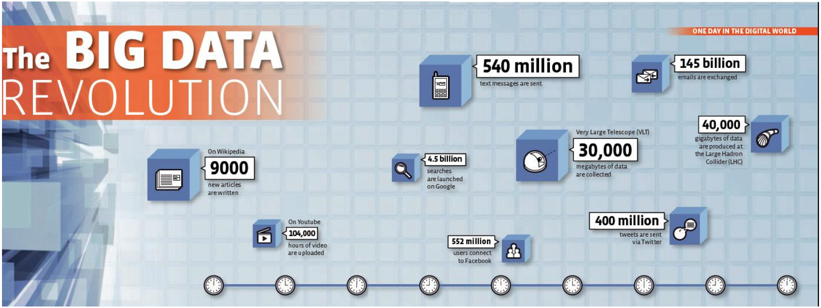
Développement logiciel pour le Cloud (TLC)

Introduction

Davide Frey



Source: CNRS magazine 2013

Big data

"Big data refers to data sets whose size is beyond the ability of typical database software tools to capture, store, manage and analyze." — *The McKinsey Global Institute, 2011.*

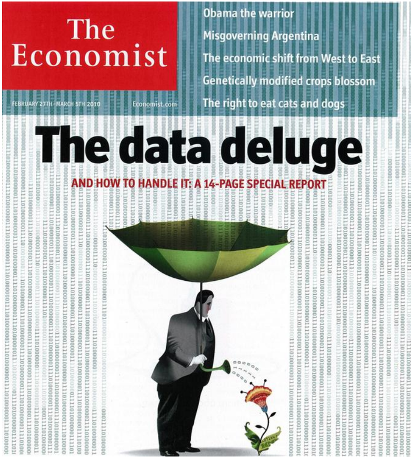
"Big data is the term for a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications." — *Wikipedia.*

How big is big data?

Earlier Berkeley studies estimated that by the end of 1999, the sum of human-produced information (including all audio, video recordings and text/books) was about 12 Exabytes of data *(1 exabyte = 1 million TB)*.

Eric Schmidt: Every 2 Days We Create As Much Information As We Did Up To 2003.

http://techcrunch.com/2010/08/04/schmidt-data/

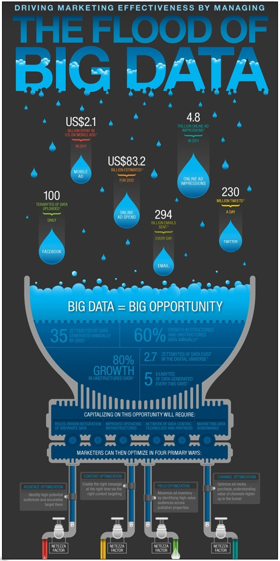


In 2010 the Digital Universe contained 1.2 zettabytes (1 zettabyte = 1 *billion* TB)

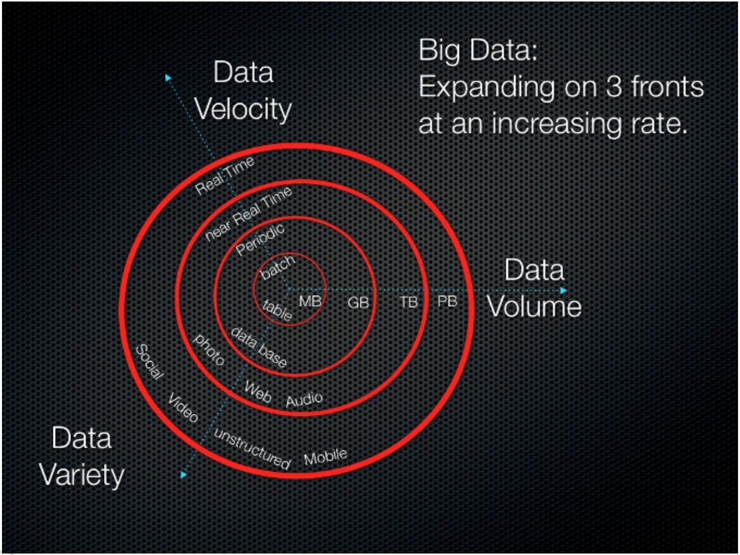
In 2020 the Digital Universe will contain 35 zettabytes.

1 NEW DEFINITION IS ADDED ON urban
1,600+ READS ON Scribd
13,000+ HOURS MUSIC STREAMING ON PANDORA
12,000+ NEW ADS POSTED ON craigslist
370,000+ MINUTES VOICE CALLS ON skype
98,000+ TWEETS
20,000+ NEW POSTS ON tumblr.
THE LARGEST SOCIAL READING PUBLISHING COMPANY!!
320+ NEW twitter ACCOUNTS
13,000+ iPhone APPLICATIONS DOWNLOADED
100+ NEW LinkedIn ACCOUNTS
1 NEW ARTICLE IS PUBLISHED
associatedcontent
THE WORLD'S LARGEST COMMUNITY CREATED CONTENT!!
QUESTIONS ASKED ON THE INTERNET...
100+ Answers.com
40+ YAHOO! ANSWERS
IN 60 SECONDS
6,600+ NEW PICTURES ARE UPLOADED ON flickr
600+ NEW VIDEOS
YouTube
25+ HOURS TOTAL DURATION
70+ DOMAINS REGISTERED
50+ WORDPRESS DOWNLOADS
60+ NEW BLOGS
168 MILLION EMAILS ARE SENT
694,445 SEARCH QUERIES
1,700+ Firefox DOWNLOADS
695,000+ facebook STATUS UPDATES
125+ PLUGIN DOWNLOADS
1,500+ BLOG POSTS
79,364 WALL POSTS
510,040 COMMENTS
Google
Google Search
GO-Globe.com

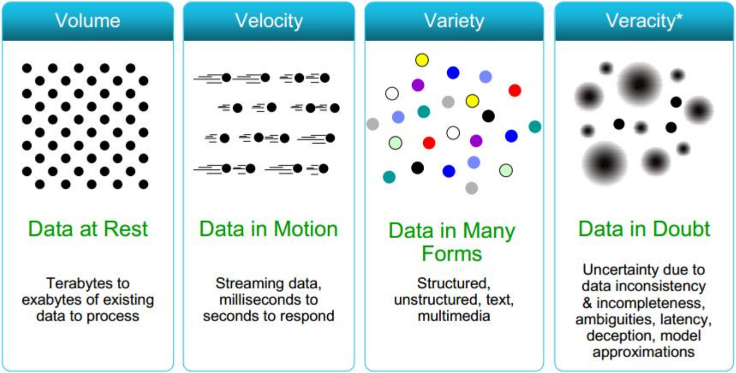
Why do we want to analyze this data?



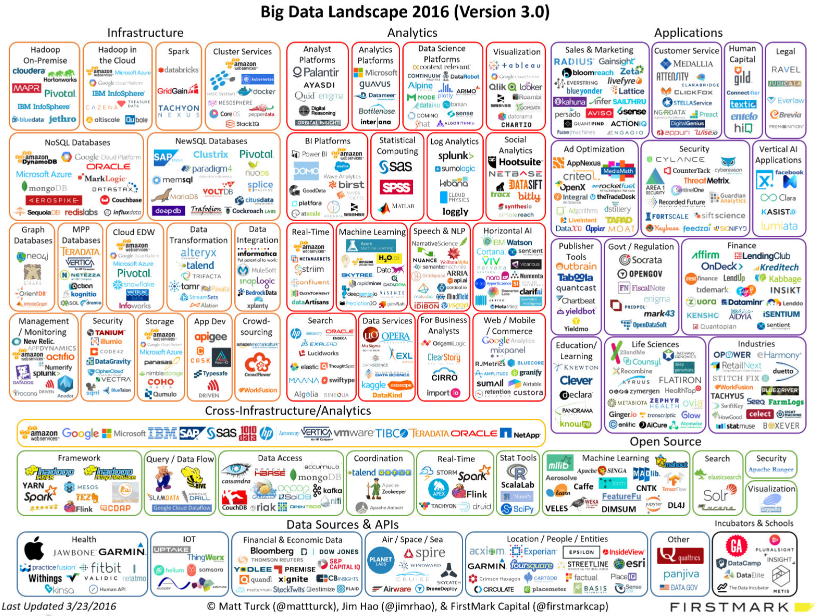
Big data challenges: the "three V's"



The “three V’s” are becoming the “four V’s”



Big data == big privacy concerns. . .


Big Data Landscape 2016 (Version 3.0)
Infrastructure
Analytics
Applications
Hadoop On-Premise
Hadoop in the Cloud
Spark
Cluster Services
Analyst Platforms
Analytics Platforms
Data Science Platforms
Visualization
Sales & Marketing
Customer Service
Human Capital
Legal
NoSQL Databases
NewSQL Databases
BI Platforms
Statistical Computing
Log Analytics
Social Analytics
Ad Optimization
Security
Vertical AI Applications
Graph Databases
MPP Databases
Cloud EDW
Data Transformation
Data Integration
Real-Time
Machine Learning
Speech & NLP
Horizontal AI
Publisher Tools
Govt / Regulation
Finance
Management / Monitoring
Security
Storage
App Dev
Crowd-sourcing
Search
Data Services
For Business Analysts
Web / Mobile / Commerce
Education/ Learning
Life Sciences
Industries
Cross-Infrastructure/Analytics
Open Source
Framework
Query / Data Flow
Data Access
Coordination
Real-Time
Stat Tools
Machine Learning
Search
Security
Visualization
Data Sources & APIs
Incubators & Schools
Health
IOT
Financial & Economic Data
Air / Space / Sea
Location / People / Entities
Other
Last Updated 3/23/2016
© Matt Turck (@mattturck), Jim Hao (@jimrhao), & FirstMark Capital (@firstmarkcap)
FIRSTMARK

MapReduce

MapReduce was introduced by Google in 2004:

- Big data at that time: 20+ billion web pages x 20 kB = 400+ TB
- One computer can read 30-35 MB/sec from disk
  - $\Rightarrow$ 4 months to read the Web
  - $\Rightarrow$ $\sim$1000 hard drives just to store the web

MapReduce

MapReduce was introduced by Google in 2004:

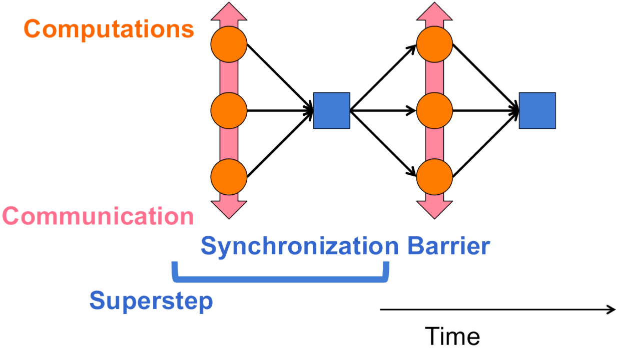
- Big data at that time: 20+ billion web pages x 20 kB = 400+ TB
- One computer can read 30-35 MB/sec from disk
  - $\Rightarrow$ 4 months to read the Web
  - $\Rightarrow$ ~1000 hard drives just to store the web
- But they wanted to process the data! This requires much more computation, data, etc.

"Google Infrastructure for Massive Parallel Processing",

Walfredo Cirne, Presentation in the industrial track in CCGrid'2007.

The Bulk Synchronous Parallel model

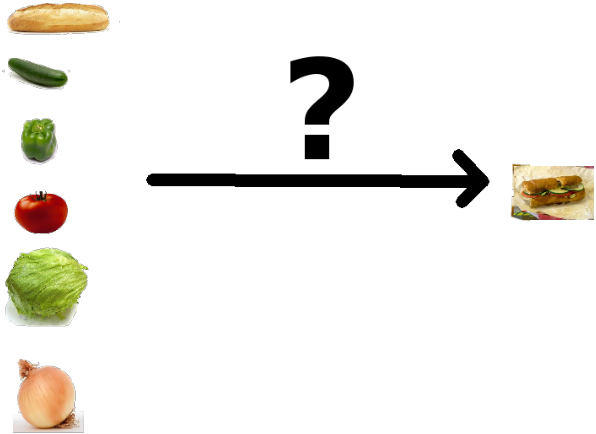
- Maximize I/O
- Minimize coordination



Parallelization is not so easy

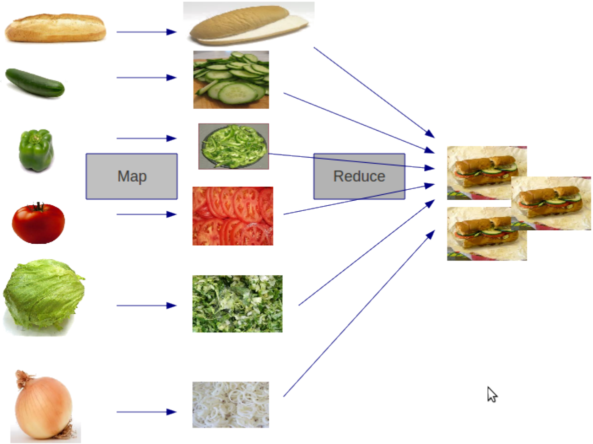
- ☺ "Easy" parallelization
  - Reading the Web on 1000 machines $\Rightarrow$ less than 3 hours
- ☹ This requires lots of programming work
  - Communication & coordination
  - Debugging
  - Fault-tolerance
  - Management and monitoring
  - Optimization
- ☹ ☹ Repeat the same painful process for every problem you want to solve

Let's make sandwiches



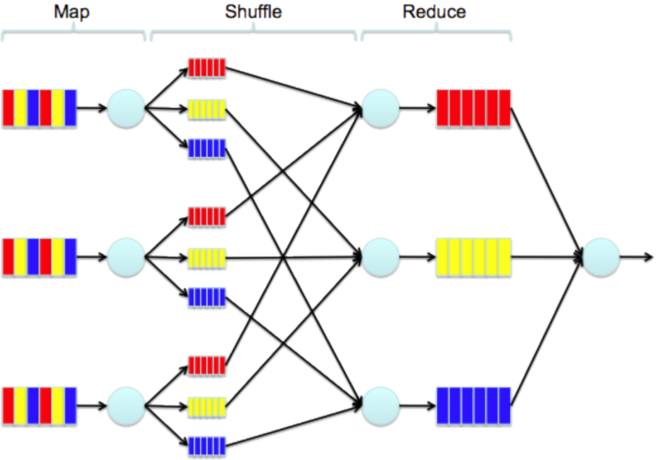
https://twitter.com/tgrall

Let's make sandwiches



https://twitter.com/tgrall

Back to MapReduce



http://www.slideshare.net/lynnlangit/hadoop-mapreduce-fundamentals-21427224/

Programmer must write two simple functions

- map(key,value) $\rightarrow$ ¡key',value'¿*
  The map function reads input data and produces intermediate tuples which are ready for the second phase

Programmer must write two simple functions

- map(key,value) $\rightarrow$ ¡key',value'¿*
  The map function reads input data and produces intermediate tuples which are ready for the second phase

- reduce(key',¡value'¿*) $\rightarrow$ ¡key',value''¿*
  The reduce function takes all intermediate tuples with the same key, and produces output tuples

Example: word count

Let's take a (long) piece of text. Can we compute the number of occurrences of each word?

- Map function: take a subset of the input, generate one intermediate tuple for every word in the text

```
def map(String input_key, String doc):
for each word w in doc:
    EmitIntermediate(w, 1)
```

- Shuffle operation: all tuples with the same key are automatically sent to the same reducer
- Reduce function: count the occurences we received for each word

```
def reduce(String output_key, Iterator output_vals):
  int res = 0
  for each v in output_vals:
      res = res + v
  Emit(res)
```

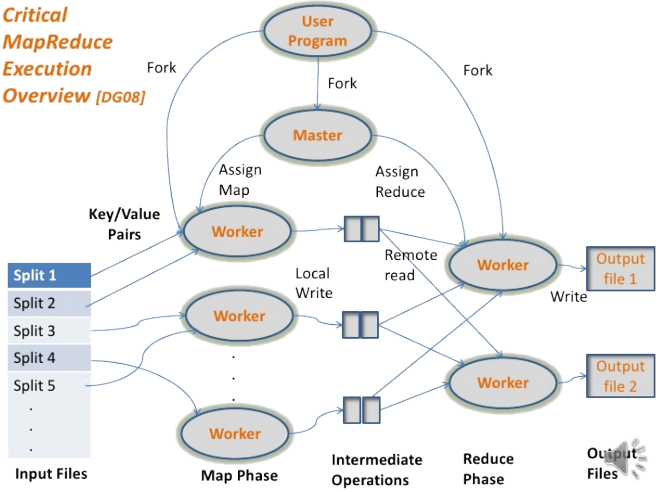
What makes MapReduce so great

- ☺ map() functions run in parallel, creating different intermediate values from different input data sets
- ☺ reduce() functions also run in parallel, each working on a different output key
- ☺ All values are processed independently

What makes MapReduce so great

- ☺ map() functions run in parallel, creating different intermediate values from different input data sets
- ☺ reduce() functions also run in parallel, each working on a different output key
- ☺ All values are processed independently
- ☹ Limitation: the reduce phase cannot start until the map phase is totally finished

MapReduce architecture



http://www.slideshare.net/diliprk/mapreduce-paradigm

MapReduce architecture

- One master server, many worker servers
  - Input data is split in chunks (∼64 MB)
  - Tasks are assigned to workers dynamically
- The master assigns each map task to a free worker
  - Considers locality of data to worker when assigning task
  - Worker reads task input (often from local disk!)
  - Worker produces R local files containing intermediate key/value pairs
- The master assigns each reduce task to a free worker
  - Worker reads intermediate key/value pairs from map workers
  - Worker sorts & applies the userâĂŹs Reduce function to produce the output

Fault tolerance

- If a worker fails:
  - The master will detect failure thanks to periodic heartbeats
  - The master re-executes the completed and in-progress map() tasks
  - The re-executes the in-progress reduce() tasks
- If the same input *always* makes the map() function crash:
  - The master will detect it and skip these values on re-execution

Fault tolerance

- If a worker fails:
  - The master will detect failure thanks to periodic heartbeats
  - The master re-executes the completed and in-progress map() tasks
  - The re-executes the in-progress reduce() tasks
- If the same input *always* makes the map() function crash:
  - The master will detect it and skip these values on re-execution
- If the master fails:

Fault tolerance

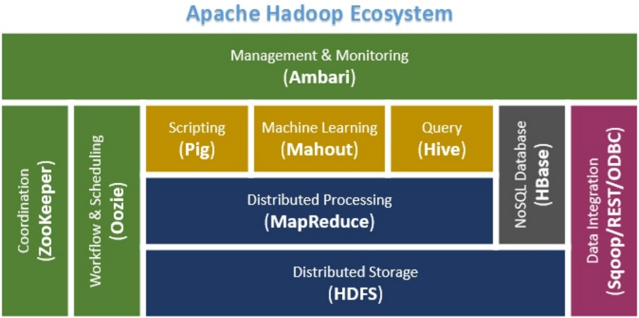
- If a worker fails:
  - The master will detect failure thanks to periodic heartbeats
  - The master re-executes the completed and in-progress map() tasks
  - The re-executes the in-progress reduce() tasks
- If the same input *always* makes the map() function crash:
  - The master will detect it and skip these values on re-execution
- If the master fails:
  ☹

MapReduce in the real world

The reference open-source implementation: Apache Hadoop

All the good clouds provide Hadoop (or similar) under a PaaS model

Hadoop's stack



http://bit.do/cSi9J

Limitations of MapReduce

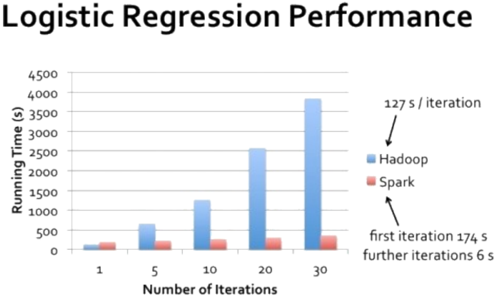
- Rigid programming model
  - Programs must be developed as a pair of map/reduce functions
  - Complex programs may be designed as a succession of iterative map/reduce steps
- By default, all intermediate result are stored in the HDFS file system
  - Replicated, fault-tolerant, etc.
  - But there are *lots* of intermediate results in a map/reduce/map/reduce/map/reduce program!

  $\Rightarrow$ Not-so-great performance

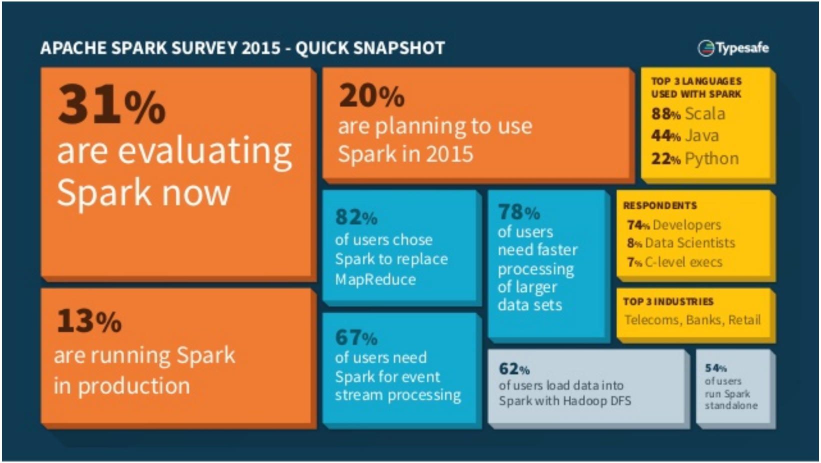
Spark

- Let's simplify application development: write "normal" code, let the system figure out how to execute it efficiently
- Let's use main memory for all intermediate data

⇒ Major performance improvements

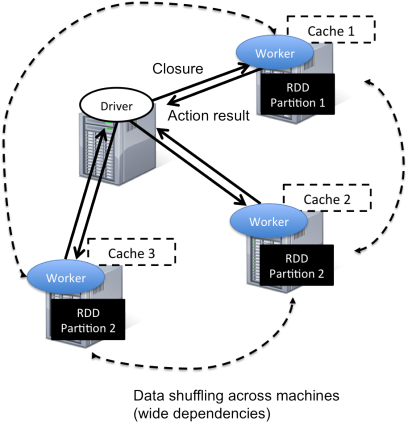


Survey within the big data developer community (2015)



Spark's architecture

- One driver node
  $\sim$ master
  - orchestrates computation, assigns work
- Many worker nodes
  - execute tasks, report to driver node



http://horicky.blogspot.fr/2013/12/spark-low-latency-massively-parallel.html

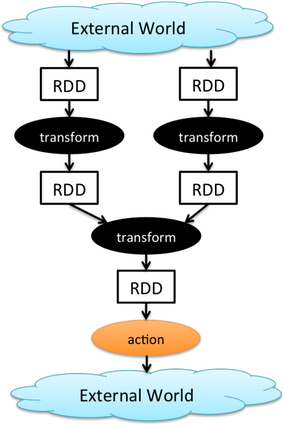
Spark RDDs

- RDD = Resilient Distributed Dataset
- Conceptually an array (or a map) of entries
  - Entries might be strings, numbers, maps, pairs, ...
  - Transparently partitioned / distributed by Spark
  - Transparently resilient (either by recomputation or storage)
- Creation:
  - Read from a local or distributed file system
  - Or produced by another Spark computation

Spark applications

A Spark application is composed of transformations and actions:

- Transformations specify how to produce an RDD from another RDD
  - But the system does not execute them immediately
- Actions trigger an actual computation
  - The system explores the graph of dependencies, and produces a directed acyclic graph of necessary transformations
  - Optimizes computations to be done
  - Distributes and organizes work



http://horicky.blogspot.fr/2013/12/spark-low-latency-massively-parallel.html

Spark transformations

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

etc...

https://spark.apache.org/docs/latest/programming-guide.html

Spark actions

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

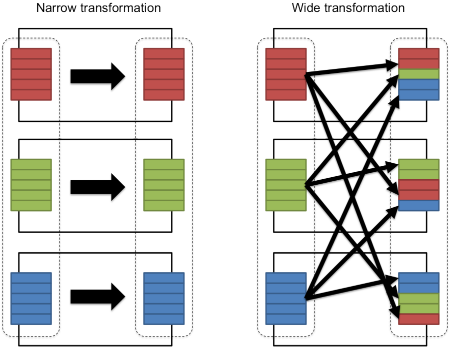
and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.

etc...

https://spark.apache.org/docs/latest/programming-guide.html

Wide and Narrow Transformations



- Wide transformations require shuffling
  - e.g., reduceByKey(. . .)
  - Network costs, higher latency

http://horicky.blogspot.fr/2013/12/spark-low-latency-massively-parallel.html

The word count example in Spark

Transformations

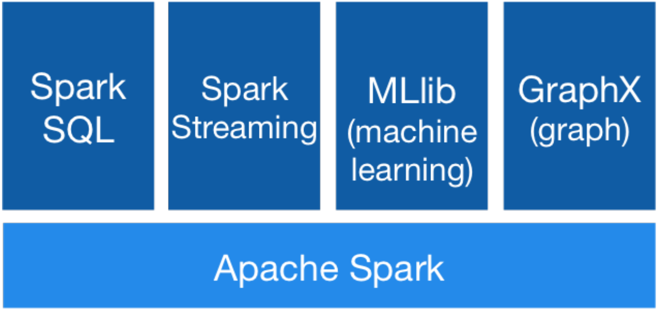
```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Action

- The developer writes simple, sequential code using the Spark transformations and actions
- Spark automatically parallelizes the code, distributes it across many nodes, and coordinates the distributed execution

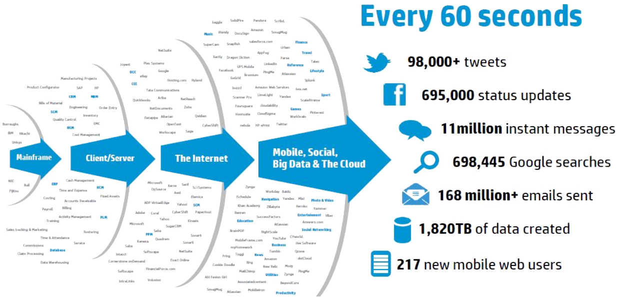
http://spark.apache.org/examples.html

Spark's stack



http://spark.apache.org/

Limitations of MapReduce/Spark



How can we keep up with the velocity of big data?

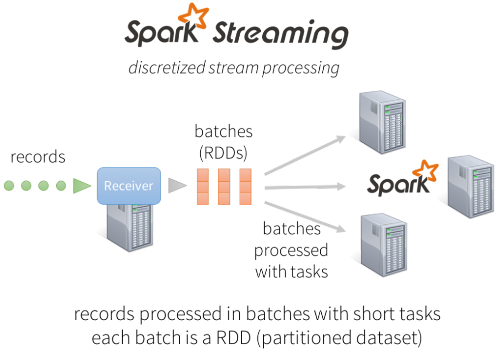
- Store incoming data (e.g., tweets)
- One in a while: process the new data, produce new results

⇒ The results are always late!

☞ We need to be able to process incoming data in real time, not as a succession of batch jobs

Spark Streaming

- Spark Streaming relies on micro-batches
  - Ingest incoming real-time data from various sources
  - Generate a new "micro-batch" at fixed time intervals (e.g., 1 second)
  - Process each micro-batch as a separate Spark job



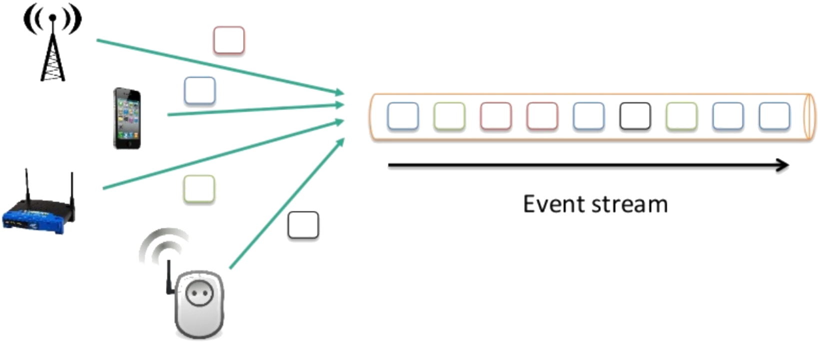
https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

Limitations of micro-batches

- Data arriving out of order is hard to handle
  - How do you detect missing data, data gaps, correct out of time order data etc?
- Batch length restricts Window-based analytics
  - Large batches $\Rightarrow$ poor responsiveness
  - Small batches $\Rightarrow$ the system is obliged to work on very small window sizes
- Code is hard to write
  - As soon as you try to update existing results with each micro-batch

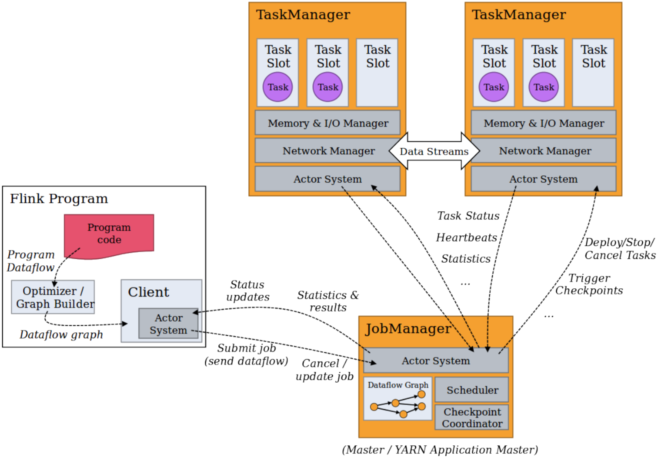
http://bit.do/cSi4L

Many applications are fundamentally based on streaming



Apache Flink is a big-data framework based on a distributed streaming dataflow engine

Flink's architecture

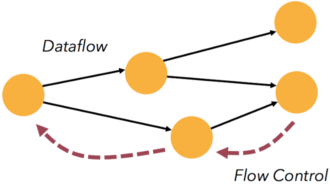


https://ci.apache.org/projects/flink/flink-docs-release-1.1/concepts/concepts.html

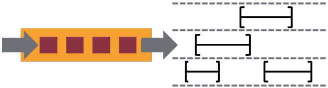
Streaming operators

Flink uses similar directed acyclic graphs (DAGs) of operators to Spark. But:

- In streaming mode, the DAG remains in place, and data flows along the DAG

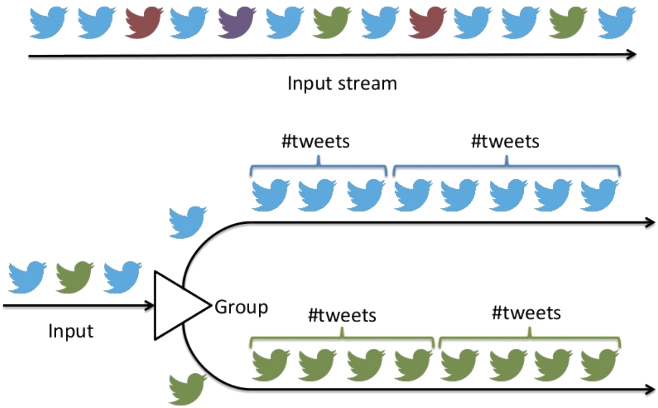


- Each operator works over a window of data items



http://flink.apache.org/features.html

Example: classify and count tweets



http://www.slideshare.net/tillrohrmann/apache-flink-streaming-done-right-fosdem-2016

Flink programs are compiled into an operator DAG

```
DataStream<String> lines = env.addSource(
                              new FlinkKafkaConsumer<>(…));
```

Source

```
DataStream<Event> events = lines.map((line) -> parse(line));
```

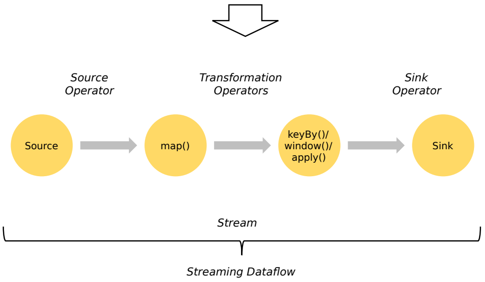
Transformation

```
DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());
```

Transformation

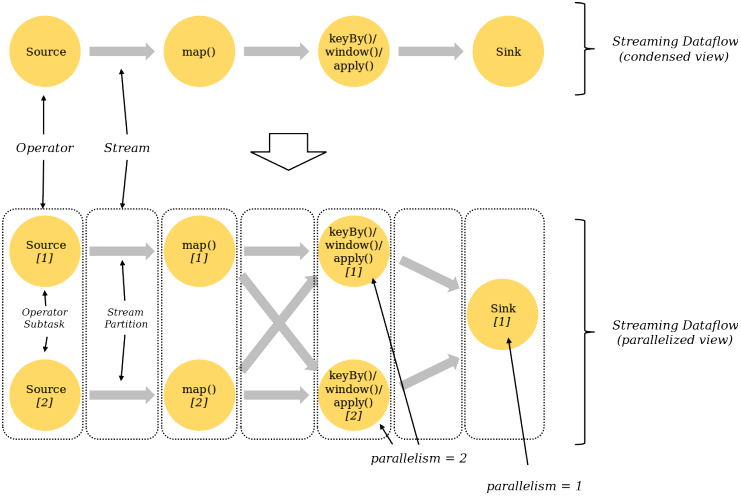
```
stats.addSink(new RollingSink(path));
```

Sink



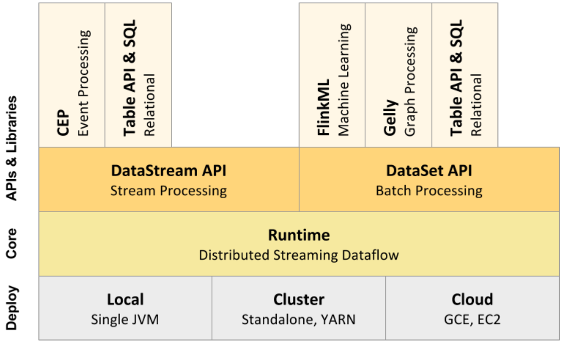
https://ci.apache.org/projects/flink/flink-docs-release-1.1/concepts/concepts.html

And each operator can be parallelized



https://ci.apache.org/projects/flink/flink-docs-release-1.1/concepts/concepts.html

Flink's stack



https://ci.apache.org/projects/flink/flink-docs-release-1.1/

Conclusion

- Processing big data is very difficult
  - Volume, Variety, Velocity
  - Parallel programming is hard!
- Cloud frameworks are being proposed to facilitate the developers' task
  1. MapReduce automatically parallelizes programs expressed as pairs of map/reduce functions
  2. Spark simplifies the development model by automatically compiling sequential code based on specific operators
  3. Flink extends Spark with data stream processing
- Many new frameworks are being proposed. Stay tuned for very fast progress in this exciting domain!