Big Data Processing summary

Sep 1, 2021

This is a summary of CSE2520 Big Data Processing (DO NOT PRINT PAGE 54 (black background image))

Big and Fast Data

What is big data?

- Besides a buzzword is also used to describe:
  - *Data too large to be efficiently processed on a **single computer***
  - *Massive amounts of **diverse**, **unstructured** data produced by **high-performance applications***

How big is "big"?

- Typical numbers associated with big data:
  - 2.5 Exabytes ($2.5 \cdot 10^6 TBG \frac{\Delta d}{\Delta t} \gg \frac{\Delta c}{\Delta t}$) produced daily
  - IoT: 21.5 billion devices with internet access
  - Facebook, Amazon, Microsoft and Google store at least 1,200 petabytes of information
  - 100k google seraches per second
    - each query involves more than 1k machines
    - each query search touches more than 200 services
  - Amazon processes more than 1k orders per second
  - 1 billion of daily instagram users

Vs of Big data:

Main Vs:

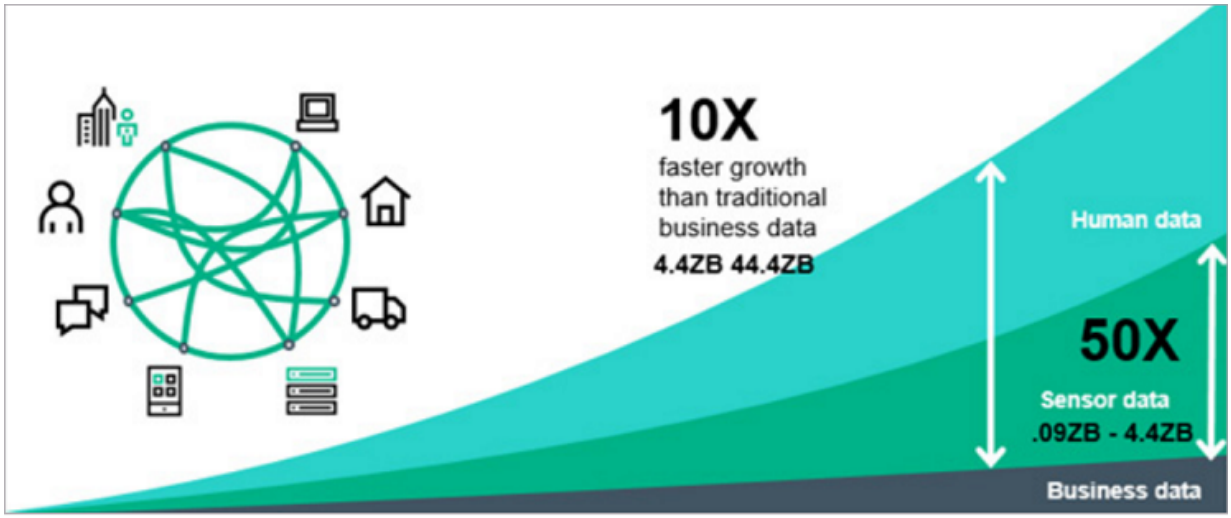
- Volume
- Variety (different forms and sources)
- Velocity (content changes quickly)

Other:

- (Business) value
- Veracity (accuracy)
- Validity (interpretation)
- Visibility
- Volability
- Virality

Volume

- We call big data big because of the volume
  - 90% of all data ever was created in the last 2 years
  - The global big data and business analytics market was valued at 138.9 billion in 2020 and is expected to grow



Variety

- Structured data: SQL tables, images, format is known
- Semi-structured data: JSON, XML
- Unstructured data: Text

Velocity

- Big data is not just big (volume) (and varied), but it's also generated and processed fast:
  - Data centers write a lot to log files
  - Social media posts
  - Stock market high-frequency trading (latency costs money)
  - Online advertising
- Data needs to be processed with soft or hard real-time guarantees

Big data processing

ETL cycle

- Extract: Convert raw or semi-structured data into structured data (i.e. JSON to database tables)
- Transform: Convert units, join data sources, clean data…
- Load: load the data into another system for further processing

Big data engenieering

- It's about building "pipelines"

Big data analytics

- It's about discovering patterns

Batch processing

- All data exists in some data store, a program processes the whole dataset at once (i.e. FRISS csv historical fraud batches)

Stream processing

- Processing of data as they arrive to the system (i.e. FRISS real time fraud score)

Data processing distribution

- Divide the data (i.e. csv of historical fraud) in chunks and apply the same task on all chunks at the same time, i.e. via multiple machines/CPUs with each machine assigned with it's unique chunk (data-parallelism: one task, many data splits)
- If possible, divide the task into independent sub-tasks that use the same data source (i.e. replace(",","."") for all records in a column and at the same time replace blanks with "0.0")

Desired properties of a big data processing system

- Robustness and fault-tolerance
- Low latency reads and updates
- Scalability
- Generalization
- Extensibility
- Ad hoc queries
- Minimal maintenance
- Debuggability

Large-scale computing

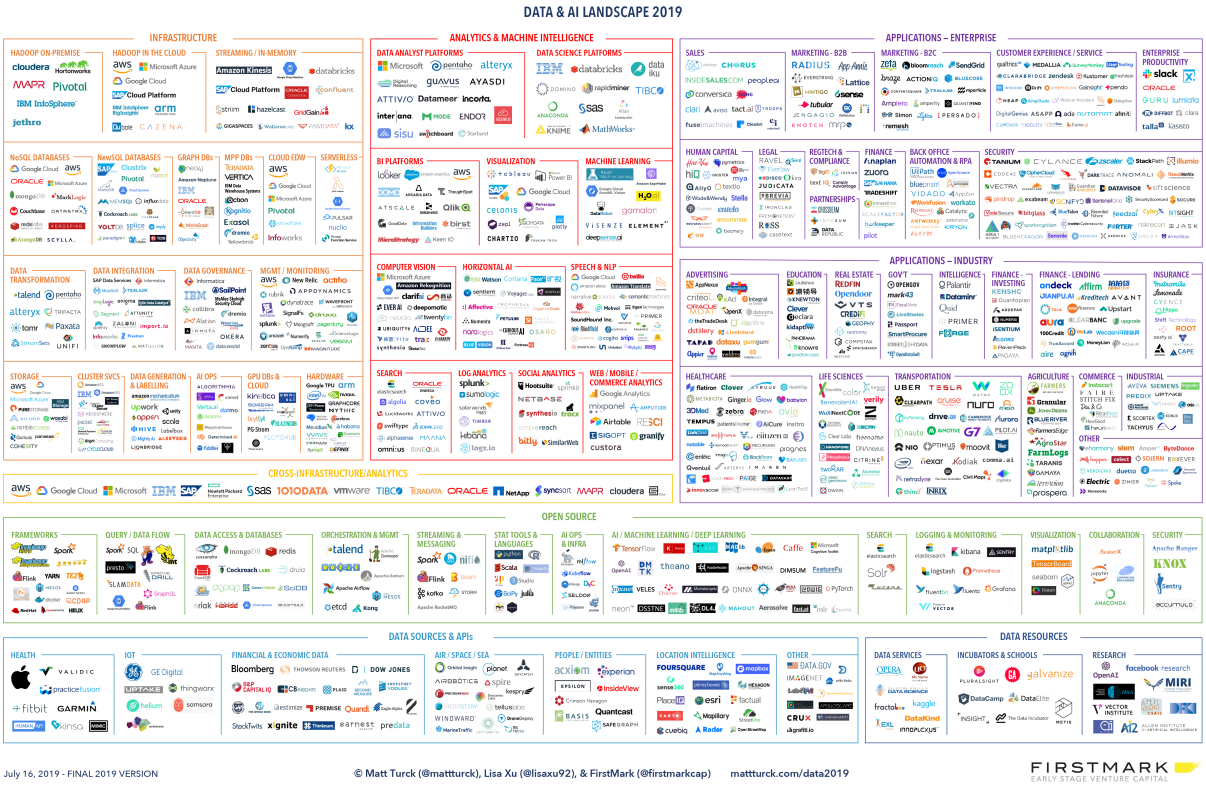
- Emerged in the 70's
- Phisicists used super computers for simulations in the 80's
- Shared-memory designs are still in large scale use
- What's new is: Large scale processing on **distributed**, **commodity** computers (i.e. average linux user home computer) enabled by advanced software using **elastic** resource allocation
- It is **software** and not hardware what drives the Big Data industry

Big Data tech timeline

Progress is mostly industry-driven:

- 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 launches 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

Current Big Data tech landscape



Problems solved with Big Data

- Modelling: What factors influence particular outcomes/behaviour?
- Inormation retrivals: Search engines, web scrappers
- Collaborative filtering: Recommending items based on items other users with similar tests ahve chosen
- Outlier detection: Discovering outstanding transactions

Big Data Programming Languages: Scala & Python

- The Big data and data science languages are
  - **scala**: for intensive systems
    - Strong point is the combination of functional programming and object oriented programming
    - documentation
  - **python**: for data analytics tasks
    - Strong point is the combination of object oriented and imperative programming
    - documentation
  - Both support object oriented programming, functional programming and imperative programming. But python is interpreted and scala is combined
- Other languages include
  - **Java**: the language in which most big data infrastructure is written into
  - **R**: Statistics language with great selection of libraries for serious data analytics and plotting tools
- **In CSE2520 we will be using only scala**

Hello world

- Scala is not only similar to java, but it can also actually run java code itself
  - Both Scala and Java are compiled to JVM bytecode
  - Scala can interoperate with JVM libraries
  - Scala is not sensitive to spaces/tabs. Blocks are denoted by `{}`

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

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

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

Declarations

- Scala:
  - Type inference used extensively (no need to explicitly declare the type of the variable like in js, but you may do it)
  - Two types of variables: vals are constants, vars are variables
  - In CSE2520 we will only use `val`

```
object Declarations extends App {
  var a: Int = 5
  val b = 6

  println(a)

  //b = 6 // wont compile program because val is like js' "const"

  println(b)

  // Type of foo is inferred
  val foo = new Array[Int](5)

  // var a = "Foo" // wont compile because a is already defined with val
  var c = "Foo"
  // c = 42 // won't compile due to type mismatch
  c = "Bar"

  print(c)
}
```

- python:
  - Optional typing, not enforced at runtime

```
import numpy as np

a: int = 5
a = "Foo"

a = np.array([a, 6, 7, 8])

print(a)
```

```
['Foo' '6' '7' '8']
```

Declaring functions

- Scala:
  - Statically typed

  - Evaluated expressions have types
  - The return type is the most generic type of all return expressions

```
object Functions extends App {
 println(max(3,1))

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

- Python:
  - Dynamically typed
  - Mostly based on statements
  - Types are optional

```
def maxi(x: int, y: int) -> int:
    if x >= y:
        return x
    else:
        return y


print(maxi(5, 3))  # you cant use functions before they're defined since python is execu
```

Declaring classes

- Scala
  - A default constructor is created automatically

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

```
  // Type of a is inferred
  val a = new ClassExample(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
```

```
1
4.0
10.0
```

- Python

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

```
import class_example as ce

a = ce.Foo(3, 2)
print(a.x)

a.x = "foo"
```

```
print(a.x)
```

```
3
foo
```

Inheritance

- Scala
  - Traits are equivalent to java interfaces (abstract classes (cant be initiliazed itself) whose methods don't have body) and includes attributes

```
object Inheritance extends App {
  var c = new Baz(5, 6f, 7)
  println(c.asString())
}


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: Double, private val z: Int)
  extends Foo(x, y) with Printable {
  override val s: String = new String( //java code
    String.valueOf(x)
      + " "
      + String.valueOf(y)
      + " "
      + String.valueOf(z))

  override def asString(): String = s
}
```

```
5 6.0 7
```

- Python

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


class ThreeDimensionPoint(TwoDimensionPoint):
    def __init__(self, x, y, z):
        TwoDimensionPoint.__init__(self, x, y)
        self.z = z


a = TwoDimensionPoint(1, 2)
b = ThreeDimensionPoint(10, 20, 30)

print(a.x, a.y)
print(b.x, b.y, b.z)
```

```
1 2
10 20 30
```

- In both scala and python children can override parents

Data classes

- Data classes are blueprints for immutable objects.
- We use them to represent data records.
- Both languages implement equals (or **eq**) for them, so we can compare objects directly.

- Scala:

```
object DataClass extends App {
  val p = Person("Name", Address("Street", 2))
  // p.name = "sergio" //wont compile

  println(new String(p.name + " lives at " + p.address.street + " " + p.address.number))
}

case class Address(street: String,
                   number: Int)

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

```
Name lives at Street 2
```

- Python:

```
from dataclasses import dataclass


@dataclass
class Address:
    street: str
    number: int


@dataclass
class Person:
    name: str
    address: Address


p = Person("G", Address("a", 2))
p.name = "Sergio"  # does compile
print(p)
```

```
Person(name='Sergio', address=Address(street='a', number=2))
```

Pattern matching in Scala

- It's the equivalent of switch in java

- Java:

```
public String getInstruction() {
        switch (instruction) {
            case LDA:
                return "0001";
```

```
            case ADD:
                return "0010";
            case SUB:
                return "0011";
            case STA:
                return "0100";
            case LDI:
                return "0101";
            case JMP:
                return "0110";
            case JPC:
                return "0111";
            case JPZ:
                return "1000";
            case OUT:
                return "1110";
            case HLT:
                return "1111";
            default:
                return "0000";
        }
    }
```

- Scala:

```
object PatternMatching extends App {
  val instruction = "LDA"

  def getInstruction(opcode: String): String =
    instruction match {
      case "LDA" => "0001"
      case "ADD" => "0010"
      case "SUB" => "0011"
      case "STA" => "0100"
      case "LDI" => "0101"
      case "JMP" => "0110"
      case "JPC" => "0111"
      case "JPZ" => "1000"
      case "OUT" => "1110"
      case "HLT" => "1111"
      case _ => "0000"
    }

  println(getInstruction(instruction))

}
```

```
0001
```

Basic data types

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

Sequences/Lists

- Basic properties:
  - Size is bounded by memory
  - Items can be accessed by an index (`list1[i]` or `l[j]`)
  - Items can only be inserted at the end (append)
  - Can be sorted
- Python

```
list1 = [1, 2, 3, 4]
```

- Although numpy arrays are actually more handy

```
import numpy as np

list1 = [1,2,3,4]
array1 = np.array(list1)
```

- Scala

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

Sets

- Stores unique value without any particular order
  - Size is bounded by memory
  - Can be queried for containment
  - Set operations: union, intersection, difference, subset
- Scala:

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

Maps or Dictionaries

- Maps or Dictionaries or Associative arrays are a collection of `(k,v)` pairs in such a way that each `k` appears only once.
  - Accessomg a value given a key is very fast ($O(1) V \frac{\Delta c}{\Delta t} \gg \frac{\Delta d}{\Delta t}$)
- Python

```
values = {

     "key1" : 1.0,

     "key2" : 2,

     "key3" : [1,2,3]

 }
```

- Scala

```
val m = Map(("a", 1), ("b",2))
val m: scala.collection.immutable.Map[String,Int] = Map(a -> 1, b -> 2)
```

Graphs

- A graph data structure consists of a finite set of vertices or nodes
  - if these nodes/vertices are stored in ordered pairs, the graph is "directed"
  - If the nodes/vertices are stored in unordered pairs, then it's an undirected graph

- Nodes can contain attributes
- Edges can contain weights and directions
- Graphs are usually represented as `Map[Node, List[Edge]]` where in Scala:

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

Nested data types: Trees

- They are ordered graphs without loops
  - Which is basically a set of nested maps

```
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. In Scala:

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

Tuples

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

```
val record = Tuple4[Int, String, String, Int] (1, "Matt", "Damon", 1970)
// alternatively
val record = (1, "Matt", "Damon", 1970)
```

- You can also have nested tuples (recall that scala automatically infers the types of the tup0le contents, including another tuple)

```
val a = (1, ("Foo", 2)) // type: Tuple2[Int, Tuple2[String, Int]]
// alternatively: (Int, (String, Int))

println(a._1) // prints 1
println(a._2._1) // prints Foo
```

Relations

- A relation is a Set of n-tuples (d1,d2,...,dn) of the same type; one of the tuple elements denotes a key. Keys cannot be repeated.
- 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.

- Scala

```
val movie1 = (1, "Martian", "PG-13", 2015, 2)
val movie2 = (2, "Prometheus", "R", 2012, 2)
val movie3 = (3, "2001: Space Odyssey", "G", 1968, 1)

val movies = Set(movie1, movie2, movie3)

val stanley = (1, "Stanley Kubrick", 1928)
val ridley = (2, "Ridley Scott", 1937)

val directors = Set(stanley, ridley)
```

Key/Value pairs

- A key/value pair (or K/V) is a more general type of a relation, where each key can appear more than once.
- Scala

```
// We assume that the first Tuple element represents the key
val a = (1, ("Martian", 2015))
val b = (1, ("Prometheus", 2012))

val kv = List(a, b)
// type: List[(Int, (String, Int))]
```

- Another way to represent K/V pairs is with a Map

```
val xs = Map(1 -> List(("Martian", 2015), ("Prometheus", 2012)))
// type: Map[Int, List[(String, Int)]]
```

- K and V are flexible: that's why the Key/Value abstraction is key to NoSQL databases, including MongoDB, DynamoDB, Redis etc.

Functional programming

- The basics of functional programming apply to data processing with tools like Hadoop, Spark and Flink.
- Functional programming is a programming paradigm where programs are constructed by applying and composing functions.
- Functional programming characteristics:
  - Absence of side-effects: A function, given an argument, always returns the same results irrespective of and without modifying its environment.
  - Immutable data structures: Side-effect free functions operate on immutable data.
  - Higher-order functions: Functions can take functions as arguments to parametrize their behavior
  - Laziness: The art of waiting to compute till you can wait no more
- Functional programming comes from lambda calculus, a formal mathematical logic system for expressing computation based on functions that operate on immutable data

Function signatures

- Function foo takes as arguments an array/list of type A and an argument of type B and returns an argument of type C
  - foo = function name
  - x and y = names of function arguments
  - [A] and B = types of function arguments
  - $\rightarrow n \times n$Where = denotes return type
  - C = type of the returned result
  - [A] = denotes that type A can be traversed (i.e. an array)

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.

```
var max = -1

def greaterOrEqual(a: Int, b: Int): Boolean = {
  if(a >= b) {
      max = a // side effect!
      true
    } else {
      max = b // side effect!
      false
    }
}
```

- As a general rule, any function that returns nothing (`void` or `Unit`) does a side effect!
- Example of side effects
  - Modifying a variable
  - Modifying a data structure in place: In FP, data structures are always persistent.
  - Setting a field on an object: OO is not FP!
  - Throwing an exception or halting with an error: In FP, we use types that encapsulate and propagate erroneous behavior
  - Printing to the console or reading user input, reading writing to files or the screen: In FP, we encapsulate external resources into Monads.

Pure functions

- A pure function depends only on its declared inputs and its internal algorithm to produce its output.
  - It does not read any other values
  - It does not have any side effects

```
def greaterOrEqual(a: Int, b: Int, max: Int): (Boolean, Int) = {
  if(a >= b) (true, a)
  else (false, b)
}
```

- Pure functions offer referential transparency.
  - An expression is said to be referentially transparent if it can be replaced by its value and not change the program's behavior.
  - Referential transparency enables simpler reasoning about programs.

Immutable data structures

- Functional data structures are operated on pure functions and they are immutable
- Scala has immutable lists, tuples, maps and sets

```
val oneTwoThree = List(1, 2, 3)
val oneTwoThree_2 = 1 :: 2 :: 3 :: Nil
val one = oneTwoThree.head
val twoThree = oneTwoThree.tail
```

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

Pattern matching on data structures

- In the context of functional programming pattern matching is just the java equivalent of switch
  - Syntax sugar for if else
- In scala, on top of checking if the contents of the object equals the content of the "switch" case, scala's "switch" feature also allows you to check for type matching and to match for certain values in a list:

```
object PatternMatching extends App {

  val x = List(1, 2, 3, 4, 5) match {
    case x :: 2 :: 4 :: xs => x
    case Nil => 42
    case x :: z :: 3 :: 4 :: xs => x + z
    case h :: t => h
    case _ => 404
  }
  println(x)
}
```

`3`

- Note that for a list in scala in the pattern matching using a variable nambe for one of the element is like using a wildcard, which we can use to extract the value from the list at the position the wildcard was used

Higher-order functions

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

```
// Apply f to all elements of list
def filter(xs: List[A], f: A => Boolean) : List[A]
```

Apply to all

- Using applyToAll to define other functions:

```
def incrementAll2(ints: List[Int]): List[Int] = applyToAll(ints, (x:Int) => x + 1)
def doubleAll2(ints: List[Int]): List[Int] = applyToAll(ints, (x:Int) => x * 2)
```

- See how `applyToAll(ints, (x:Int) => x + 1)` is like a foreach loop
  - For each `int x` in `ints` return `x + 1`
- But besides functional programming being a set of syntactic sugar to define functions in one line, it's also a philosophy of coding (i.e. "no side effects", "functions are first class citizens", "immutable data structures" (no states), "laziness")
- Scala collections has built-in `map` function to abstract the syntactic sugar even more:

```
def incrementAll3(ints: List[Int]): List[Int] = ints.map((x:Int) => x + 1)
def doubleAll3(ints: List[Int]): List[Int] = ints.map((x:Int) => x * 2)
```

Important higher-order functions

- `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.
- `foldL(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. AKA `reduce`.

Aux higher-order functions

- `groupBy(xs: List[A], f: A => K): Map[K, List[A]]`
  - Partitions `xs` into a map of traversable collections according to a discriminator function.
- `filter(xs: List[A], f: A => Boolean) : List[A]`
  - Takes a predicate and returns all elements that satisfy it
- `scanL(xs: List[A], f: (B, A) => B, init: B) : List[B]`
  - Like `foldL`, but returns a list of all intermediate results
- `zip(xs: List[A], ys: List[B]): List[(A,B)]`
  - Returns an iterable collection formed by iterating over the corresponding items of xs and ys.

Laziness

- Laziness is an evaluation strategy which delays the evaluation of an expression until its value is needed
  - Separating a pipeline construction from its evaluation
  - Not requiring to read datasets in memory: we can process them in lazy-loaded batches
  - Generating infinite collections
  - Optimizing execution plans

```
// Scala LazyList
val fibs: LazyList[Int] = {
  0 #:: 1 #:: fibs.zip(fibs.tail).map{ n =>
    println("Adding " + n._1 + " and " + n._2)
    n._1 + n._2
  }
}
fibs.take(5).foreach(println)
```

```
## 0
## 1
## Adding 0 and 1
## 1
## Adding 1 and 1
## 2
## Adding 1 and 2
## 3
```

Monads

- Monads are the tool FP uses to deal with (side-)effects
  - a design pattern that defines how functions can be used together to build generic types.
  - s a value-wrapping type that:
    - Has an identity function
    - Has a flatMap function, that allows data to be transferred between monad types
- Example monads:
  - Null points: `Option[T]`
  - Exceptions: `Try[T]`
    - `Success[T]`, where T represents the type of the result
    - `Failure[E]`, where E represents the type of error, usually an exception
  - Latency in asynchronous actions: `Future[T]`
- In functional programming exceptions are preferably not used as they break referential transparency. The solution is to return a predefined bogus value
  - Allows errors to silently propagate

- Not applicable to polymorphic code
- Difficult to use the result – requires special policy

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.
- 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.

Iteration

- 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.

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

Traversal

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

```
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 neighbors first, then its children.
  - Depth-first traversal: From any node A, visit its children first, then its neighbors.
- In most programming environments, traversal is implemented by iterators.

Operations

- Operations are transformations, aggregations or cross-referenceing of data stored in data types. All of container data types can be iterated.
- We generally have two types of operations:
  - Element-wise ops apply a function to each individual message. This is the equivalent of map or flatMap in batch systems.
  - Aggregations group multiple events together and apply a reduction (e.g. fold or max) on them.

Element-wise operations

- Conversion: Convert values of type A to type B
  - This is generalized to the map function:
    - 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

Aggregations

- Aggregations apply a combining operator on a traversable sequence to aggregate the individual items into a single result.
- Aggregation is implemented using reduction (or folding). Two variants exist
  - With left reduction, we traverse items from the first to last
  - With right reduction, we traverse items from the last to first
  - The end result of reduceR abd reduceL is the same iff the operation is commutative:
    - An operation `o` is commutative if x `o` y = y `o` x
- Example of aggregation function: calculate the total sum of a list of integers

Grouping

- Grouping splits a sequence of items to groups given a classification function: $groupBy(xs : [A], f : A \to K) : Map[K, [A]]MWhen$

```
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))
```

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

Key value pairs

- 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
- KV stores is the most common format for 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
- Python

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

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

Common operations

Recall that in these scenarios the keys do no not need to be unique:

- `mapValues`: Transform the values part: $mapVal(kv : [(K, V)], f : V \to U) : [(K, U)]n = |V|How$
- `groupByKey`: Group the values for each key into a single sequence. $groupByKey(kv : [(K, V)]) : [(K, [V])]M_{ij}t$
- `reduceByKey`: Combine all elements mapped by the same key into one $reduceByKey(kv : [(K, V)], f : (V, V) \to V) : [(K, V)]V_it - s$

- `join`: Return a sequence containing all pairs of elements with matching keys $join(kv1 : [(K, V)], kv2 : [(K, W)]) : [(K, (V, W))]V_js$
- In functional programming we desire to apply transformations to a separate new instances and keep the original data untouched

KV pair to relation and viceversa

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

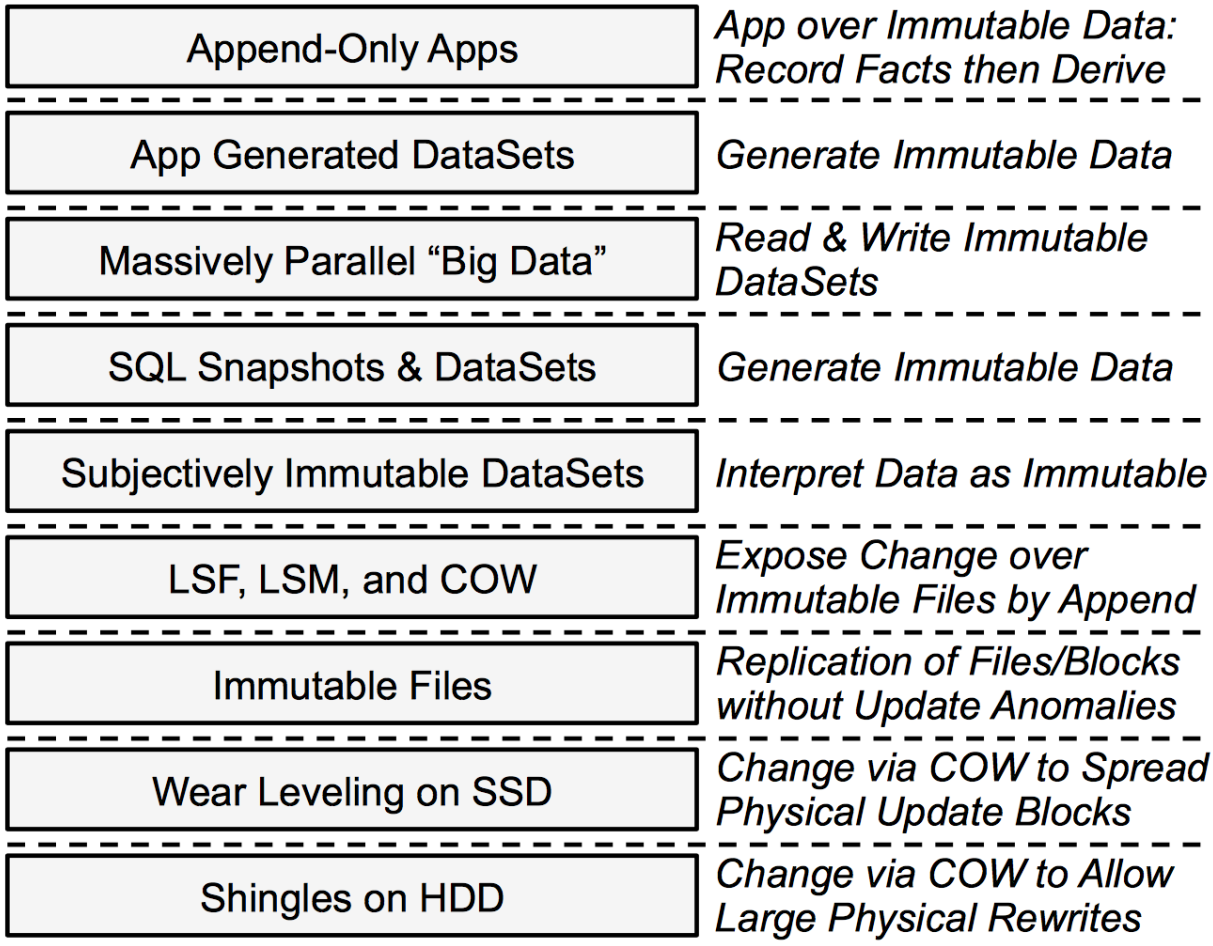
```
val kvpair : Set[Tuple2[Int, Tuple2[Int, String]]] =
  relation.map(x => (x._1, (x._2, x._3)))
```

- KV pair to relation

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

Big data Immutability

- 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.



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/persistent datastructures

- Immutable or persistent data structures always preserve the previous version of themselves when they are modified
- 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
  - Seamlessly persist the data on disk
  - Reason about changes in the data
- They come at a cost of increased memory usage (data is never deleted).

Data structures in Scala

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

Why use Scala?

- It seems that FP is something that can be achieved with Java anyway right?
  - Java is the "assembly" of scala
  - FP is just a set of constraints on how to write code, which could be done within Java
- The reason to use Scala (and other FP languages) is that the syntactic sugar is optimized specifically for this type of programming philosophy/applications
  - You'll see that in big data applications 1 piece of data has to go over many transformations
    - Fetch
    - Clean
    - Transform 1
    - Transform 2
    - ...
    - Transform N
    - Export
    - ...
- Implementing a lot of (simple) transformations (in sperate methods) in Java would result in huge amounts of boilerplate code
- In scala you can apply lots of function compositions in a few lines (and just 1 file instead of having many files for the abstract classes):

```
type WebRequest = HttpRequest => HttpResponse
val = deSerialisePerson: HttpRequest => Person = ???
val createCustomer: Person => Customer = ???
val saveCustomer: Customer => Customer = ???
val serialiseCustomer: Customer => HttpResponse = ???

val registerCustomer: WebREquest =
  deserialise Person andThen
    createCustomer andThen
    saveCustomer andThen
    serialiseCustomer
```

- This syntax works very well for http requests, data pipelines, etc.
  - Because each function is completely independent
  - You can test them without mocking dependencies

Distributed Systems

- A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages.
  - Parallel systems use shared memory
  - Distributed systems use no shared components
- Distriubted systems offer:
  - Scalability
    - Moore's law: The number of transistors on a single chip doubles about every two year.
    - The advancement has slowed since around 2010.
    - Distribution provides massive performance.
  - Distribution of tasks and collaboration
  - Reduced latency
  - Fault tolerance
  - Mobility

Characteristics

- Computational entities each with own memory
  - Need to synchronize distributed state
- Entities communicate with message passing
- Each entity maintains parts of the complete picture
- Need to tolerate failure
- 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

- 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

Problems

- Partial failures: Some parts of the system may fail nondeterministically, while other parts work fine.
- Unreliable networks: Distributed systems communicate over unreliable networks.
- Unreliable time: Time is a universal coordination principle. However, we cannot use time determine order.
- No single source of truth, different opinions: Distributed systems need to co-ordinate and agree upon a (version of) truth.

Asynchronous vs syncrhonous systems

- Synchronous system: Process execution speeds or message delivery times are bounded. In a synchronous system, we can have:
  - Timed failure detection
  - Time based coordination
  - Worst-case performance
- Asynchronous system: No assumptions about process execution speeds or message delivery times are made.
  - Upon waiting for a response to a requests in an asynchronous system, it is not possible to distinguish whether:
    - the request was lost
    - the remote node is down
    - the response was lost
  - Timeouts is a fundamental design choice in asynchronous networks: Ethernet, TCP and most application protocols work with timeouts.
- Purely synchronous systems only exist in theory.
- Most distributed systems use some form of asynchronous networking to communicate.

Time is essential

- In a distributed system, time is the only global constant nodes can rely on to make distributed decisions on ordering problems.
- When do we need to order?
  - Sequencing items in memory
  - Mutual exclusion of access to a shared resource
  - Encoding history ("happens before" relationships)
  - Transactions in a database
  - Consistency of distributed data
  - Debugging (finding the root cause of a bug)
- 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 centralized servers. (e.g. System.getCurrentTimeMillis).
    - (Synced time from online server)
  - Monotonic clocks: Those clocks only move forward. (e.g. System.nanoTime)
    - (Harward time mantained by the OS)
    - They are (usually!) good for determining order within a node, but each node only has its own notion of time.

Logical time

- Idea: Instead of using the precise clock time, capture the events relationship between a pair of events
- Based on causality: If some event possibly causes another event, then the first event happened-before the other
- Order: A way of arranging items in a set so that the following properties are maintained.
  - Strict partial order:
    - `Irreflexivity: ∀a.¬a<a (items not comparable with self)`

        - `Transitivity: if a≤b and b≤c then a≤c`
        - `Antisymmetry: if a≤b and b≤a a=b`
    - Strict total order:
        - `An additional property: ∀a,b,a≤b∨b≤a∨a=b`
- Lamport introduced happens-before relation to capture dependencies between events
    - It is a strict partial order: it is irreflexive, antisymmetric and transitive.
- Two events not related to happened-before are concurrent.

Distributed decision making

- Nodes in distributed systems cannot know anything for sure
    - 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)
- Consensus for distributed decision making:
    - Resource allocation
    - Committing a transaction
    - Synchronizing state machines
    - Leader election
    - Atomic broadcasts
- Approximate solution to 2 generals problem:
    - 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.
- Byzantine generals solution: Fault tolerant consensus
    - `PBFT [6]` - Byzantine fault tolerant consensus with at least 3f+1 nodes with f traitors
    - `Paxos [7]`, `Raft [8]` - Crash fault tolerant consensus with at least 2f+1 nodes with f faulty nodes
- Conesus protocol properties
    - 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.

Distributed databases

- A distributed database is a distributed system designed to provide read/write access to data, using the relational or other format
- Important topics
    - 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: Units of work that group several reads and writes to be performed together in the database

Replication

- With replication, we keep identical copies of the data on different nodes.
- Why do we need to replicate data?
    - Data scalability: To increase read throughput, by allowing more machines to serve read-only requests
    - Geo-scalability: To have the data (geographically) close to the clients

- Fault tolerance: To allow the system to work, even if parts of it are down

Replication Architecture

- In a replicated system, we have two node roles:
  - Leaders or Masters: Nodes that accept write queries 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 leader accepts writes, which are distributed to followers
  - Multi-leader or master-master: Multiple leaders accept writes, keep themselves in sync, then update followers
  - Leaderless replication All nodes are peers in the replication network
- Postgres and Oracle use WAL-based (write-ahead log) replication.
  - WAL-based replication writes all changes to the leader WAL and also to the followers. The followers apply the WAL entries to get consistent data.
- MongoDB and MySQL use a stream of logical updates for each update to the WAL. Logical updates can be:
  - For new records, the values that were inserted
  - For deleted records, their unique id
  - For updated records, their id and the updated values
- Process for creating a replica:
  - Take a consistent snapshot from the leader
  - Ship it to the replica
  - Get an id to the state of the leader's replication log at the time the snapshot was created
  - Initialize the replication function to the latest leader id
  - The replica must retrieve and apply the replication log until it catches up with the leader
- How to avoid or resolve write conflicts?
  - One leader 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 conflict resolution policies:
    - last-write-wins policy to order writes by timestamp (may lose data)
    - report conflict to the application and let it resolve it (same as git or Google docs)
    - use conflict-free data types with specific conflict resolution logics

Brewer's CAP theorem

- (Strong) Consistency: – All nodes in the network have the same (most recent) value
- Availability: – Every request to a non-failing replica receives a response
- Partition tolerance: The system continues to operate in the existence of component or network faults

Partitioning

- Partitioning breaks whole the dataset into fractions and distributes them to different hosts, also known as sharding.
- Why do it? 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:
  - 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).

Transactions

- Blocks of operations (reads and writes), that either succeed or fail, as a whole.
- started with the first SQL database, System R, ACID scheme:
  - Atomicity: The transaction either succeeds or fails; in case of failure, outstanding writes are ignored (all or nothing).
  - Consistency: Any transaction will bring the database from one valid state to another.
  - Isolation: Concurrent execution of transactions do not interfere and effect each other.
  - Durability: Once a transaction has been committed, it will remain so.
- 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.

Distributed filesystems

- File systems determine how data is stored and retrieved. A file system keeps track of the following data items:
  - Files, where the data we want to store are.
  - Directories, which group files together
  - Metadata, such as file length, permissions and file types
- The primary job of the filesystem is to make sure that the data is always accessible and in tact. To maintain consistency, most modern filesystems use a log (remember databases!).
- Common filesystems are EXT4, NTFS, APFS and ZFS
- A distributed filesystem is a file system which is shared by being simultaneously mounted on multiple servers. Data in a distributed file system is partitioned and replicated across the network. Reads and writes can occur on any node.
- The Google Filesystem paper kicked-off the Big Data revolution. Why did they need it though?
  - Hardware failures are common (commodity hardware)
  - Files are large (GB/TB) and their number is limited (millions, not billions)
  - Two main types of reads: large streaming reads and small random reads
  - Workloads with sequential writes that append data to files
  - Once written, the files are seldom modified, they are read often sequentially
  - High sustained bandwidth trumps low latency
- Large-scale distributed file systems:
  - Google File System (GFS)
  - Hadoop Distributed File System (HDFS)
  - CloudStore
  - Amazon Simple Storage Service (Amazon S3)

GFS storage model

- A single file can contain many objects (e.g. Web documents)
- Files are divided into fixed size chunks (64MB) with unique identifiers, generated at insertion time.
  - Disk seek time small compared to transfer time
  - A single file can be larger than a node's disk space
  - Fixed size makes allocation computations easy
- Files are replicated (≥3) across chunk servers.

- The master maintains all file system metadata (e.g. mapping between file names and chunk locations)
- Chunkservers store chunks on local disk as Linux files
  - Reading & writing of data specified by the tuple (chunk_handle, byte_range)
- Neither the client nor the chunkserver cache file data

GFS operation

- The master does not keep a persistent record of chunk locations, but instead queries the chunk servers at startup and then is updated by periodic polling.
- GFS is a journaled filesystem: all operations are added to a log first, then applied. Periodically, log compaction creates checkpoints. The log is replicated across nodes.
- If a node fails:
  - If it is a master, external instrumentation is required to start it somewhere else, by rerunning the operation log
  - If it is a chunkserver, it just restarts
- Chunkservers use checksums to detect data corruption
- The master maintains a chunk version number to distinguish between up-to-date and stale replicas (which missed some mutations while its chunkserver was down
  - Before an operation on a chunk, master ensures that version number is advanced

HDFS - Haddop FileSystem

- HDFS started at Yahoo as an open source replica of the GFS paper, but since v2.0 it is different system.
- The main difference with GFS is that HDFS is a user-space filesystem written in Java.
- HDFS looks like a UNIX filesystem, but does not offer the full set of operations.

```
# List directory
$ hadoop fs ls /
Found 7 items
drwxr-xr-x   - gousiosg sg          0 2017-10-04 08:23 /audioscrobbler
-rw-r--r--   3 gousiosg sg 1215803135 2017-10-04 08:25 /ghtorrent-logs.txt
-rw-r--r--   3 gousiosg sg   66773425 2017-10-04 08:23 /imdb.csv
-rw-r--r--   3 gousiosg sg     198376 2017-10-23 12:39 /important-repos.csv
-rw-r--r--   3 gousiosg sg     611300 2017-10-04 08:24 /odyssey.mb.txt
-rw-r--r--   3 gousiosg sg  388422973 2017-10-03 15:40 /pullreqs.csv

# Create a new file
$ dd if=/dev/zero of=foobar bs=1M count=100

# Upload file
$ hadoop fs -put foobar /
-rw-r--r--   3 gousiosg sg  104857600 2017-11-27 15:42 /foobar
```

Akka actor programming

- Documentation
- Akka is a open-source library for Scala and Java
  - To create reactive, distributed, parallel and resilient concurrent applications in Scala or Java
- Nowadays most databases are distributed

Actor Concurrency

- Actor is an independent computation units that keep a copy of the state (like a github repo user)
- But it's non-blocking, you send a message and dgaf what's going on elsewhere. There are no conflicts by design.

- Messages are sent asynchronously
- Each actor has a mailbox (queue)
- Each actor is like a thread pulling on its mailbox running them one after each other

Example program

```
object HelloWorld {
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])
  final case class Greeted(whom: String, from: ActorRef[Greet])

  def apply(): Behavior[Greet] = Behaviors.receive { (context, message) =>
    context.log.info("Hello {}!", message.whom)
    message.replyTo ! Greeted(message.whom, context.self)
    Behaviors.same
  }
}
```

- ! denotes asyncrhonous message (send and forget)

```
object HelloWorldBot {
  def apply(max: Int): Behavior[HelloWorld.Greeted] = { bot(0, max) }

  private def bot(greetingCounter: Int, max: Int): Behavior[HelloWorld.Greeted] =
    Behaviors.receive { (context, message) =>
      val n = greetingCounter + 1
      context.log.info2("Greeting {} for {}", n, message.whom)
      if (n == max) {
        Behaviors.stopped
      } else {
        message.from ! HelloWorld.Greet(message.whom, context.self)
        bot(n, max)
      }
}}
```

```
object HelloWorldMain {
  final case class SayHello(name: String)

  def apply(): Behavior[SayHello] = Behaviors.setup { context =>
    val greeter = context.spawn(HelloWorld(), "greeter")
    Behaviors.receiveMessage { message =>
      val replyTo = context.spawn(HelloWorldBot(max = 3), message.name)
      greeter ! HelloWorld.Greet(message.name, replyTo)
      Behaviors.same }
} }
```

Running the actor system

```
val system: ActorSystem[HelloWorldMain.SayHello] = ActorSystem(HelloWorldMain(), "hello"
system ! HelloWorldMain.SayHello("World")
system ! HelloWorldMain.SayHello("Akka")
```

```
14:26:33.904 [hello-akka.actor.default-dispatcher-6] INFO HelloWorld$ - Hello World!
14:26:33.905 [hello-akka.actor.default-dispatcher-6] INFO HelloWorld$ - Hello Akka!
14:26:33.906 [hello-akka.actor.default-dispatcher-5] INFO HelloWorldBot$ - Greeting 1 fo
14:26:33.906 [hello-akka.actor.default-dispatcher-3] INFO HelloWorldBot$ - Greeting 1 fo
14:26:33.906 [hello-akka.actor.default-dispatcher-6] INFO HelloWorld$ - Hello World!
14:26:33.906 [hello-akka.actor.default-dispatcher-5] INFO HelloWorldBot$ - Greeting 2 fo
14:26:33.906 [hello-akka.actor.default-dispatcher-6] INFO HelloWorld$ - Hello Akka!
14:26:33.906 [hello-akka.actor.default-dispatcher-5] INFO HelloWorldBot$ - Greeting 2 fo
14:26:33.906 [hello-akka.actor.default-dispatcher-6] INFO HelloWorld$ - Hello World!
```

```
14:26:33.906 [hello-akka.actor.default-dispatcher-5] INFO HelloWorldBot$ - Greeting 3 fo
14:26:33.906 [hello-akka.actor.default-dispatcher-6] INFO HelloWorld$ - Hello Akka!
14:26:33.906 [hello-akka.actor.default-dispatcher-6] INFO HelloWorldBot$ - Greeting 3 fo
```

Fault tolerance

- Fault-tolerance is provided via hierarchical supervision
  - Actors can create child actors which they monitor for failure
  - They can restart actors if necessary

```
val behavior = Behaviors.supervise(behavior).onFailure[IllegalStateException](Supervisor
val ref = context.spawn(behavior, "my-actor")
context.watchWith(ref, ActorTerminated)
```

Types

- Classic Akka API is untyped
  - Actors are untyped and can recive and send any type of messages

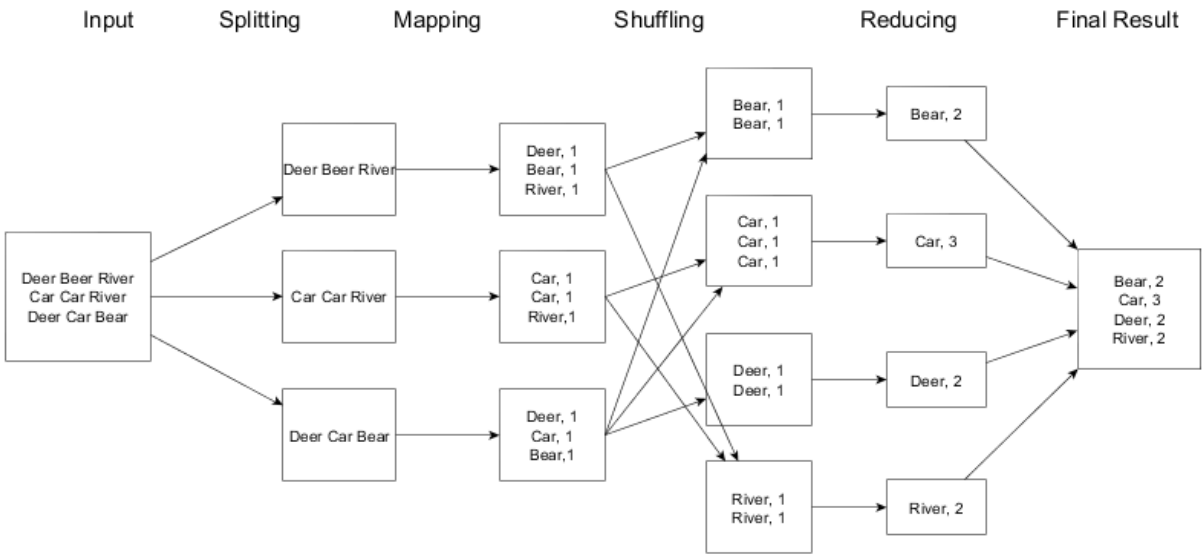
```
class HelloWorld extends Actor {
  def receive: Receive = {
    case Greet (whom, replyTo) =>
      context.log.info("Hello {}!", message.whom)
      replyTo ! Greeted(message.whom, context.self)
}}
```

- Akka Typed is the new Akka Actor API
  - Protocol-first approach (programmer defines the protocol in terms of exchanged messages)

```
object HelloWorld {
  def apply(): Behavior[Greet] = Behaviors.receive { (context, message) =>
    . . .
}}
```

MapReduce

- Takes a set of input key/value pairs, and produces a set of output key/value pairs.
  - The computation is expressed using two functions: Map and Reduce.
  - Map takes an input pair and produces a set of intermediate key/value pairs
- Reduce takes an intermediate key, and a set of values for that key. It merges together these values



Full example program

```
//#full-example
package com.example


import akka.actor.typed.ActorRef
import akka.actor.typed.ActorSystem
import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors
import com.example.GreeterMain.SayHello

//#greeter-actor
object Greeter {

  /*
  Each actor defines a type T for the messages it can receive.
    - Here Greeter defines Behaviour[Greet]
  messages are the Actor's public API
  Messages should be immutable, since they are shared between different threads.
  This example is a simple request-reply protocol but Actors can model arbitrarily compl
   */

  // Case classes and case objects make excellent messages since they are immutable and
  // It is a good practice to put an actor's associated messages in its object.
  // This makes it easier to understand what type of messages the actor expects and hand
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])
  final case class Greeted(whom: String, from: ActorRef[Greet])

  //Behaviour for Greet command: Upon receiving the Greet command responds with a Greete
  def apply(): Behavior[Greet] = Behaviors.receive { (context, message) =>
    context.log.info("Hello {}!", message.whom)
    //#greeter-send-messages:
    message.replyTo ! Greeted(message.whom, context.self) // reply from the Greeter acto
    /* the ! operator (pronounced "bang" or "tell"). It is an asynchronous operation tha
    Since the replyTo address is declared to be of type ActorRef[Greeted], the compiler
     */



    /*
    we don't need to update any state, so we return Behaviors.same,
    which means the next behavior is "the same as the current one".
     */
    Behaviors.same
  }
```

```
}

//#greeter-bot
object GreeterBot {
  // Greeted (reply) behaviour: sends a number of additional greeting messages
  // and collect the replies until a given max number of messages have been reached.
  def apply(max: Int): Behavior[Greeter.Greeted] = {
    bot(0, max)
  }

  private def bot(greetingCounter: Int, max: Int): Behavior[Greeter.Greeted] =
    Behaviors.receive { (context, message) =>
      val n = greetingCounter + 1
      context.log.info("Greeting {} for {}", n, message.whom) //logs (collect) reply
      if (n == max) { //until max number of messages is reached
        Behaviors.stopped
      } else {
        message.from ! Greeter.Greet(message.whom, context.self)
        bot(n, max) //recursion as loop
      }
    }
}

//#greeter-main: The guardian actor that bootstraps everything.

object GreeterMain {

  final case class SayHello(name: String)
//command to the GreeterMain to start the greeting process
  def apply(): Behavior[SayHello] = {

    //bootstrap the application
    Behaviors.setup { context =>
      //#create-actors
      val greeter = context.spawn(Greeter(), "greeter")
      /*
      In Akka you can't create an instance of an Actor using the new keyword. Instead, y
       */

      Behaviors.receiveMessage { message =>
        //#create-actors
        val replyTo = context.spawn(GreeterBot(max = 3), message.name)
        //#create-actors
        greeter ! Greeter.Greet(message.name, replyTo) //Greet command
        Behaviors.same
      }
    }
  }
}

//#main-class
object AkkaQuickstart extends App {
  //#actor-system
  ////An ActorSystem is the intial entry point into Akka. Usually only one ActorSystem i
  //An ActorSystem has a name and a guardian actor. The bootstrap of your application is
  val greeterMain: ActorSystem[GreeterMain.SayHello] = ActorSystem(GreeterMain(), "AkkaQ
  //#actor-system

  //#main-send-messages
  greeterMain ! SayHello("Charles") //Only behaviour that we have defined in main
  //#main-send-messages
}
```

Output:

```
/home/sergio/.jdks/openjdk-16.0.2/bin/java -javaagent:/home/sergio/ij/lib/idea_rt.jar=39(
SLF4J: A number (1) of logging calls during the initialization phase have been intercept
SLF4J: now being replayed. These are subject to the filtering rules of the underlying lo
SLF4J: See also http://www.slf4j.org/codes.html#replay
[2021-10-04 11:14:44,001] [INFO] [akka.event.slf4j.Slf4jLogger] [AkkaQuickStart-akka.act(
[2021-10-04 11:14:44,052] [INFO] [com.example.Greeter$] [AkkaQuickStart-akka.actor.defau
[2021-10-04 11:14:44,054] [INFO] [com.example.GreeterBot$] [AkkaQuickStart-akka.actor.de
[2021-10-04 11:14:44,054] [INFO] [com.example.Greeter$] [AkkaQuickStart-akka.actor.defau
[2021-10-04 11:14:44,055] [INFO] [com.example.GreeterBot$] [AkkaQuickStart-akka.actor.de
[2021-10-04 11:14:44,055] [INFO] [com.example.Greeter$] [AkkaQuickStart-akka.actor.defau
[2021-10-04 11:14:44,055] [INFO] [com.example.GreeterBot$] [AkkaQuickStart-akka.actor.de
```

Spark

Parallelism

- Parallelism is about speeding up computations by using multiple processors
  - Task parallelism: Different computations performed on the same data (Do different tasks on parallel, same data)
  - Data parallelism: Apply the same computation on dataset partitions (Do the same task on parallel data)
    - When processing big data, data parallelism is used to move computations close to the data, instead of moving data in the network.
    - 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

Map/Reduce

- General computation framework based on functional programming
- It assumes that data exists in a key -> value data structure
- Reduce tasks work on each key separately and combine all the values associated with a specific key.

Hadoop

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

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.

Spark and RDDs

- Spark is an open source cluster computing framework
  - Automates distribution of data and computations on a cluster of computers
  - Provides a fault-tolerant abstraction to distributed datasets
  - Based on functional programming primitives
  - Provides two abstractions to data
    - List-like (Resilient distributed datasets)
    - Table-like (Datasets)

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

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

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.

Counting words with Spark

```
// 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 occurrences of same pairs
  .sortBy(x => x._2, false)            // Sort by number of occurrences
  .take(50)                            // Take the first 50 results
  .foreach(println)
```

How to create an RDD?

RDDs can only be created in the following 3 ways

- Reading data from external sources

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

- Convert a local memory dataset to a distributed one

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

- Transform an existing RDD

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

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.
  - Other operations include `reduceByKey`, `aggregateByKey` and I assume anythingByKey
- We can create Pair RDDs by applying an indexing function, using keyBy function, or by grouping records:

```
val rdd = sc.parallelize(List("foo", "bar", "baz")) // 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.keyBy(x => x.toLowerCase.head) // RDD[(Char, String)]
pairRDD2.collect // Array((f,foo), (b,bar), (b,baz))

val pairRDD3 = rdd.groupBy(x => x.charAt(0))  // RDD[(Char, Iterable(String))]
pairRDD3.collect // Array((b,CompactBuffer(bar, baz)), (f,CompactBuffer(foo)))
```

Pair RDD examples: groupByKey and reduceByKey

```
val odyssey = sc.textFile("sample.txt").flatMap(_.split(" "))
val words = odyssey.flatMap(_.split(" ")).map(c => (c, 1))

//Word count using groupByKey:

val counts = words.groupByKey() // RDD[(String, Iterable[Int])]
  .map(row => (row._1, row._2.sum)) // RDD[(String, Int)]
  .collect() // Array[(String, Int)]

//Word count using reduceByKey:

val counts2 = words.reduceByKey(_ + _) // RDD[(String, Int)]
  .collect()  // Array[(String, Int)]
```

Pair RDD example: aggregateByKey

```
//Word count using aggregateByKey:

  val counts3 = words.aggregateByKey(0)(_ + _, _ + _) // RDD[(String, Int)]

//How can we count the number of occurrences of part of speech elements?

object PartOfSpeech {
  sealed trait Word
  case object Verb extends Word
  case object Noun extends Word
  case object Article extends Word
  case object Other extends Word
}

def partOfSpeech(w: String): Word = ???
```

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

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) // RDD[(Int, Person)]
val pairAs = as.keyBy(_.person_id) // RDD[(Int, Addr)]

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]))]

Partitioning

Spark supports 3 types of partitioning schemes:

```
Default partitioning: Splits in equally sized partitions, without knowing the underlying
```

Extended partitioning is only configurable on Pair RDDs.

```
Range partitioning: Takes into account the natural order of keys to split the dataset in

Hash partitioning: Calculates a hash over each item key and then produces the modulo of
```

`key.hashCode() % numPartitions`

Shuffling

- The process of re-arranging data so that similar records end up in the same partitions is called shuffling.
  - Shuffling is needed when operations need to calculate results using a common characteristic (e.g. a common key), as this data needs to reside on the same physical node.

- 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!

Debugging RDD lineage

- RDDs contain information on how to compute themselves, including dependencies to other RDDs.
- Lineage information allow an RDD to be traced to its ancestors.

```
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))
```

```
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 []
```

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 emory-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.

Persistence storage levels

Storage level	Meaning
`MEMORY_ONLY`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
`MEMORY_AND_DISK`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
`MEMORY_ONLY_SER` \ (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). More space-efficient than deserialized objects but more CPU-intensive to read.

Storage level	Meaning
`MEMORY_AND_DISK_SER` (Java and Scala)	Similar to `MEMORY_ONLY_SER`, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
`DISK_ONLY`	Store the RDD partitions only on disk.

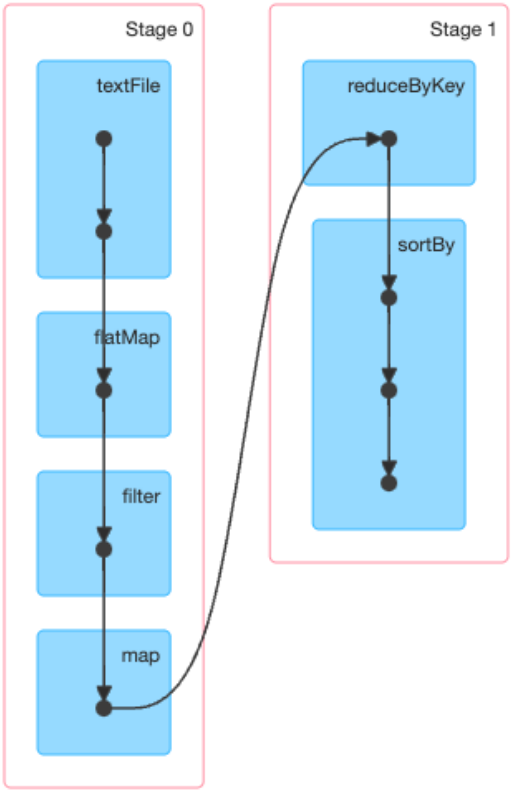
Spark applications

- 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
  - 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 reshuffle/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.
- when it finishes, the cluster manager frees the resources.
  - The cluster manager starts as many jobs as the data partitions.

Job graph

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

```
val result = sc.textFile("sample.txt")
  .flatMap(_.split(" "))
  .filter(x => x.length > 1)
  .map(x => (x, 1))
  .reduceByKey((a,b) => a + b)
  .sortBy(_._1)
```



Start application

- 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
```

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

Broadcasts

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks so that they do not have to retransfer them on every shuffle.

Accumulators

- 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.

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

Sparq SQL

- RDDs only see binary blobs with an attached type
- Databases can do many optimizations because they know the data types for each field
- 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.
- 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 ≈ 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.
- If a SparkContext object exists, it is straightforward to get a SparkSession

```
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()
```

Creating Data Frames and Datasets

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

```
import spark.implicits._
val df = rdd.toDF("name", "id", "address")
```

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

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

- From RDDs, with manual schema definition

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

- 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)
```

- 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:

```
//java?
df["team_size"]
df.team_size
```

```
//or in Scala
df("team_size")
$"team_size" //scala only
```

- Columns are defined by expressions. The API overrides 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()
```

Documentation

- RDD Documentation
- Spark documentation

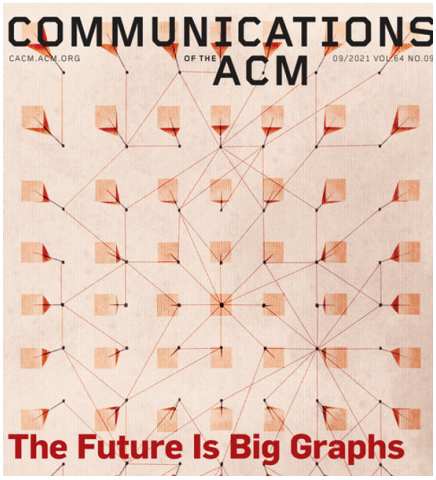
Graph processing

Graph Processing

Georgios Gousios and Burcu Kulahcioglu Ozkan

08 October 2021

The Future is Big Graphs



CACM, 09/2021

"*We are witnessing a unprecedented growth of interconnected data, which underscores the vital role of graph processing in our society.*"

Processing Graphs

Graphs and other forms of hierarchical data structures appear every time a system models a *dependency relationship*. Common big graphs are:

- The *social graph* in social networking applications
- The *web graph* of linked pages

- The *dependency graph* in (software) package ecosystems

Graph representations in short

A graph (G) comprises *nodes* (V) and *edges* (E). Both can carry metadata. We represent graphs as:

- *Adjacency matrix*: An $n \times n$ matrix M ($n = |V|$) where a non-zero element M_{ij} indicates and edge from V_i to V_j
- *Adjacency list*: A `List[(V, List[V])]` where each tuple represents a node and its connections to other nodes.
- *Edge list*: A `List[(V, V)]` of node pairs that represents and edge

Graph (sub-)structures

- *Graph components*: Subgraphs in which any two vertices are connected to each other by paths.
- *Strongly connected component*: The largest sub-graph with a path from each node to every other node
- *Triangles* or *polygons*: A triangle occurs when one vertex is connected to two other vertices and those two vertices are also connected.
- *Spanning trees*: A sub-graph that contains all nodes and the minimum number of edges

Typical graph algorithms

- *Traversal*: Starting from a node, find all connected nodes
  - Depth-first: Recursively follow all graph edges until all reachable nodes are visited
  - Breadth-first: Follow graph edges per level; maintain a work-queue of visited nodes
- *Node importance*: Calculate the importance of a node relative to other nodes
  - Centrality measures or PageRank
- *Shortest paths*
  - Dijkstra's algorithm or 'traveling salesman' approaches

Typical graph applications

- Exploring the structure and evolution of communities or of systems of systems
  - WWW
  - Disease spreading / epidemiology
  - Software libraries
- Link prediction:
  - Recommending friends
  - Recommending pages
- Community detection: sub-graphs with shared properties

Approaches for graph processing

To process graphs, we can:

- Use SQL databases and recursive queries
- Use a graph database

For really big graphs, our options are somewhat limited

- Efficiently compress the graphs so that they fit in memory
- Use a message-passing architecture, like the bulk synchronous parallel model

Typical graph applications

Not all applications need to process billions of nodes and trillions of edges. For small to medium sized graphs (< 500M edges), existing tools can go a long way.

Graphs in SQL databases

```
CREATE TABLE nodes (
    id INTEGER,
    metadata ...
)

CREATE TABLE edges (
    src INTEGER,
    target INTEGER,
    metadata ...,
    CONSTRAINT src_fkey
        FOREIGN KEY (src) REFERENCES nodes(id),
    CONSTRAINT target_fkey
        FOREIGN KEY (target_id) REFERENCES nodes(id)
)
```

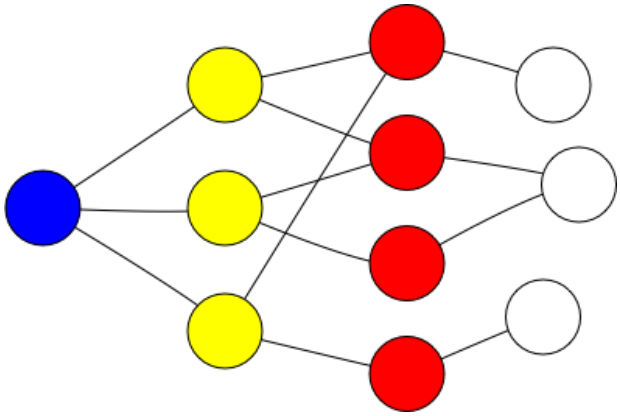
We model graphs as node pairs. Nodes and edges have metadata.

SQL-based graph traversals

```
WITH RECURSIVE transitive_closure (a, b, distance, path) AS
(
    SELECT a, b, 1 as distance, a || '->' || b AS path
    FROM edges

  UNION ALL

    SELECT tc.a, e.b, tc.distance + 1, tc.path || '->' || e.b
    FROM edges e
    JOIN transitive_closure tc ON e.a = tc.b
    WHERE a.metadata = ...   -- Traversal filters on nodes/edges
        and tc.path NOT LIKE '%->' || e.b || '->%'
)
SELECT * FROM transitive_closure
```

Recursive queries have a starting clause that is called on and a recursion clause

Example: Friend recommendation



A simple social network

Given that we (blue node) are direct friends with the yellow nodes, we could recommend second level (red) friends as potential new connections.

Recommending friends with SQL

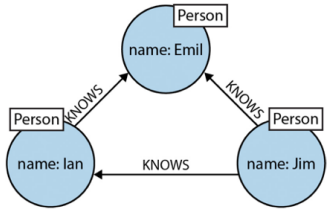
```
WITH RECURSIVE transitive_closure (a, b, distance, path) AS
(
    -- Find the yellow nodes
    SELECT a, b, 1 as distance, a || '->' || b AS path
    FROM edges
    WHERE a = src -- the blue node

  UNION ALL
    -- Find the red nodes
    SELECT tc.a, e.b, tc.distance + 1, tc.path || '->' || e.b
    FROM edges e
    JOIN transitive_closure tc ON e.a = tc.b
    WHERE tc.path NOT LIKE '%->' || e.b || '->%'
      AND tc.distance < 2 -- don't recurse into white nodes
)
SELECT a, b FROM transitive_closure
GROUP BY a, b
HAVING MIN(distance) = 2 -- only report red nodes
```

The base expression will find all directly connected nodes, while the second will recurse into their first level descendants.

Graph databases

Graph databases are specialized RDBMs for storing recursive data structures and support CRUD operations on them, while maintaining transactional consistency (ACID or otherwise).

The most commonly used language for graph databases is Cypher, the base language for Neo4J.




```
(emil:Person {name:'Emil'})
  <-[:KNOWS]-(jim:Person {name:'Jim'})
  -[:KNOWS]->(ian:Person {name:'Ian'})
  -[:KNOWS]->(emil)
```

Find mutual friends of a user named Jim:

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b:Person)-[:KNOWS]->(c:Person), (a)-[:KNOWS]->(
RETURN b, c
```

Big Graphs

Graphs are an inherently recursive data structures, which means that computations may have dependencies to previous computation steps (and thus they are not trivially parallelizable).

Traditional frameworks are not well-suited for processing graphs.

- Poor locality of memory accesses
  - Access patterns not very suitable for distribution

    - Further complicated due to latency issues
- Little work to be done per node
    - Applications mostly care about the edges

Computation example: PageRank

PageRank is a centrality measure based on the idea that nodes are important if multiple important nodes point to them. For node p_i, its Page rank is recursively defined as

$$PR(p_i;0) = \frac{1}{N}$$

$$PR(p_i;t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j;t)}{L(p_j)}$$

where d is a damping factor (usually set 0.85) and $M(p_i)$ are the nodes pointing to p_i We notice that each node updates other nodes by propagating its state.

Simplified PageRank on Spark

```
val links: RDD[(V,List(E))] = ....cache()
var ranks = links.mapValues(v => 1.0)

for (i <- 1 to iters) {
  val contribs = links.join(ranks).values.flatMap {
    case (links, rank) =>
      val size = links.size
      links.map(url => (links, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

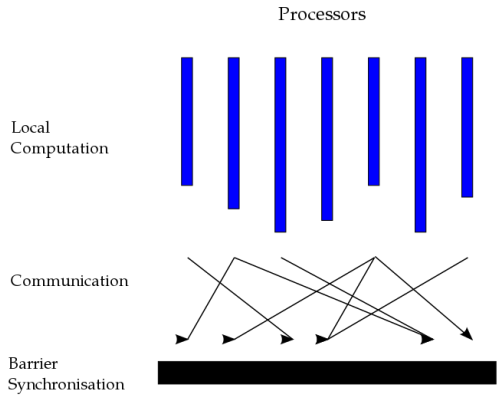
The computation is *iterative* and *side-effecting* and therefore non-parallelizable. To make it side-effect free, we need to write each step of the computation to external storage.

The bulk synchronous parallel model

The BSP model is a general model for parallel algorithms.

It assumes that a system has:

- multiple processors with fast local memory
- pair-wise communication between processors
- a barrier implementation (hardware or other) to synchronize super steps



The BSP model

BSP supersteps

BSP computation is organized in *supersteps*. A superstep comprises three phases:

- Local execution: Processors use own memory to perform computations on local data partitions
- Data exchange / remote communication: Exchange of data between processes
- Barrier synchronization: Processes wait until all processes finished communicating

Pregel: Using BSP to process graphs

Pregel (by Google) is a distributed graph processing framework.

Pregel computations impose a BSP structure on program execution:

- Computations consist of a sequence of supersteps
- In a superstep, the framework invokes a user-defined function for each vertex
- Function specifies behaviour at a single vertex (V) and a single superstep (S)
  - it can read messages sent to V in superstep ($S-1$)
  - it can send messages to other vertices that will be read in superstep ($S+1$)
  - it can modify the state of V and its outgoing edges

Open source implementations: Apache Giraph and GraphX.

Vertex centric approach

Pregel is a vertex-centric graph processing model: the algorithm iterates over vertices

- Reminiscent of MapReduce
  - User (i.e. algorithm developer) focus on a local action
  - Each vertex is processed independently
- By design: well suited for a distributed implementation
  - All communication is from superstep S to (S+1)
  - No defined execution order within a superstep
  - Free of deadlocks and data races

Algorithm termination

BSP programs run until the programs stop themselves. Termination works as follows

- Superstep 0: all vertices are active
- All active vertices participate in the computation at each superstep
- A vertex deactivates itself by voting to halt

- No execution in subsequent supersteps
- A vertex can be reactivated by receiving a message

Roles in a Pregel cluster

Graphs are stored as adjacency lists, partitioned (using hash partitioning) and distributed using a network filesystem

- *Leader*: Maintains a mapping between data partitions and cluster node. Implements the BSP barrier
- *Worker*: For each vertex, it maintains the following *in memory*:
  - Adjacency list
  - Current calculation value
  - Queue of incoming messages
  - State (active / inactive)

The worker applies all computationally intensive operations.

A Pregel superstep

1. Workers *combine* incoming messages for all vertices.
   - The combinator function updates the vertex state
2. If a termination condition has been met, the vertex votes to exclude itself for further iterations
3. (Optional) The vertex updates a global aggregator
4. Message passing:
   - If receiving vertex is local: update its message queue
   - Else wrap messages per receiving node and send them in bulk

Spark GraphX

GraphX is a new component in Spark for graphs and graph-parallel computation. It provides a variant of Pregel's API for developing vertex-centric algorithms.

Spark uses its underlying fault tolerance, check pointing, partitioning and communication mechanisms to store the graph. Halting is determined by examining if the vertex is sending / receiving messages.

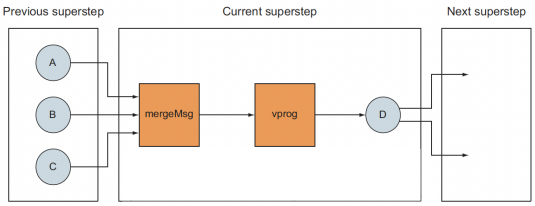
GraphX allows for operating on the underlying data structures as both as a *graph* using graph concepts and processing primitives, and also as separate *collections of edges and vertices* that can be transformed using fp primitives.

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  . . .
}
```

Pregel API in Spark

```
def pregel[A](
    // Initialization message
    initialMsg: A,
    // Max super steps
    maxIter: Int = Int.MaxValue,
    activeDir: EdgeDirection = EdgeDirection.Out,
    // Program to update the vertex
```

```
    vprog: (VertexId, VD, A) => VD,
    // Program to determine edges to send a message to
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
    // Program to combine incoming messages
    mergeMsg: (A, A) => A
) : Graph[V, E]
```



Processing a superstep

PageRank with Pregel/Spark

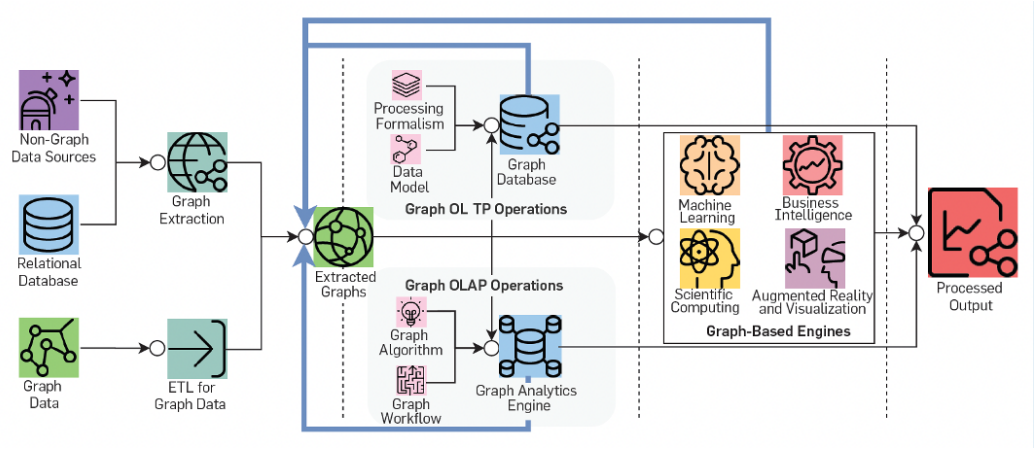
```
val pagerankGraph: Graph[Double, Double] = graph
  .mapVertices((id, attr) => 1.0) // Initial Pagerank for nodes

def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =
  resetProb + (1.0 - resetProb) * msgSum
def sendMessage(id: VertexId, edge: EdgeTriplet[Double, Double]): Iterator[(VertexId, Do
  Iterator((edge.dstId, edge.srcAttr * edge.attr))
def messageCombiner(a: Double, b: Double): Double = a + b
val initialMessage = 0.0

// Execute Pregel for a fixed number of iterations.
Pregel(pagerankGraph, initialMessage, numIter)(
  vertexProgram, sendMessage, messageCombiner)
```

From Pregel.scala in Apache Spark

Fault tolerance

- The fault tolerance model is reminiscent of Spark.
- Periodically, the leader instructs the workers to save the state of their in-memory data to persistent storage
- Worker failure detected through keep-alive messages the leader issues to workers
- In case of failure, the leader reassigns graph partitions to live workers; they reload their partition state from the most recently available checkpoint

Data pipeline for graph processing



Ongoing research for: Abstractions, ecosystems, and performance

Links

- Graphs in the Database
- Graph database example from Neo4j blog on SQL vs. Cypher Query Languages
- GraphX - Graphs on Spark
- The image for Pregel API superstep processing is from [1].
- A comparison of state of the art graph processing systems
- Recommending items to a billion people
- The image for data pipeline for graph processing is from the article: The Future is Big Graphs, Communications of ACM, September 2021.

[1]
M. Malak and R. East, *Spark GraphX in action*. Simon; Schuster, 2016.

Stream processing

Stream processing

Georgios Gousios

23 August 2021

What is streaming?

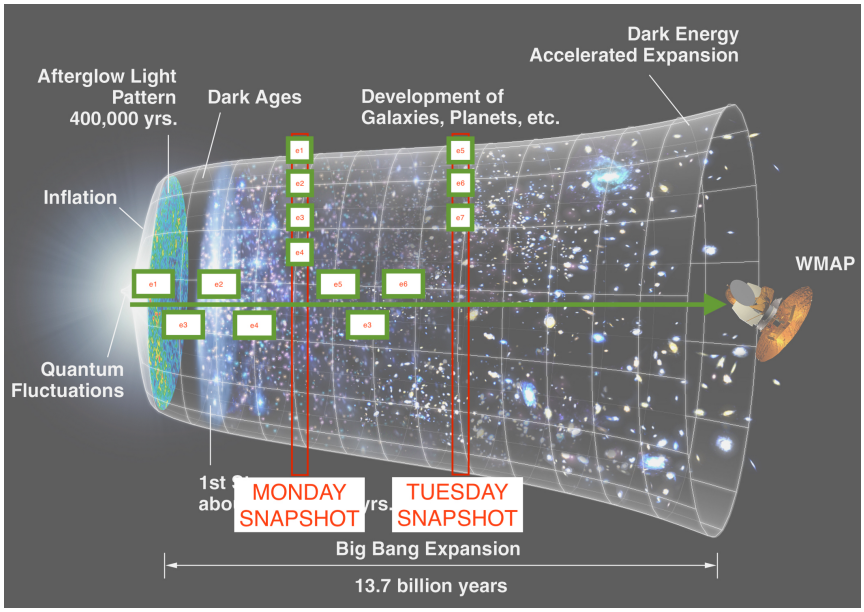
Big data is big

- Typical processing scenario for big data:
  - Aggregate data in intermediate storage
  - Run batch job overnight, store results in permanent storage
  - Use Spark for interactive exploration of recent data

Assumes that the value of the data is hidden in it ("needle in haystack")

Data is NOT static

Running processes generate data continuously, users need to continously monitor processes. *The fact that we use mostly static data is due to legacy constraints.*



Static data vs time

Bounded and unbounded datasets

- **Bounded Data:** A dataset that can be enumerated and/or iterated upon
  - Students in CSE2520
  - Countries in the world
- **Unbounded Data:** A dataset that we can only enumerate given a snapshot. Unbounded data does not have a size property.
  - Natural numbers
  - Facebook/Twitter posts

Q: Most datasets are:

you guessed right, unbounded.

Stream and batch processing

- **Batch processing** applies an algorithm on a bounded dataset to produce a single result at the end
  - Unix, Map/Reduce and Spark are batch processing systems
- **Stream processing** applies an algorithm on continuosly updating data and continuously creates results
  - Flink and Storm are stream processing systems
  - Natural fit for unbouded datasets
  - Bounded data are usually a time-restricted view of unbounded data

Use cases for stream procesing

- Intrusion and fraud detection
- Algorithmic trading
- Process monitoring (e.g. production processes, or logs)
- Traffic monitoring
- When we can discard raw data and prefer to store aggregates

> *What changes faster over time; data or code?*
>
> *If* $\frac{\Delta d}{\Delta t} \gg \frac{\Delta c}{\Delta t}$ *, this is a streaming problem*
>
> *If* $\frac{\Delta c}{\Delta t} \gg \frac{\Delta d}{\Delta t}$ *, this is an exploration problem*

> — *Joe Hellerstein*

Data in streaming processing

Some real-world examples of such data include:

- Tweets
- Github events
- Web server logs

```
[11/Oct/2018:09:02:41 +0000] "GET /pub/developer-testing-in-IDE.pdf HTTP/1.1" 200 823280
[11/Oct/2018:09:04:36 +0000] "GET /courses/bigdata/spark.html HTTP/1.1" 200 2145339 "-"
[11/Oct/2018:09:06:20 +0000] "GET /atom.xml HTTP/1.1" 200 255069 "-" "Gwene/1.0 (The gwe
[11/Oct/2018:09:08:37 +0000] "GET /pub/eval-quality-of-open-source-software.pdf HTTP/1.1
```

Unix for stream processing

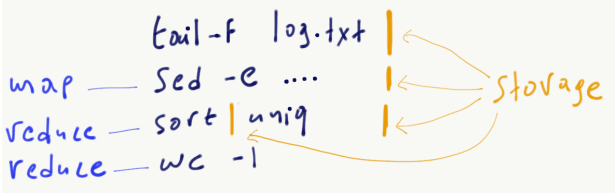
Suppose we want to calculate the total number of users on our system per hour every hour.

```
tail -f log.txt |               # Monitor interactions
sed -e …        |               # Extract user from logline
sort | uniq     |               # Unique users
wc -l           |               # Get user count
xargs -I {} echo -n `date` {}   # Print with timestamp
```

Q When will the above command finish?

Never! There is no way to tell `tail` that we want it to aggregate data *per hour* and make the pipeline recompute when `tail` emits.

What can we learn from Unix?



Unix streaming

Unix has many components required for stream processing:

- Streaming data acquisition: `tail` or `pipe`
- Intermediate storage: pipes
- Ways of applying functions on streaming data

It is missing:

- Splitting streams in batches (windowing)
- Recomputing when new batches arrive (triggers)

Stream processing in a nutshell

Stream processing is a set of techniques and corresponding systems that process **timestamped events**. For a stream processing *system* to work, we need two major components:

- A component that acquires events from producers and forwards it to consumers
- A component that processes events

To make stream processing viable in the real world, both components must be scalable, distributable and fault-tolerant.

Messaging systems

The fundamental entity that stream processing deals with is an *event*. Events are produced by continuous processes and in order to be processed they must be consumed.

Messaging systems solve the problem of connecting *producers* to *consumers*.

Unix again

```
tail -f log.txt | wc -l
```

`tail` is the producer and `wc` is the consumer. The messaging system is the *pipe*. A pipe has the following functionality

- Reads data from the producer and buffers it
- Blocks the producer when the buffer is full
- Notifies the consumer that data is available

Pipes implement the *publish / subscribe* model for 1 producer to 1 consumer.

Publish / Subscribe

Publish/subscribe systems connect multiple producers to multiple consumers.

- *Direct messaging systems* use simple network communication (usually UDP) to broadcast messages to multiple consumers. They are fast, but require the producers/consumers to deal with data loss. Example: ZeroMQ
- *Message brokers* or *queues* are centralized systems that sit between producers and consumers and deal with the complexities of reliable message delivery.

Broker-based messaging

The producers send messages in any of the following modes:

- *Fire and forget*. The broker acks the message immediately
- *Transaction-based*: The broker writes the message to permanent storage prior to ack'ing it.

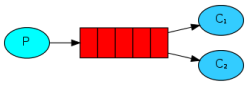
The broker:

- Buffers the messages, spilling to disk as necessary
- Routes the messages to the appropriate queues
- Notifies consumers when messages have arrived

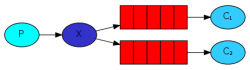
The consumers:

- Subscribe to a queue that contains the desired messages
- Ack the message receipt (or successful processing)

Messaging patterns

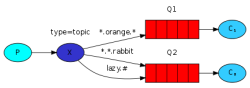


Competing workers: Multiple consumers read from a single queue, competing for incoming messages



Fan out pattern

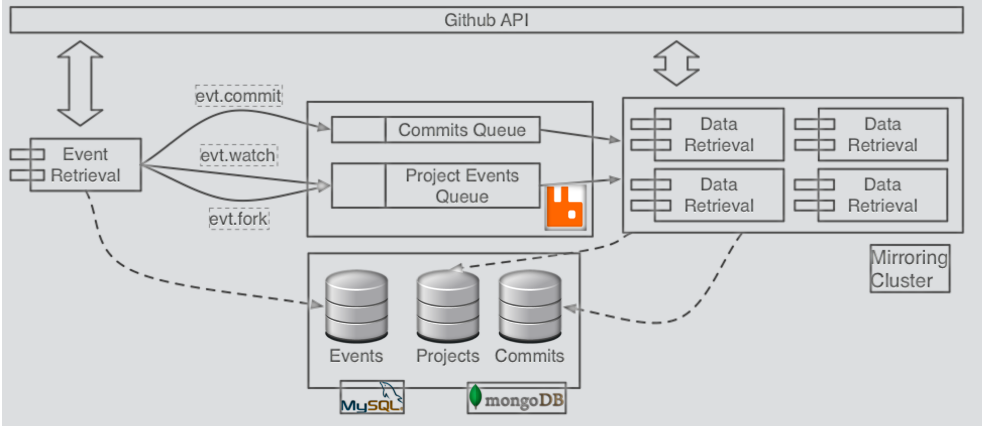
Fan out: Each consumer has a queue of its own. Incoming messages are replicated on all queues



Topics pattern

Message routing: The producer assigns keys to msg metadata. The consumer creates topic queues by specifying the keys it is interested to receive messages for.

Broker-based example: GHTorrent



GHTorrent architecture

GHTorrent uses topic queues to decouple the following the GitHub event stream from the retrieval of items linked from events. Events are written to the RabbitMQ broker with a routing key according to their event type; a configurable number of data retrieval processes subscribes to those queues.

Drawbacks of broker-based messaging

Broker-based messaging is widely used and well understood. It has however one drawback: *after a message is received, it dissapears*. This leads to lost opportunities:

- We cannot reprocess messages (e.g. when a new application version is installed)
- We cannot prove that a message was delivered

Q: How can we solve those problems?

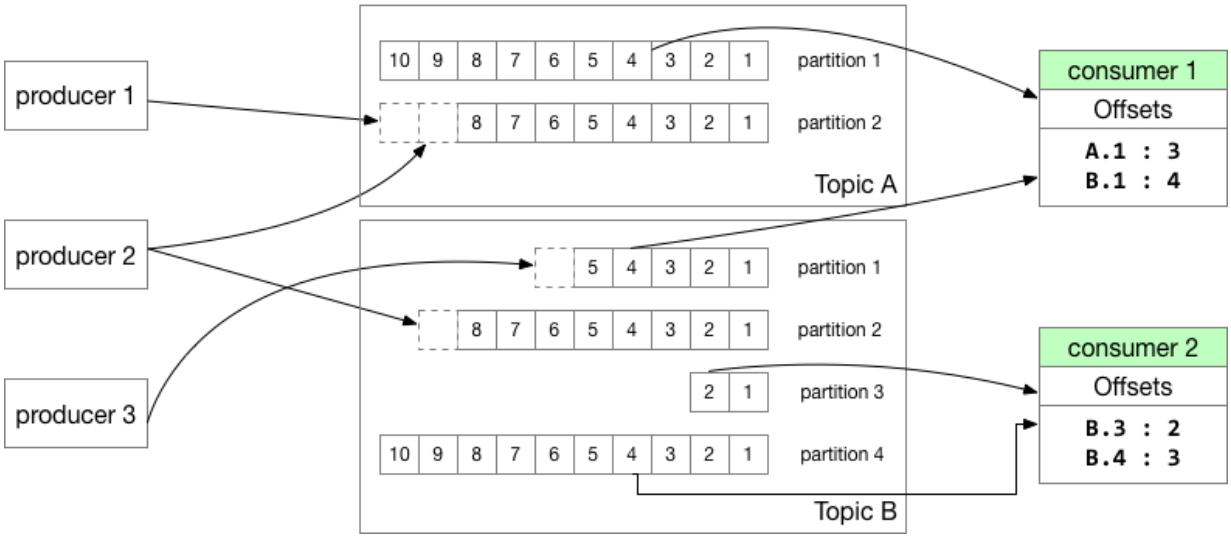
A: Instead of just forwarding messages, we can *store and forward* them.

Log-based messaging systems

A *log* is an append-only data structure stored on disk. We can exploit logs to implement messaging systems:

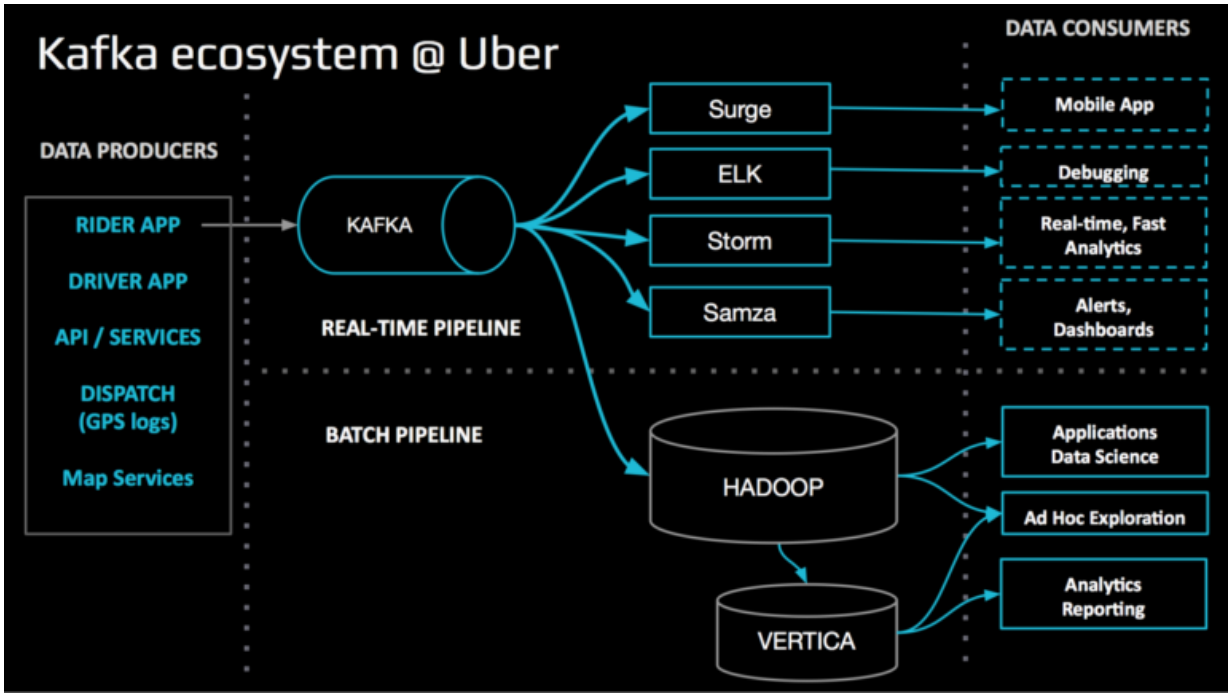
- Producers *append* messages to the log,
- All consumers connect to the log and *pull* messages from it. A new client starts processing from the *beginning* of the log.
- To maximize performance, the *broker* partitions and distributes the log to a cluster of machines.
- The *broker* keeps track of the current message offset for each consumer per partition

Log-based messaging overview



A generic partitioned log system

Kafka



Kafka use at Uber

Kafka is the most well known log server. It is being used both to aggregate and store raw events and as an intermediary between systems.

Programming models for stream processing

What programming models for streams enable processing of events to derive (some form of) state.

- Event sourcing / Command Query Segregation (CQS)
- Reactive programming
- The DataFlow model

Read also this comprehensive blog post by Martin Kleppmann.

Event sourcing and CQS

> *Capture all changes to an application state as a sequence of events.*

Instead of mutating the application state (e.g. in a database), we store the event that causes the mutation in an immutable log. The application state is generated by processing the events. This enables us to:

- Use specialized systems for scaling writes (e.g. Kafka) and reads (e.g. Redis), while the application remains stateless.
- Provide separate, continuously updated views of the application state (e.g. per user, per location etc)
- Regenerate the application state at any point in time by reprocessing events

Reactive programming

Reactive programming is a declarative programming paradigm concerned with **data streams** and the **propagation of change** (Wikipedia).

Reactive APIs model event sources as *infinite collections* on which *observers subscribe* to receive events.

```
Observable.from(TwitterSource).      // List of tweets
  filter{_.location == 'NL'}.        // Do some filtering
  flatMap{t => GeolocateService(t)}. // Precise geolocation
  groupBy{loc => loc.city}.          // Group results per city
  flatMap{grp => grp.map(v => (grp.key, v))}.
  subscribe(println)
```

Example is in the Reactive Extensions (Rx) formulation. .NET Rx and Java 9 (Flow), include facilities for reactive programming.

The Dataflow model

The Dataflow model was introduced by Akidau et al. [1] as a generic implementation of the MillWheel system [2]. Flink was heavily inspired by it.

The DataFlow model attempts to explain stream processing in four dimensions:

- What: results are being computed
- Where: in *event time* they are being computed
- When: in *processing time* they are materialized
- How: earlier results relate to later refinements

Time in stream processing

In streaming systems, we have two notions of time:

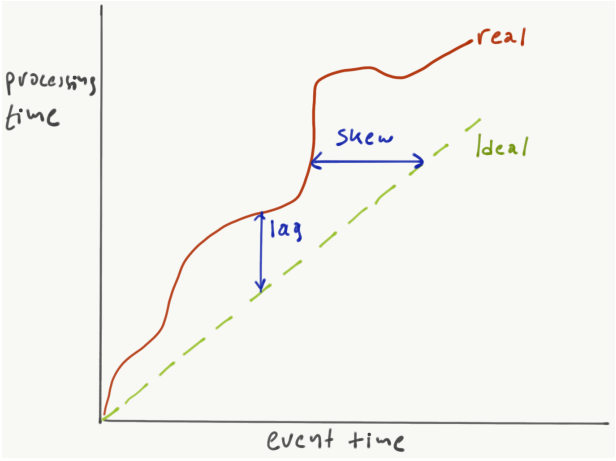
- **Processing time**: the time at which events are observed in the system
- **Event time**: the time at which events occurred

D: Describe a scenario where those are different.

Applications that calculate streaming aggregates (e.g. avg rainfall per country per hour) don't care much about the event order.

Applications with precise timing requirements (e.g. bank transactions, fraud detection) care about *event* time. Events may however enter the system delayed or out of order.

Event Time skew



Event time skew

If processing (wall-clock) time is t

- **Skew** is calculated $t - s$, where s is the timestamp of the latest event processed
- **Lag** is calculated as $t - s$, where s is the actual timestamp of an event

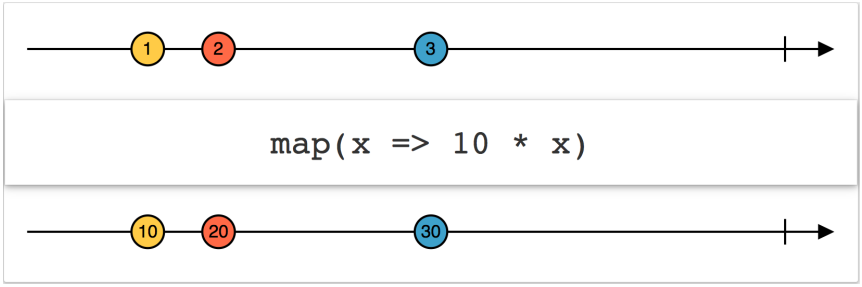
What : operations on streams

- **Element-wise** ops apply a function to each individual message. This is the equivalent of `map` or `flatMap` in batch systems.
- **Aggregations** group multiple events together and apply a reduction (e.g. `fold` or `max`) on them.

Element-wise operations: `map`

$Stream[A].map(x : A \rightarrow B) : Stream[B]$

Convert types of stream elements



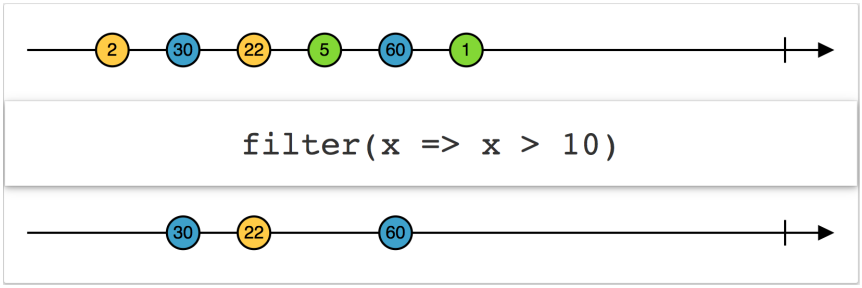
Map

```
// Rx
Observable.from(List(1,2,3)).map(x => 10 * x)

// Flink
env.fromCollection(List(1,2,3)).map(x => 10 * x)
```

Element-wise operations: `filter`

$Stream[A].filter(x : A \rightarrow Boolean) : Stream[A]$

Only keep events that satisfy the predicate.



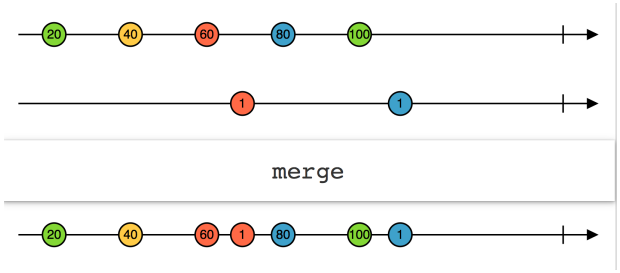
Filtering

```
// Rx
Observable.from(List(2,30,22,5,60,1)).map(x => x > 10)

// Flink
env.fromCollection(List(2,30,22,5,60,1)).map(x => x > 10)
```

Element-wise operations: `merge`

$$Stream[A].merge(b : Stream[B >: A]) : Stream[B]$$

Emit a stream that combines all events from both streams.



Merging streams

```
// Rx
val a = Observable.from(List(20,40,60,80,100))
val b = Observable.from(List(1,1))
a.merge(b)

// Flink
val a = env.fromCollection(List(20,40,60,80,100))
val b = env.fromCollection((List(1,1))
a.union(b)
```

Element-wise operations: `flatMap`

$$flatMap(f : A \rightarrow Stream[B]) : Stream[B]$$

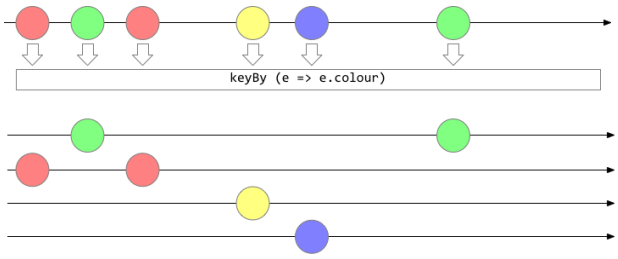
Apply `f` on all elements of `Stream[A]` and flatten any nested results to a new stream.

```
// Rx
Observable.from(List('foo', 'bar')).
           flatMap(x => Observable.from(x.toCharArray))

// Flink
env.fromCollection(List('foo', 'bar'))
    flatMap(x => x.toCharArray)
```

Element-wise operations: `keyBy`

$$Stream[A].keyBy(f : A \rightarrow K) : Stream[(K, Stream[K])]$$

Partition a stream using a discriminator function and produce (a stream of) streams that emit the partitioned data.



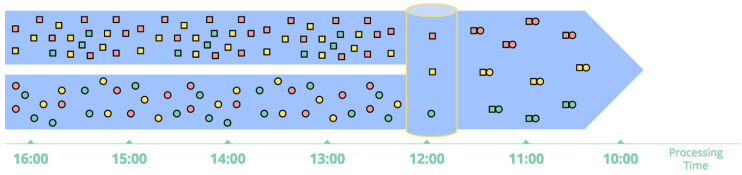
Stream keyBy

`keyBy` (or `groupBy` in Rx) creates partitioned streams that can be processed in parallel. Moreover, keys are required for various operations combining data.

Element-wise operations: `join`

```
Stream[A].join(b: Stream[B],
               kl: A => K,
               kr: B => K,
               rs: (A,B) => R): Stream[R]
```

Join streams A and B. Key selector functions `kl` and `kr` extract keys of type K, on which the join operation is performed. On each joined pair, the result selector function `rs` is applied to derive the result type.



Stream join

Stream joining example

Find commits that caused exceptions and notify the authors.

```
case class StackEntry(file: String, line: Int)
case class Exception(exception: String, entries: Seq[StackEntry])
case class DiffLine(file: String, line: Int, content: ...)
case class Commit(author: String, diff: Seq[DiffLine])

val logs : Stream[(Exception, StackEntry)] = env.socketTextStream(host, port).
  flatMap(e => for(s <- e.entries) yield (e, s))
val diffs : Stream[(Commit, Diff)] = env.GitRepoSource(...).
  flatMap(c => for(d <- c.diff) yield(c, d))

logs.join(diffs).
     where(stackEntry => stackEntry.file).
     equalTo(diff => diff._2._file).
     apply((log, diff) => (diff._1.author, diff._1.commit.sha, log.exception)).
     map(e => sendEmail(...))
```

Q: How can a stream processor execute this?

A: Presumably, stack traces is a faster stream than commits; joining will require all keys to be kept in memory (per processing node). Theoretically, this requires infinite memory.

Aggregations / Reductions

```
Stream[A].aggregate(f: AggregationFunction[A, T, B]): Stream[B]

trait AggregationFunction[IN, ACC, OUT] {
  def createAccumulator(): ACC
  def add(value: IN, acc: ACC): ACC // Type conversion IN -> ACC
  def getResult(acc: ACC): OUT      // Type conversion ACC -> OUT
  def merge(a: ACC, b: ACC): ACC
}
```

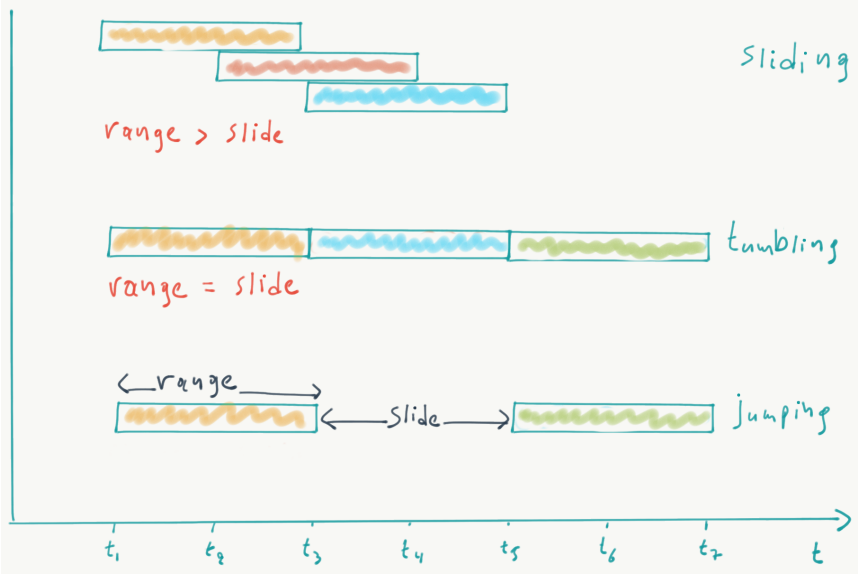
Aggregations group multiple events together and apply a reduction on them.

Q: How can we aggregate events, when our event stream is infinite?

Hint: remember the Unix example from before.

A: We can create event groups by time (e.g. *every 2 minutes*) or by count (e.g., *every 100 events*).

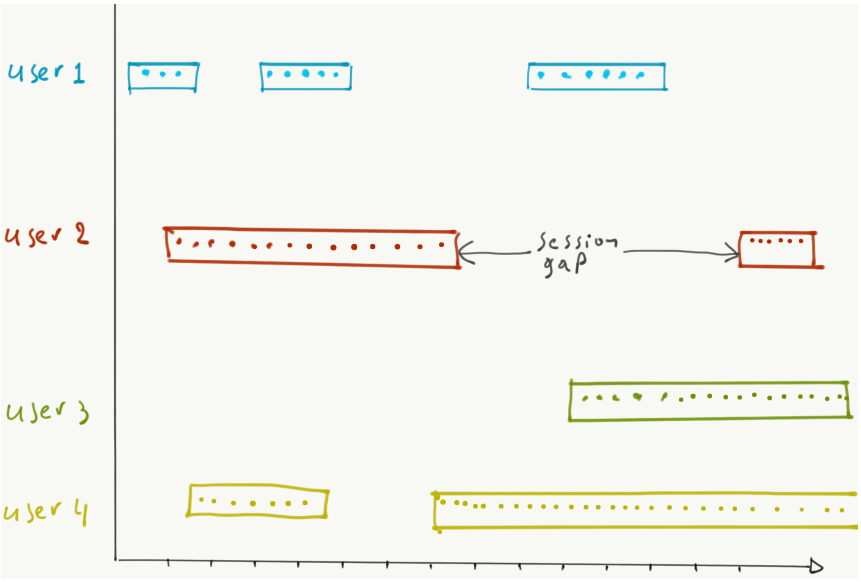
Where: Streaming Windows



Windowing in streaming systems

Windows are static size (e.g., 1000 events) or time-length (e.g., 10 secs) "batches" of data.

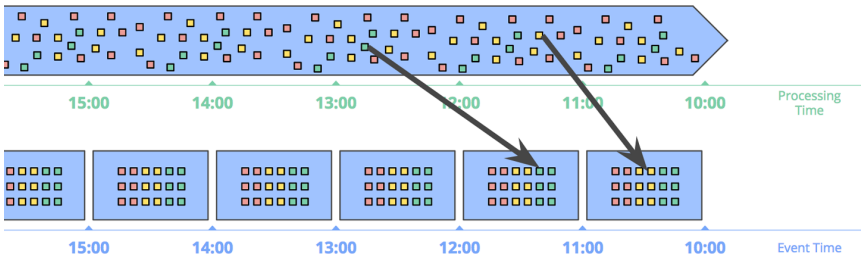
Session windows



Session windows

Session windows are dynamically sized windows that aggregate batches of (typically) user activity. Windows end after *session gap* time.

Example: Aggregating via windows



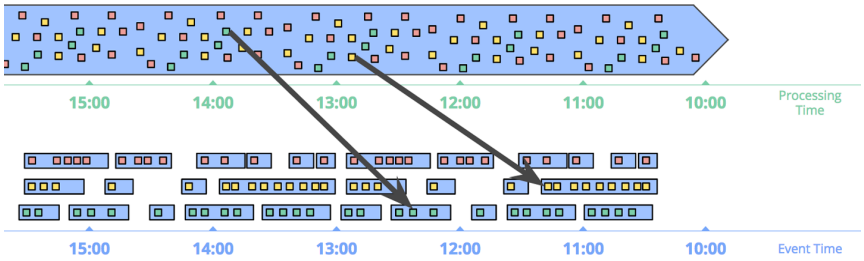
Out of order events

```
// Count number of tweets per user per minute
tweets.map(t => (t.user_id, 1))
      .keyBy(x => x._1)
      .timeWindow(Time.minutes(1))
      .reduce((a,b) => a._2 + b._2)
```

Every minute, this will produce a list of pairs like so:

```
(323, 1)
(44332, 4)
(212, 32)
...
```

Example: Using session windows



Session Windows

```
// Number of clicks per user session
case class Click(id: Integer, link: String, ...)
clickStream.map(c => (c.id, 1))
           .keyBy(x => x._1)
           .window(EventTimeSessionWindows.withGap(Time.minutes(10)))
           .sum(1)
```

When a session terminates, we get results like:

```
(323, 1)
(44332, 4)
(212, 32)
...
```

Windows: Things to remember

There are 2 things to remember when using event-time windows.

- **Buffering:** Aggregation functions are applied when the window finishes (see When). This means that in-flight events need to be buffered in RAM and spilled to disk.

- **Completeness:** Given that events may arrive out of order, how can we know that a window is ready to be materialized and what do we do with out of order events?

When: Window Triggers

A trigger defines when in *processing time* are the results of a window materialized / processed. Two types of triggers can be defined:

- **Per-record triggers** fire after x records in a window have been encountered.
- **Aligned delay triggers** fire after a specified amount of time has passed across all active windows (aka micro-batching).
- **Unaligned delay triggers** fire after a specified amount of time has passed after the first event in a single window.

Click on the links to watch Akidau's excellent visualizations

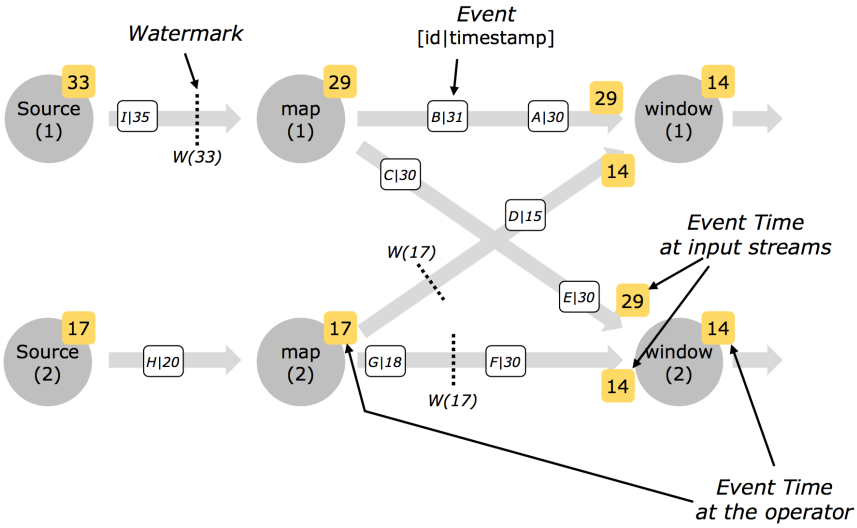
Watermarks

Event-time processors need to determine when *event time* has progressed enough so that they can trigger windows. When reprocessing events from storage, a system might process weeks of event-time data in seconds; relying on processing time to trigger windows is not enough.

Watermarks flow as part of the data stream and carry a timestamp. They are a declaration that by a point in the stream, all events carrying a timestamp up to the watermark timestamp should have arrived.

Watermarks allow late messages to be processed up to a specified amount of (event-time) delay (*allowed lateness*).

Watermarks in parallel streams



Watermarks

As the watermarks flow through the streaming program, they advance the event time at the operators where they arrive. Whenever an operator advances its event time, it generates a new watermark downstream for its successor operators.

How: Window Refinements

In certain complex cases, a combination of triggers and watermarks flowing may cause a window to be materialized multiple times. In such cases, we can *discard*, *accumulate* or *accumulate and retract* the window results.

For further reading, consult the Beam documentation.

Stream processing systems

Messaging systems are purposed to move data from produces to consumers, in a scalable and fault-tolerant way.

However, they do not process the moved data; this is the job of dedicated, stream processing systems.

The stream as a database

There is nothing that fundamentally disallows event streams from acting as a database.

- Events can be filtered and transformed
- Event streams can be joined with other event streams
- Event streams can be aggregated (given time constraints)
- Event streams can be replicated on other hosts for scaling and fault tolerance

The main difference between streams and databases is that databases contain *state*, whereas streams contain *state modifications*. Therefore, databases can be updated, while streams can be appended.

Approaches to processing streams

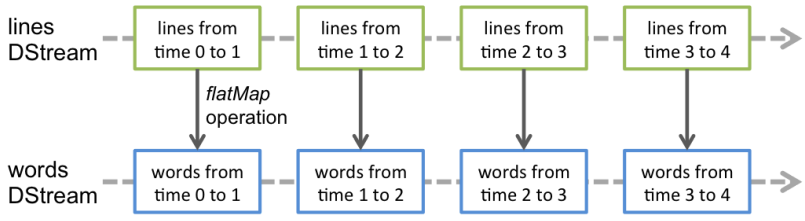
- **Micro-batching**: Aggregate data in batches of configurable (processing-time) duration
- **Event-based streaming:** Process events one by one

Event-time systems can emulate micro-batching by setting an *aligned delay trigger* to keyed windows.

Apache Spark Streaming

Spark Streaming is an example of a *micro-batching* architecture. Spark Streaming breaks the input data into batches (of `x` seconds processing time length) and schedules those in the cluster using the exact same mechanisms for fault tolerance as normal RDDs.

```
ssc.socketTextStream("localhost", 9999).
  flatMap(_.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
```



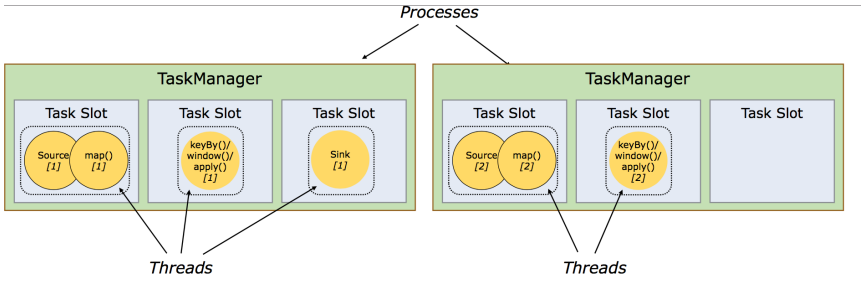
Micro-batch

Issues with micro-batching

- *Latency*
  - The micro-batch computation is triggered after the batch times out
  - Each batch needs to be scheduled, libraries need to be loaded, connections need to be open etc

- *Programming model*
  - No clean separation of mechanism from business logic
  - Changing the micro-batch size leads to different results

Real streaming: Flink

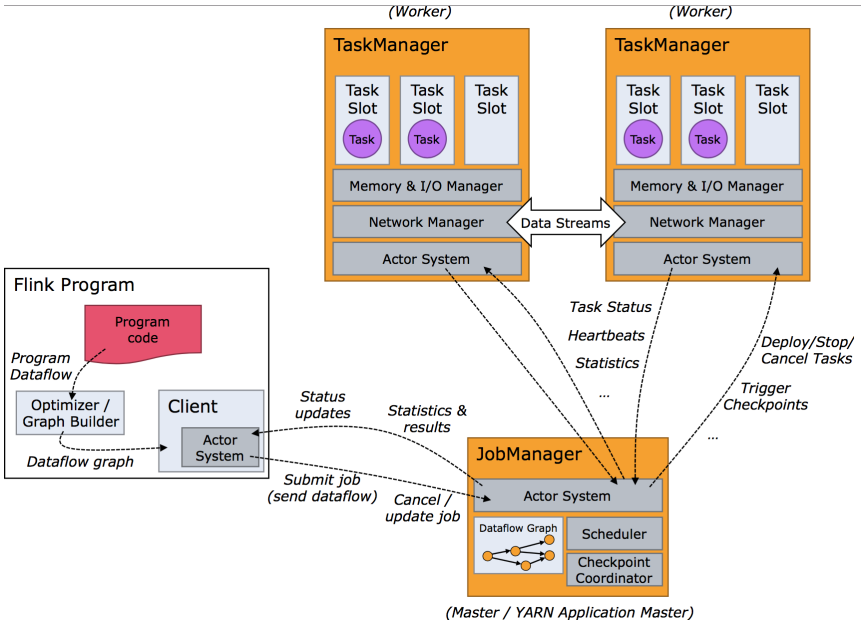


Flink job splitting

In Flink, a program is first compiled to a *data-flow graph*. Each node in the DFG represents a *task*, and can be scheduled within a *task manager* (essentially, a JVM instance).

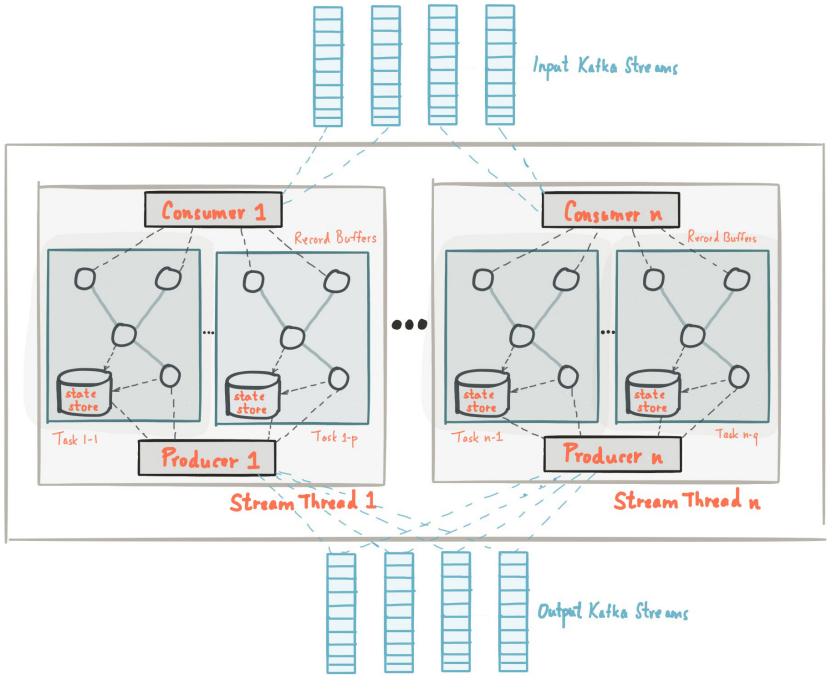
A DFG consists of a Source, a Sink and intermediate computations. A sink cannot be a source: this means that 2 Flink computations cannot exchange data directly.

Anatomy of a Flink cluster



Flink cluster architecture

More streaming: Kafka Streams cluster



Kafka Stream architecture

Stateful streaming

Imagine that for each item we process, we would like to keep a counter.

```
val stream: DataStream[(String, Int)] = ...

val counts: DataStream[(String, Int)] = stream
  .keyBy(_._1)
  .mapWithState((in: (String, Int), count: Option[Int]) =>
    count match {
      case Some(c) => ( (in._1, c), Some(c + in._2) )
      case None => ( (in._1, 0), Some(in._2) )
    })
```

The `mapWithState` operator takes and returns an optional state, which the stream processor must maintain.

Implicit states

In addition, many operators, for example windowing and aggregation ones, are inherently stateful.

```
env.addSource(...)
.map(bytes => Event.parse(bytes) )                          stateless
.keyBy("producer")
.mapWithState { (event: Event, state: Option[Int]) => {     stateful
   // rules
}
.filter(alert => alert.msg.contains("CRITICAL"))            stateless
.keyBy("msg")
.timeWindow(Time.seconds(10))                               stateful
.sum("count")
```

Stateful vs Stateless processing

Q: As the processing graph is distributed, we need a consistent, fault-tolerant global view of the counter. How can we implement this?

A: An idea would be to pause all operators, start a 2-phase commit process and restart the processing when all nodes are committed.

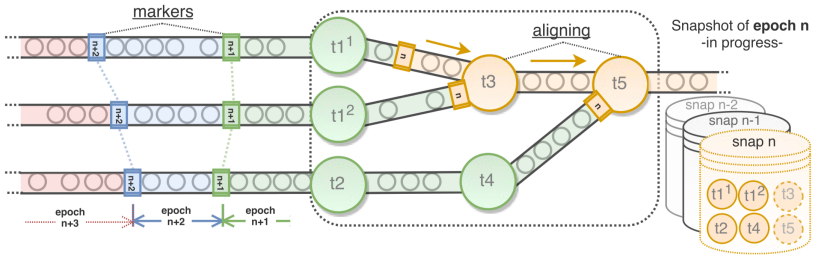
But we can do better than that!

The Chandy-Lamport Algorithm

The Chandy-Lamport algorithm [3] can be used to capture consistent global snapshots. It models a distributed system as a graph of processes that have input and output channels, overseen by a snapshot initiator:

- The snapshot initiator saves its local state and sends a marker to all its output channels
- All receiving nodes: i) save their local state and the state of the channel that delivered the marker, ii) forward the marker to all outgoing channels
- When the marker reaches the initiator, the snapshot is done

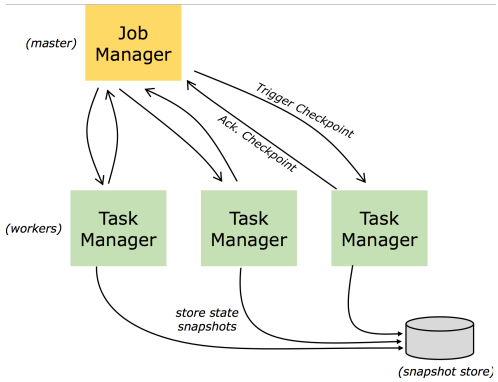
Flink snapshots



The Flink snapshot algorithm

Flink takes incremental snapshots by interleaving epoch markers with messages [4]. It assumes that input streams are durably logged and repeatable (e.g., with Kafka). Operators wait for the same epoch markers from all channels before they take a snapshot.

System view of snapshots



How Flink co-ordingates snapshots

Event processing guarantees

The following guarantees are offered by streaming systems

- **At most once** an event will be processed once (if delivered at all)
- **At least once** an event might flow through a system twice, in case of failure.
- **Exactly once** an event only flows through a set of operators once.

Flink supports *exactly once*. To do so, it requires the source to support event replay on request and the sink to be transactional. Both requirements are satisfied by Apache Kafka.

Deployment view of a simple application

Image credits

- Spacetime continuum image isfrom the NASA/WMAP science team
- Many of the stream processing figures are (c) Tyler Akidau, available here
- The message broker pattern graphics are from the RabbitMQ documentation
- Kafka @ Uber, from the Uber engineering blog
- Streaming Operator screenshots from RxMarbles
- Watermark image, by the Flink documentation
- Flink snapshoting, by Carbone et al.[4]

Bibliography

[1]
T. Akidau *et al.*, "The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[2]
T. Akidau *et al.*, "MillWheel: Fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[3]
K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[4]
P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink: Consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.

[5]
M. Kleppmann, *Designing data-intensive applications*. O'Reilly Media, Inc., 2017.

[6]
T. Akidau, S. Chernyak, and R. Lax, *Streaming systems: The what, where, when, and how of large-scale data processing*. O'Reilly, 2018.

[7]
M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 423–438.

Copyright