp : (B, B) \to B) : B$

Examples of RDD actions

How many words are there?

```
val odyssey = sc.textFile("datasets/odyssey.mb.txt").flatMap(_.split(" "))

odyssey.map(x => 1).reduce((a,b) => a + b)
```

How can we sort the RDD?

```
odyssey.sortBy(x => x)
```

How can we sample data from the RDD?

```
val (train, test) = odyssey.randomSplit(Array(0.8, 0.2))
```

Pair RDDs RDDs can represent any complex data type, if it can be iterated. Spark treats RDDs of the type `RDD[(K,V)]` as special, named PairRDDs, as they can be both iterated and indexed.

Operations such as `join` are only defined on Pair RDDs, meaning that we can only combine RDDs if their contents can be indexed.

We can create Pair RDDs by applying an indexing function or by grouping records:

```
val rdd = List("foo", "bar", "baz").parallelize // RDD[String]
val pairRDD = rdd.map(x => (x.charAt(0), x))  // RDD[(Char, String)]
pairRDD.collect
// Array((f,foo), (b,bar), (b,baz))
val pairRDD2 = rdd.groupBy(x => x.charAt(0))  // RDD[(Char, Iterable(String))]
pairRDD2.collect
//Array((b,CompactBuffer(bar, baz)), (f,CompactBuffer(foo)))
```

Transformations on Pair RDDs The following functions are only available on `RDD[(K,V)]`

`reduceByKey`: Merge the values for each key using an associative and commutative
$reduceByKey(f : (V, V) \to V) : RDD[(K, V)]$

. . .

`aggregateByKey`: Aggregate the values of each key, using given combine functions
$aggrByKey(zero : U)(f : (U, V) \to U, g : (U, U) \to U) : RDD[(K, U)]$

. . .

`join`: Return an RDD containing all pairs of elements with matching keys
$join(b : RDD[(K, W)]) : RDD[(K, (V, W))]$

Pair RDD example: `aggregateByKey`

How can we count the number of occurrences of part of speach elements?

```
object PartOfSpeach {
  sealed trait EnumVal
  case object Verb extends EnumVal
  case object Noun extends EnumVal
  case object Article extends EnumVal
  case object Other extends EnumVal
  val partsOfSpeach = Seq(Verb, Noun, Article, Other)
}

def partOfSpeach(w: word): PartOfSpeach = ...

odyssey.groupBy(partOfSpeach).
        aggregateByKey(0)((acc, x) => acc + 1,
                          (x, y) => x + y)
```

D: What type conversions take place here?

Pair RDD example: `join`

```
case class Person(id: Int, name: String)
case class Addr(id: Int, person_id: Int,
                address: String, number: Int)

val pers = sc.textFile("pers.csv") // id, name
val addr = sc.textFile("addr.csv") // id, person_id, street, num

val ps = pers.map(_.split(",")).map(x => Person(x(0).toInt, x(1)))
val as = addr.map(_.split(",")).map(x => Addr(x(0).toInt, x(1).toInt,
                                              x(2), x(3).toInt))
```

Q: What are the types of `ps` and `as`? How can we join them?

. . .

```
val pairPs = ps.keyBy(_.id)
val pairAs = as.keyBy(_.person_id)

val addrForPers = pairAs.join(pairPs) // RDD[(Int, (Addr, Person))]
```

Join types

Given a "left" `RDD[(K,A)]` and a "right" `RDD[(K,B)]`

- Inner Join (`join`): The result contains only records that have the keys in both RDDs.
- Outer joins (`left/rightOuterJoin`): The result contains records that have keys in either the "left" or the "right" RDD in addition to the inner join results.

$left.loj(right) : RDD[(K, (A, Option[B]))]$ $left.roj(right) : RDD[(K, (Option[A], B))]$

- Full outer join: The result contains records that have keys in any of the "left" or the "right" RDD in addition to the inner join results.

$left.foj(right) : RDD[(K, (Option[A], Option[B]))]$

7.3 RDDs under the hood

Internals

From the RDD.scala, line 61.

Internally, each RDD is characterized by five main properties:

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a Partitioner for key-value RDDs
- Optionally, a list of preferred locations to compute each split on

D: Why does an RDD need are all those?

Partitions

Even though RDDs might give the impression of continuous memory blocks spread across a cluster, data in RDDs is split into *partitions*.

Partitions define a unit of computation and persistence: any Spark computation transforms a partition to a new partition.

If during computation a machine fails, Spark will redistribute its partitions to other machines and restart the computation *one those partitions only*.

The partitioning scheme of an application is configurable; better configurations lead to better performance.

How does partitioning work?

Spark supports 3 types of partitioning schemes:

- **Default partitioning:** Splits in equally sized partitions, without knowing the underlying data properties.

Extended partitioning is only configurable on Pair RDDs.

- **Range partitioning:** Takes into account the natural order of keys to split the dataset in the required number of partitions. Requires keys to be naturally ordered and keys to be equally distributed across the value range.
- **Hash partitioning:** Calculates a hash over the each item *key* and then produces the modulo of this hash to determine the new partition. This is equivalent to:

```
key.hashCode() % numPartitions
```

Partition dependencies

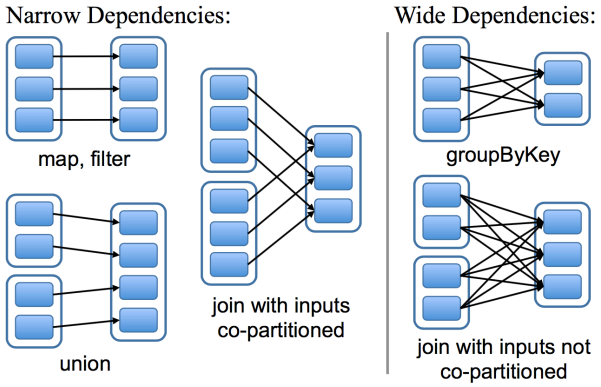


Figure 7.4: Partition Dependencies

Narrow dependencies: Each partition of the source RDD is used by at most one partition of the target RDD

Wide dependencies: Multiple partitions in the target RDD depend on a single partition in the source RDD.

RDD lineage

RDDs contain information on how to compute themselves, including dependencies to other RDDs. Effectively, RDDs create a directed acyclic graph of computations.

To demonstrate this, consider the following example:

```
val rdd1 = sc.parallelize(0 to 10)
val rdd2 = sc.parallelize(10 to 100)
val rdd3 = rdd1.cartesian(rdd2)
val rdd4 = rdd3.map(x => (x._1 + 1, x._2 + 1))
```

Q: What are the computation dependencies here?

. . .

```
scala> rdd4.toDebugString
```

```
res3: String =
(16) MapPartitionsRDD[3] at map at <console>:30 []
 |   CartesianRDD[2] at cartesian at <console>:28 []
 |   ParallelCollectionRDD[0] at parallelize at <console>:24 []
 |   ParallelCollectionRDD[1] at parallelize at <console>:24 []
```

Shuffling

When operations need to calculate results using a common characteristic (e.g. a common key), this data needs to reside on the same physical node. This is the case with all "wide dependency" operations. The process of re-arranging data so that similar records end up in the same partitions is called *shuffling*.

Shuffling is very expensive as it involves moving data across the network and possibly spilling them to disk (e.g. if too much data is computed to be hosted on a single node). Avoid it at all costs!

See this link for a great write up on Spark shuffling

Shuffling example

Persistence

Data in RDDs is stored in three ways:

- **As Java objects**: Each item in an RDD is an allocated object
- **As serialized data**: Special (usually memory-efficient) formats. Serialization is more CPU intensive, but faster to send across the network or write to disk.
- **On the filesystem**: In case the RDD is too big, it can be mapped on a file system, usually HDFS.

Persistence in Spark is configurable and can be used to store frequently used computations in memory, e.g.:

```
val rdd = sc.textFile("hdfs://host/foo.txt")
val persisted = rdd.map(x => x + 1).persist(StorageLevel.MEMORY_ONLY_SER)
```

`persisted` is now cached. Further accesses will avoid reloading it and applying the `map` function.

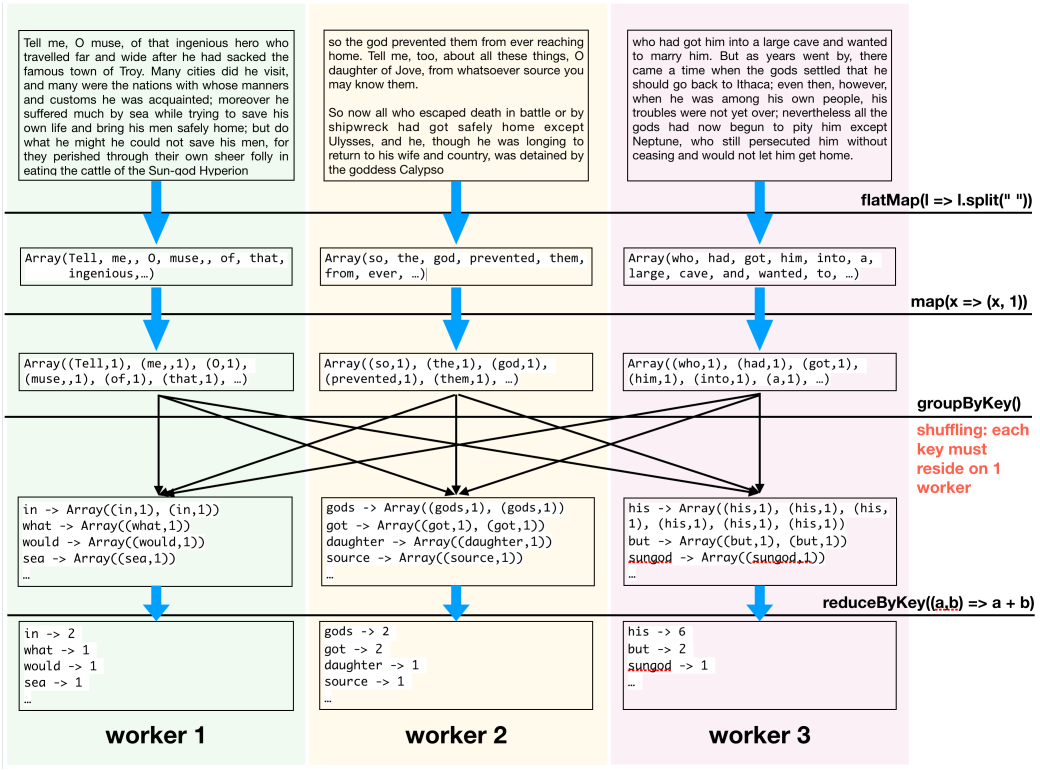


Figure 7.5: Shuffling explained

7.4 Spark applications

Spark cluster architecture

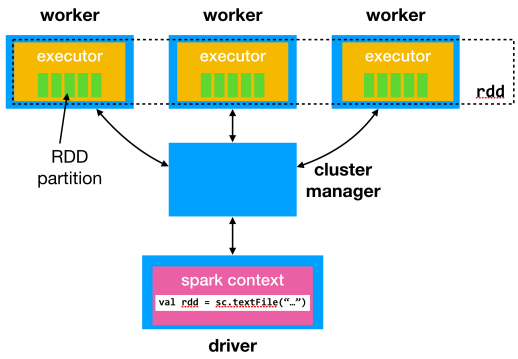


Figure 7.6: Spark cluster architecture

- *Executors* do the actual processing; *worker* nodes can contain multiple executors.
- The *driver* accepts user programs and returns processing results
- The *cluster manager* does resource allocation

A Spark application

A Spark application:

- begins by specifying the required cluster resources, which the cluster manager seeks to fulfil. If resources are available, executors are started on worker nodes.
- creates a Spark context, which acts as the driver.
- issues a number of *jobs*, which load data partitions on the executor heap and run in threads on the executor's CPUs
- when it finishes, the cluster manager frees the resources.

What is a Spark job?

Jobs are created when an action is requested. Spark walks the RDD dependency graph *backwards* and builds a graph of stages.

Stages are jobs with wide dependencies. When such an operation is requested (e.g. `groupByKey` or `sort`) the Spark scheduler will need to reshuf-

fle/repartition the data. Stages (per RDD) are always executed serially. Each stage consists of one or more tasks.

Tasks is the minimum unit of execution; a task applies a narrow dependency function on a data partition. The cluster manager starts as many jobs as the data partitions.

A job graph

We can see how our job executes in we connect to our driver's WebUI (port 4040 on the driver machine).

Here is the graph for the word counting job we saw before.

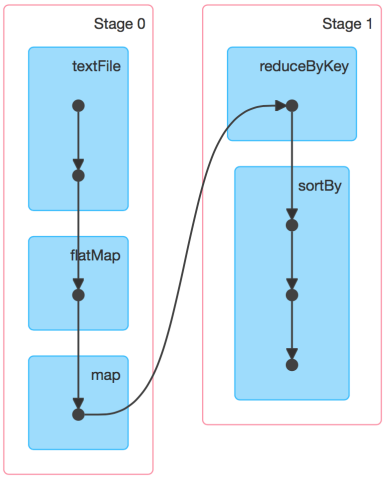


Figure 7.7: Word Count job graph

Requesting resources

Here is an example of how to start an application with a custom resource configuration.

```
spark-shell   \
    --master spark://spark.master.ip:7077 \
    --deploy-mode cluster  \
    --driver-cores 12
    --driver-memory 5g \
    --num-executors 52 \
```

```
    --executor-cores 6 \
    --executor-memory 30g
```

Fault tolerance

Spark uses RDD lineage information to know which partition(s) to recompute in case of a node failure.

Recomputing happens at the *stage* level.

To minimize recompute time, we can use checkpointing. With checkpointing we can save job *stages* to reliable storage. Checkpointing effective truncates the RDD lineage graph.

Spark clusters are reliable to node failures, but not to master failures. Running Spark on middleware such as YARN or Mesos is the only way to run multi-master setups.

7.5 Effective Spark

RDDs and formatted data

RDDs by default do not impose any format on the data they store. However, if the data is formatted (e.g. log lines with known format), we can create a schema and have the Scala compiler type-check our computations.

Consider the following data (common log format):

```
127.0.0.1 user-identifier frank [10/Oct/2000:13:55:36 -0700] \
 "GET /apache_pb.gif HTTP/1.0" 200 2326
```

We can map this data to a Scala *case class*

```
case class LogLine(ip: String, id: String, user: String,
                   dateTime: Date, req: String, resp: Int,
                   bytes: Int)
```

and use a regular expression to parse the data:

```
([^\s]+) ([^\s]+) ([^\s]+) ([^\s]+) "(.+)" (\d+) (\d+)
```

Then, we can use `flatMap` in combination with Scala's pattern matching to filter out bad lines:

```
import java.text.SimpleDateFormat
import java.util.Date

val dateFormat = "d/M/y:HH:mm:ss Z"
val regex = """([^\s]+) ([^\s]+) ([^\s]+) ([^\s]+) "(.+)" (\d+) (\d+)""".r
val rdd = sc.
    textFile("access-log.txt").
    flatMap ( x => x match {
      case regex(ip, id, user, dateTime, req, resp, bytes) =>
        val df = new SimpleDateFormat(dateFormat)
        new Some(LogLine(ip, id, user, df.parse(dateTime),
                         req, resp.toInt, bytes.toInt))
      case _ => None
      })
```

Then, we can compute the total traffic per month

```
val bytesPerMonth =
  rdd.
    groupBy(k => k.dateTime.getMonth).
    aggregateByKey(0)({(acc, x) => acc + x.map(_.bytes).sum},
                      {(x,y) => x + y})
```

Notice that all code on this slide is type checked!

Connecting to databases

The data sources that Spark can use go beyond `textFiles`. Spark can connect to databases such as

- MongoDB

```
val readConfig = ReadConfig(Map("uri" -> "mongodb://127.0.0.1/github.events"))
sc.loadFromMongoDB(readConfig),
val events = MongoSpark.load(sc, readConfig)

events.count
```

- MySQL or Postgres over JDBC

```
val users = spark.read.format("jdbc").options(
  Map("url" ->  "jdbc:mysql://localhost:3306/ghtorrent?user=root&password=",
  "dbtable" -> "ghtorrent.users",
  "fetchSize" -> "10000"
  )).load()

users.count
```

or to distributed file systems like HDFS, Amazon S3, Azure Data Lake etc

Optimizing Partitioning

Partitioning becomes an important consideration when we need to run iterative algorithms. Some cases benefit a lot from defining custom partitioning schemes:

- Joins between a large, almost static dataset with a much smaller, continuosly updated one.
- `reduceByKey` or `aggregateByKey` on RDDs with arithmetic keys benefit from range partitioning as the shuffling stage is minimal (or none) because reduction happens locally!

Broadcasts

From the docs: Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

Broadcasts are often used to ship precomputed items, e.g. lookup tables or machine learning models, to workers so that they do not have to retransfer them on every shuffle.

With broadcasts, we can implement efficient in-memory joins between a processed dataset and a lookup table.

```
val curseWords = List("foo", "bar") // Use your imagination here!
val bcw = sc.broadcast(curseWords)

odyssey.filter(x => !curseWords.contains(x))
```

Broadcasted data should be relatively big and immutable.

Accumulators

Some times we need to keep track of variables like performance counters, debug values or line counts while computations are running.

```
// Bad code
var taskTime = 0L
odyssey.map{x =>
  val ts = System.currentTimeMillis()
  val r = foo(x)
  taskTime += (System.currentTimeMillis() - ts)
  r
}
```

To make it work, we need to define an accumulator

```
val taskTime = sc.accumulator(0L)
odyssey.map{x =>
  val ts = System.currentTimeMillis()
  val r = foo(x)
  taskTime += (System.currentTimeMillis() - ts)
  r
}
```

Using accumulators is a *side-effecting* operation and should be avoided as it complicates design. It is important to understand that accumulators are aggregated at the *driver*, so frequent writes will cause large amounts of network traffic.

Image credits

- Hadoop processing model
- The DryadLINQ image is fromt the DryadLINQ OSDI 08 paper.
- How Map/Reduce works?
- Spark partition dependencies image is from the Spark OSDI paper.

Chapter 8

Spark Datasets

RDDs and structure

RDDs only see binary blobs with an attached type

Databases can do many optimizations because they know the data types for each field

How to efficiently join?

D: Say that we have an RDD of 100k people and another of 300k addresses. We need to get the details of all people that live in Delft, along with their addresses. Which of the following joins will be faster?

```
val people : RDD[Person]
val addresses: RDD[(Int, Address)]

//Option 1
people.keyBy(_.id).join(addresses.filter(x._2._2.city == "Delft"))

//Option 2
people.keyBy(_.id).join(addresses).filter(x._2._2.city == "Delft")

//Option 3
people.keyBy(_.id).cartesian(addresses).filter(x._2._2.city == "Delft")
```

. . .

Join 1 is the fastest. However, why do we have to do this optimization ourselves? Because Spark does not know the schema and what the λs that are args to `filter` do!

8.1 Spark SQL

In Spark SQL, we trade some of the freedom provided by the RDD API to enable:

- declarativity, in the form of SQL
- automatic optimizations, similar to ones provided by databases
  - execution plan optimizations
  - data movement/partitioning optimizations

The price we have to pay is to bring our data to a (semi-)tabular format and describe its schema. Then, we let relational algebra work for us.

Spark SQL basics

SparkSQL is a library build on top of Spark RDDs. It provides two main abstractions:

- Datasets, collections of strongly-typed objects. Scala/Java only!
- Dataframes, essentially a `Dataset[Row]`, where `Row` $\approx$ `Array[Object]`. Equivalent to R or Pandas Dataframes
- SQL syntax

It offers a query optimizer (Catalyst) and an off-heap data cache (Tungsteen).

It can directly connect and use structured data sources (e.g. SQL databases) and can import CSV, JSON, Parquet, Avro and data formats by inferring their schema.

How efficient is Spark SQL?

A blog post by MySQL experts Percona wanted to find the number of delayed files per airline using the air trafic dataset.

Running the query on MySQL, it took 19 mins.

On the same server, they used Spark SQL to connect to MySQL, partitioned the Dataframe that resulted from the connection and run the query in Spark SQL. It took 192 secs!

This was the result of Catalyst rewriting the SQL query: instead of 1 complex query, SparkSQL run 24 parallel ones using range conditions to restrict the examined data volumes. MySQL cannot do this!

The SparkSession

Similarly to normal Spark, SparkSQL needs a context object to invoke its functionality. This is the `SparkSession`.

If a `SparkContext` object exists, it is straightforward to get a SparkSession

```
val ss = new SparkSession(sc)
```

8.2 Creating Data Frames and Datasets

1. From RDDs containing tuples, e.g. `RDD[(String, Int, String)]`

```
import spark.implicits._

val df = rdd.toDF("name", "id", "address")
```

2. From RDDs with known complex types, e.g. `RDD[Person]`

```
val df = persons.toDF() // Columns names/types are infered!
```

3. From RDDs, with manual schema definition

```
val schema = StructType(Array(
  StructField("level", StringType, nullable = true),
  StructField("date", DateType, nullable = true),
  StructField("client_id", IntType, nullable = true),
  StructField("stage", StringType, nullable = true),
  StructField("msg", StringType, nullable = true),
))

val rowRdd = sc.textFile("ghtorrent-log.txt").
```

```
                map(_.split(" ")).
                map(r => Row(r(0), new Date(r(1)), r(2).toInt,
                             r(3), r(4)))

val logDF = spark.createDataframe(rowRDD, schema)
```

---

4. By reading (semi-)structured data files

```
val df = spark.read.json("examples/src/main/resources/people.json")
```

or

```
df = sqlContext.read.csv("/datasets/pullreqs.csv", sep=",", header=True,
                         inferSchema=True)
```

RDDs and Datasets

Both RDDs and Datasets:

1. Are stronlgy typed (they include a type parameter, i.e. `RDD[T]`)
2. Contain objects that need to be serialized

The main difference is that Datasets use special `Encoders` to convert the data in compact internal formats that Spark can use directly when applying operations such as `map` or `filter`.

The internal format is very efficient; it is not uncommon to have in-memory data that use *less* memory space than their on disk format.

Moral: *when in doubt, provide a schema!*

Columns

Columns in DataFrames/Sets can be accessed by name:

```
df["team_size"]
df.team_size
```

or in Scala

```
df("team_size")
$"team_size" //scala only
```

`Columns` are defined by *expressions*. The API overides language operators to return expression objects. For example, the following:

```
df("team_size") + 1
```

is syntactic sugar for

```
spark.sql.expressions.Add(df("team_size"), lit(1).expr)
```

Data frame operations

DataFrames/Datasets include all typical relational algebra operations:

- Projection

```
df.select("project_name").show()
df.drop("project_name", "pullreq_id").show()
```

- Selection

```
df.filter(df.team_size.between(1,4)).show()
```

Joins

Dataframes can be joined irrespective of the underlying implementation, as long as they share a key.

```
people = sqlContext.read.csv("people.csv")
department = sqlContext.read.jdbc("jdbc:mysql://company/departments")

people.filter(people.age > 30).\
       join(department, people.deptId == department.id)
```

All types of joins are supported:

- Left outer:

```
people.join(department, people.deptId == department.id,
            how = left_outer)`
```

- Full outer:

```
people.join(department, people.deptId == department.id,
            how = full_outer)
```

Grouping and Aggregations

When `groupBy` is called on a Dataframe, it is (conceptually) split in a key/value structure, where key is the different values of the column groupped upon and value are rows containing each individual value.

Same as SQL, we can only apply aggregate functions on groupped Dataframes

```
df.groupBy(df.project_name).mean("lifetime_minutes").show()
```

8.3 SparkSQL under the covers

Why is SparkSQL so fast? Because it uses *optimisation* and *code generation*.

The key to both features is that code passed to higher order functions (e.g. the predicate to `filter`) is syntactic sugar to generate *expression trees*.

```
df.filter(df("team_size") > (3 + 1))
```

is converted to

```
df.filter(GreaterThan(
            UnresolvedAttribute("team_size"),
            Add(Literal(3) + Literal(1))))
```

This corresponds to an Abstract Syntax Tree

Optimization

The optimizer uses tree patterns to simply the AST. For example, the following tree:

```
val ast = GreaterThan(
            UnresolvedAttribute("team_size"),
            Add(Literal(3) + Literal(1)))
```

can be simplified to:

. . .

```
val opt = ast.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

```
GreaterThan(UnresolvedAttribute("team_size"),
            Literal(4))
```

Catalyst knows 10s (100s?) such optimizations. The purpose is to minimize work done when the query is evaluated on real data.

Code generation

Given an expression tree and a context (the Dataframe), Catalyst exploits Scala's compiler ability to manipulate ASTs as Strings.

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case GreaterThan(left, right) =>
        q"${compile(left)} > ${compile(right)}"
}
```

The previous expression tree is converted (roughly!) to

```
df.filter(row => row.get("team_size") > 4)
```

and gets compliled at runtime to JVM code.

Deriving an execution plan

Catalyst performs the following steps:

- Analysis, to get to know the types of column expressions. This will, e.g. resolve `UnresolvedAttribute("team_size")` to `Attribute("team_size")` of type `Int`(and would also check whether `team_size` exists in `df`)
- Rule optimization, as described above
- Physical optimization, to minimize data movement
- Code generation, as described before

The end result is native code that runs in optimal locations on top of an RDD.

Chapter 9

Data processing at the command line

Figure 9.1: Ken Thomson and Dennis Ritchie, the original authors of Unix

9.1 The UNIX operating system

UNIX is a true multiuser operating system. Users have their own private space on the machine's harddisk and are identified by an id number.

- All users have a user name and a password and are required to enter them correctly before using the computer.
- The user with id 0 is the system's superuser or administrator and its traditional name is ROOT.

- UNIX processes usually act on behalf of the initiating user.
- Multiple users can be running multiple processes at the same time.
- Users are organised in groups. A group is a set of users that share the same class of permissions.

Everything is a file

Everything in UNIX is represented by files. Everything can be:

- **Configuration files.** Both user profile and system/server configuration files are plain text files. This allows to easily backup/restore/compare configuration files and remote administration using low-bandwidth text consoles.
- **Devices.** Access is done using regular file I/O operations on filesystem objects (under /dev) that represent the real devices. For example `cat file.wav >/dev/sndcard` plays an audio file directly to the soundcard or `cat file.txt|lpr` prints it to a printer.

The UNIX filesystem

```
|--bin          Binaries required before mounting /usr
|--etc          System wide configuration files
|--home         Users' home directories
|--lib          Libraries required by the system
|--tmp          Temporary files. Everyone has RW access.
|--usr          Programs
| |--bin        Programs' executables
| |--lib        Programs' libraries
| |--local      Programs that are install locally
| |   |-bin,lib,share
| |--share Programs' required files (e.g. docs, icons)
| |--sbin  System administration programs
| |--src   Source files for the kernel and programs
|--var          Temporary space for running programs
```

The UNIX file system is the same in all Unix versions. You can rely on this.

Filesystem permissions

Files need to be protected against intentional or unintentional access. Filesystem permissions provide the system with information about who can access

a file or directory and what can he do with it.

```
$ ls -la /bin/ls
-rwxr-xr-x 1 root wheel   38624   Jul 15 06:29     /bin/ls
\        /     |      |       |   \          /     |
Permissions  Owner  Owning  Size     Date of     Filename
                    group          last modification
```

	read(4)	write(2)	execute(1)
owner	×		×
group	×		×
other	×		×

The same permission set can be expressed with the number z0755z

Filesystem paths

A path is a sequence of directories to reach a certain file, i.e. `/home/gousiosg/foo.txt`

Paths can be:

- Absolute - starting from the root directory "/" e.g. `/var/log/messages` - The system log file
- Relative to the current directory `.` e.g. if the current directory is `/var`, the relative path to the system log is `./log/messages` or `log/messages`

File listing commands

- **ls**: list files in a directory
  - `-l`: list details
  - `-a`: list hidden files (files that start with .)
- **find** `<dir>`: walk through a file hierarchy starting from `<dir>`
  - `-type [dfl]`: Only display *d*irectories, *f*iles or *l*inks
  - `-name str`: Only display entries that start `str`
  - `-{max|min}depth d`

File manipulation commands

- **touch** `<file>`: Create and empty file named `<file>` or update the modification time for the existing file `<file>`

- **cp** `<from> <to>`: copy file or directory `<from>` to the location specified by `<to>`
  - `-R`: copy directories recursively
  - `-p`: preserve filesystem permissions and attributes
- **mv** `<from_1> · · · <from_n> <to>`: move files or directories `<from*>` to directory `<to>`
  - `-n`: do not overwrite existing files
- **mkdir** : create an empty directory in the path specified by the . The path to the directory must be pre-existing.
  - `-p`: also create intermediate directories as required

Text file processing

- **cat** `file_1 · · · file_n`: concatenate and print files to standard output
- **less** `file`: displays a file on the screen allowing to browse it on both directions.
  - `q` exit
  - `/pattern` search for pattern in text, pressing / repeatedly moves through all occurrences.
- **echo** `<string>` write arguments to standard output
- **head**, **tail** `<file>`: display first/last lines of `<file>`
  - `-n` Number of lines to display
  - `-f` Display newly appended lines

Process management

- UNIX can do many jobs at once, dividing the processor's time between the tasks so quickly that it looks as if everything is running at the same time. This is called multitasking.
- The UNIX shell has process management capabilities. When running a process, pressing `Ctrl+Z` will suspend it.
- A process can be killed with `Ctrl+C`

- A process can be started at the background by appending a & after the command. i.e. `find / |sort &`

Manipulating running processes

- **ps** See which processes are running

```
 UID   PID  PPID   C STIME   TTY           TIME CMD
   0     1     0   0 28Nov17 ??        21:20.80 /sbin/launchd
   0    51     1   0 28Nov17 ??         0:41.63 /usr/sbin/syslogd
   0   52    1   0 28Nov17 ??      2:19.32 /usr/libexec/UserEventAgent (System)
```

- **kill** `-<singalno> <pid>`: Send a signal to a process Important signals:
  - `TERM`: informs the process that it should terminate.
  - `KILL`: directly kill a process

Advanced text flow control

A process in Unix can output to 2 data streams: `STDOUT` or `1` and `STDERR` or `2`. Unix supports 2 times of text flow control:

- **Redirects:** send a program's output to a file (`>`) or make a program read from a file (`<`)
  - `ps -ef > processes`
  - `>` Overwrites an existing file; to append, we use `>>`

```
$ echo "-What a nice day" > file
$ echo "-Indeed" >> file
$ cat file
-What a nice day
-Indeed
```

- **Pipes:** Forward a program's STDOUT line-by-line to the input of another program.

Pipelines

What makes Unix such a joy to use is that most commands read and write easy to process text. This allows us to combine commands in surprising ways, using the pipe (`|`) operator.

- `cat file |wc -l`: Count lines of file

. . .

- `find / -type d | sort`: View all directories sorted

. . .

- `cat /var/log/access_log |grep foo|tail -n 10`: See the last 10 accesses from the host "foo" to our system's web server.

. . .

- `cat /etc/passwd |cut -f1 -d':'|sort`: Get a sorted list of the system's users.

Documentation

UNIX is a self-documenting system. All commands/tools have a manual page that describes their arguments, input and output formats and sometimes, even programming interfaces. **man** `<cmd>` invokes the manual for a command.

```
$ man ls
LS(1)                            User Commands                           LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.
```

Unix systems are also traditionally documented by providing full access to the source code that comprises them.

Figure 9.2: Brian Kernighan and Rob Pike, authors of the homonymous seminal book

9.2 The UNIX programming environment

Running a command per input line

xargs `cmd` will run `cmd` on each line in `STDIN`

```
# Get file size statistics for the current directory
$ find . -type f -maxdepth 1 |xargs wc
```

`xargs` by default appends each line at the end of `cmd`. Some times, it may be necessary to append it in the middle. We use the `-I {}` option and

```
$ find . -type f -maxdepth 1|xargs -I {} echo File {} is in `pwd`
File ./labcontents.doc is in /Users/gousiosg/Documents/course-material/isrm
File ./Makefile is in /Users/gousiosg/Documents/course-material/isrm
[...]
```

`xargs` can process things in parallel with `-P` option.

In terms of data processing, `xargs` is the equivalent of `map`

Filtering lines with patterns

`grep` prints lines matching a pattern

- `-v`: invert search result (only print those that DO NOT match the pattern)
- `-i`: make matching case insensitive

- `-n`: print the line number of the match
- `-R`: recurse a directory structure

```
# Find all processes run by user 501
$ ps -ef | egrep "^ +501"

# Find all files that extend class Foo
$ grep -Rn "(Foo)" * | grep *.py

# Same, more efficient
$ find . -type f | grep .py$ | xargs grep -n "(Foo)"
```

Regular expressions in 1 min

- `.` Match any character once
- `*` Match the previous pattern 0 or more times
- `+` Match the previous pattern 1 or more times
- [**e-fF-M**] Match any character in the (ASCII) range `F-M` or `e-f`
- [**^e**] Match all characters *except* e
- **^** and **$** Match the beginning or the end of the line, respectively
- **()** Group together items for future reference
- **|** Match either the left or the right group

Simple text processing

`tr` character translator; can convert or delete specific characters

- `-s`: replace repeating characters into
- `-d`: delete a character

```
$ echo "foo bar" | tr 'o' 'a'
faa bar
# Replace tabs with spaces
$ tr '\t' ' ' < file.txt
# Remove all instances of #
$ tr -d '#' < file.txt
```

`cut` allows us to split a line into columns, given a character, and extract specific fields.

```
# Get a list of users and home directories
$ cut -f1,6 -d: /etc/passwd
```

```
# Get details for all users that are running java
$ ps -ef|tr -s ' '|grep "java"|cut -f3,11  -d' '
```

Stream text processing

`sed` Modify a string at its input in various ways using pattern matching

```
# Replace foo with bar in the input file
$ sed -e 's/foo/bar/' < file.txt

# Change the order of columns in a 2 column file
$ sed -e 's/^\(.*\) \(.*\)$/\2 \1/' < file.txt

# Remove lines 3 and 5 from the input
$ sed -e '3d' -e '5d' < book.txt
```

`sed` is a domain specific language of its own. You can find a thorough manual here.

Sorting data

`sort` writes a (lexicographical) sorted concatenation of all input files to standard output, using Mergesort

- `-r`: reverse the sort
- `-n`: do a numeric sort
- `-k` and `-t`: merge by the nth column (argument to `-k`). `-t` specifies what is the separator character

`uniq` finds unique records in a sorted file

```
# Print the 10 most used lines in foo
$ cat foo| sort | uniq -c |sort -rn |head -n 10

# Sort csv file by the 6 field
sort -n -k 6 -t ','  datasets/file.csv
```

Joining data

`join` joins lines of two *sorted* files on a common field

- `-1`, `-2` specify fields in files 1 (first argument) and 2 (second argument) that represent keys

```
$ cat foodtypes.txt
3 Fat
1 Protein
2 Carbohydrate

$ cat foods.txt
Potato 2
Cheese 1
Butter 3

join -1 2 -2 2 <(sort foodtypes.txt) <(sort foods.txt)
```

Practically, `join` performs a join operation on KV pairs.

Orchestrating pipelines

`make` is a dependency-based command executor. It reads a rule file that specifies dependencies between files on disk along with production rules for those.

```
target: depend1 depend2 ... dependn
  commands to build the target given the dependencies
```

Make topologically sorts the specified dependency graph and executes commands (in parallel, if `-j` is specified) to generate all output files. If some of those already exist, make skips them.

```
result : file.csv

file.csv : file.txt /usr/bin/tr
  tr ' ' ',' file.txt > file.csv

file.txt :
  curl "http://a/web/page/file.txt" > file.txt
```

More `make`

```
# Find all Jupyter files
JUPYTER_INPUTS = $(shell find . -type f -name '*.ipynb')
```

```
# Generate the expected PDF file names
OUTPUTS_PDF = $(JUPYTER_INPUTS:.ipynb=.pdf)

# A recipe to convert any Jupyter notebook  to PDF
%.pdf: %.ipynb
    cd $(shell dirname $<) && \
    jupyter nbconvert --to pdf $(shell basename $<)

# Default (fake) target, depends on outputs
# to trigger computations
all : $(OUTPUTS_PDF)
```

- `$<` is an automatic variable meaning the name of the first prerequisite
- `$(shell args)` is a function that runs a command in the shell

9.3 Task-based tools

Figure 9.3: Richard Stallman, founder of the Free Software movement and co-author of many of the Unix tools we use on Linux

Execute a command on a remote host

`ssh` provides a way to securely login to a remote server and get a prompt. In addition, it enables us to remotely execute a command and capture its output

```
# List of files on host dutihr
ssh dutihr ls
```

Firewall piercing / tunneling: We can use ssh to access ports on machines where a firewall blocks them

```
# Connect to port
$ ssh -L 27017:mongoserver:27017 mongoserver

# On another terminal
$ mongo localhost:27017
```

Retrieve contents from URLs

curl queries a URL and prints the raw contents on the terminal

- `-H` Set an HTTP header, e.g. "Authorization: token OAUTH-TOKEN"
- `-i` Display all headers received
- `-s` Don't anything except from the response

```
curl -i "https://api.github.com/repos/vmg/redcarpet/issues"
```

We can then process contents with a pipeline

```
# Get all magnet links from a page
curl -s https://thepiratebay.org/browse/101 | # Get contents
tidy 2>/dev/null |                            # Tidy up HTML
grep magnet\:\? |                             # Only get links
tr -d '"'                                     # Remove quotes
```

Querying JSON data

json_pp pretty-prints JSON files

jq uses a Domain Specific Language (DSL) to query tree structures in JSON files.

```
# Extract information for a Cargo package descriptor
curl -s "https://crates.io/api/v1/crates/libc" |
```

```
jq -M '[.crate .id, .crate .repository, .crate .downloads|tostring]|join(", ")'

"libc, https://github.com/rust-lang/libc, 7267424"
```

Syncronizing files across hosts

rsync can be used to sync files between directories

- `-a` archive mode, preserve permissions and access times
- `-v` display files changed
- `--delete`

Run a command when a directory changes

inotifywait watches a directory for changes and prints a log of the changes

- `-m` enables monitor mode (run forever)
- `-r` watch directories recursively

```
## See changes to your database files
inotifywait -mr --timefmt '%d/%m/%y %H:%M' --format '%T %w %f' /mongo

## Copy all new files in the current directory to another location
inotifywait -mr . |
grep CLOSE_WRITE |
cut -f1 -d' ' |
xargs -I {} cp {} /tmp
```

9.4 Writing programs

The `bash` language

Bash (Bourne-again shell) is the name of the default command interpreter on most Unix environments. Bash is an almost complete programming language, with an interesting caveat: Many of its operators are programs that can be run individually!

Bash supports most

Variables

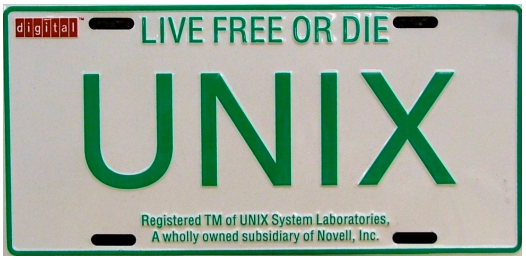


Figure 9.4: The Unix way

Variables in `bash` are strings followed by `=`, e.g. `cwd="foo"` and are dereferenced with `$`, e.g. `echo $cwd`.

```
# Store the results of running ls in a variable
listing=`ls -la`
echo $listing
```

An interesting set of variables are called *environment* variables. Those are declared by the operating system and can be read by all programs. The user can modify them with the `export` program.

```
$ export |grep PATH
$ export PATH=$PATH:/home/gousiosg/bin
$ export |grep PATH
```

Conditionals

Bash supports if / else blocks

```
if [ -e 'test' ]; then
  echo "File exists"
else
  echo "File does not exist"
fi
```

`[` is an alias to the program `test`.

- `[ $foo = 'test' ]`: Tests string equality
- `[ $num -eq 3 ]`: Tests number equality
- `[ ! expression ]`: Negates the expression

Loops

The `for` loop iterates over all items in the list provided as argument:

```
# Print 1 2 3 4...
for i in `seq 1 10`; do
  echo $i
done

# Iterate over all files in a directory
for i in $(ls); do
  echo `file --mime $i`
done
```

`while` executes a piece of code if the control expression is true

```
ls -fa |tr -s ' '|cut -f9 -d' '|
while read file; do
  echo `file --mime $file`
done
```

Command line input

`bash` maps special variables on command line inputs: `$0` is the program name, `$1` is the first argument, `$2` the second etc. More complex command lines (e.g. with switches) can be done with `getopt`.

```
#!/usr/bin/env bash

argA="defaultvalue"
while getopts ":a" opt; do
  case $opt in
    a)
      echo "-a was triggered!" >&2
      argA=$OPTARG
      ;;
    \?)
      echo "Invalid option: -$OPTARG" >&2
      ;;
  esac
done
```

The program above also illustrates the use of `case`

Content credits

- Unix license plate, by the Open Group

Bibliography

E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012. ISSN 0018-9162. doi: 10.1109/MC.2012.37.

Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. ISSN 0004-5411. doi: 10.1145/3149.214121. URL http://doi.acm.org/10.1145/3149.214121.

Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, August 2011. ISSN 0146-4833. doi: 10.1145/2043164.2018477. URL http://doi.acm.org/10.1145/2043164.2018477.

Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981. URL http://dl.acm.org/citation.cfm?id=1286831.1286846.

Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL http://doi.acm.org/10.1145/289.291.

Pat Helland. Immutability changes everything. *Queue*, 13(9):40, 2015.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12

(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972.

Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 2017. ISBN 978-1449373320.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL http://doi.acm.org/10.1145/279227.279229.

Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. Elsevier, 1988.

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.