Major Key – INFO-H515

Key questions:

- what is “big data”, what are the characteristics of such data ?
- what is a “compute cluster”?
- how do clusters store data ?
- how are they programmed ?
- what are notions of efficiency for distributed algorithms ?
- what is “big data analytics” ?
- how do you perform machine learning on big data ?

Key understanding :

- understand the characteristics of big data, and the challenges these represent;
- know the principal architectures of Big Data Management and Analytics Systems, be able to explain the purpose of each their components, and be able to recognize and explain the key properties, strengths and limitations of each type of system and their components;
- understand the key bottlenecks in managing and analyzing massive amounts of data and be familiar with modern algorithms for overcoming these bottlenecks using parallel and distributed computation;
- actively use this algorithmic knowledge in the design and implementation of applications that solve common data management and analytics problems using different types of BDMAS;
- build applications using specific instances of each type of BDMAS.

1. Introduction

Big data is part of the big three :

- Big data
- Data science
- Data analytics

Limitation of classical approach

The traditional data analytics architecture approach is that moving the data to another infrastructure to perform computation does not scale !

GFS – HDFS – M/R

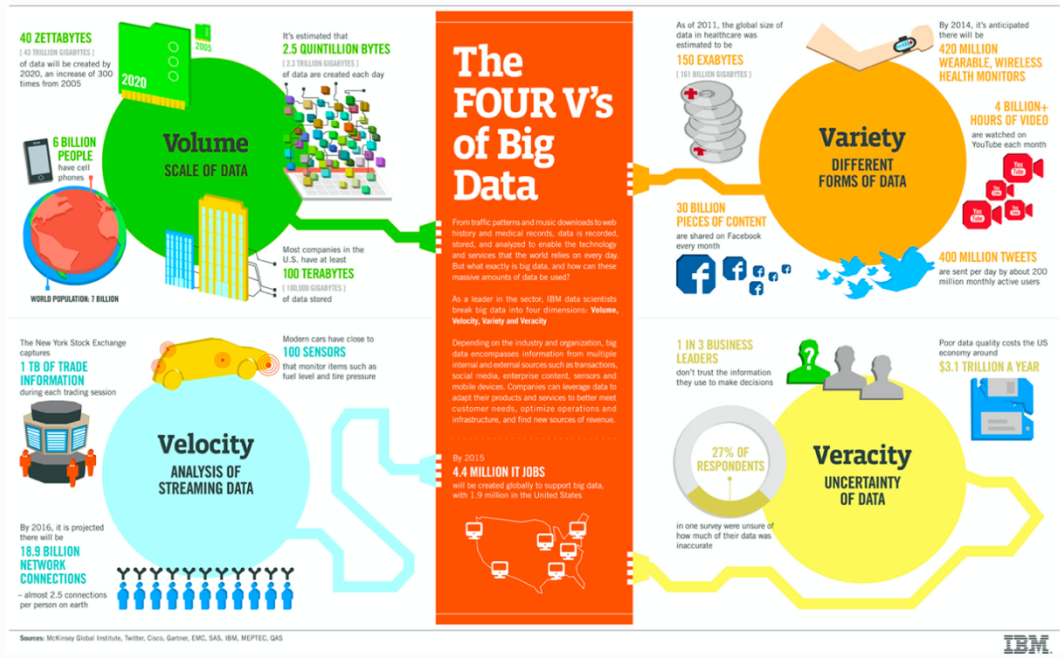
The solution developed by Google to tackle this issue (for distributed and scalable data analysis) is the “Google File System” which is a distributed file system for scalable storage and high-throughput retrieval and fault tolerant. On top of that, the Map Reduce paradigm was developed.(distributed batch processing).

The open-source alternative is Hadoop File System (HDFS) and M/R. Apache Spark is based on this.

Name	Purpose	Open Source Impl
Google File System	A distributed file system for scalable storage and high-throughput retrieval	Apache Hadoop • HDFS • M/R
Map Reduce	A programming model + execution environment for general-purpose distributed batch processing	
BigTable	A NoSQL Database	Apache HBase
Dremel, F1	A query language for interactive SQL-like analysis of structured datasets	Apache Spark/Drill
…	*Lots of research ongoing!*	….

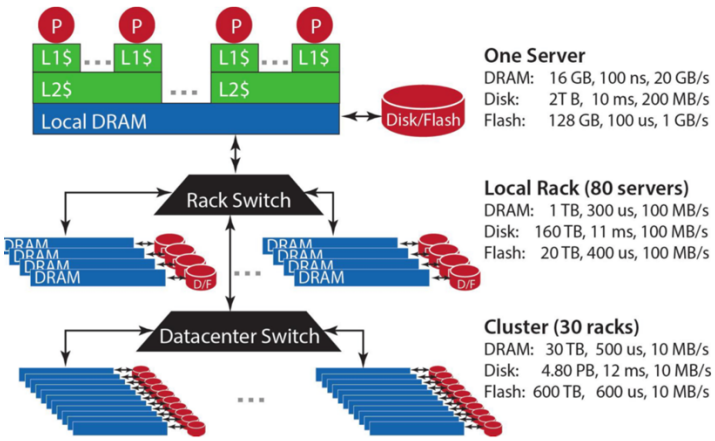
4 V’s (volume, variety, velocity, veracity)

For Big Data, the challenge is the management of the 4 V’s. Currently, the volume is constantly increasing, faster than velocity for instance and our computing power. It is creating some issues. The velocity is the speed of analysing data very fast. Variety is simply the diversity among the different types of data (text, image, etc.). Finally, veracity is the quality, the accuracy of the data.



Compute server – rack – data center

Compute servers consist of multiple CPUs (possibly with multiple cores per CPU), and attached hard disks. A group of compute servers is a rack, many racks create a data center. (it is slower to access data between different racks that within the same rack)



Capacity : the amount of data we can store per server/rack/data center
Latency : the **time** it takes to fetch a data item, when we asked one server to another server of one rack to another rack.
Bandwidth : the **speed** at which data can be transferred to one server to another or one rack to another.

Conclusion:

- Huge storage capacity
- Latency between racks
  = 1/10 latency on rack level
  ≈ 1/10 latency on server level
- Bandwidth between racks
  = 1/10 bandwidth on rack level
  = ½ to 1/10 bandwidth on server level

Parallelism advantage – Maximal aggregate bandwidth

Component		Max Aggr Bandwidth
1 Hard Disk		100 MB/sec (≈ 1 Gbps)
Server	= 12 Hard Disks	1.2 GB/sec (≈ 12 Gbps)
Rack	= 80 servers	96 GB/sec (≈ 768 Gbps)
Cluster/datacenter	= 30 racks	2.88 TB/sec (≈ 23 Tbps)

- Scanning 400TB hence takes 138 secs ≈ 2,3 minutes
- Scanning 400TB *sequentially* at 100 MB/sec takes ≈ 46,29 days

Challenges :

- It should be **scalable**, it should allow growth without requiring re-architecting algorithm or application
- **Fault-tolerance** (with 1000 computer, failure happens every day) → solution : **redundancy and/or re-execution**

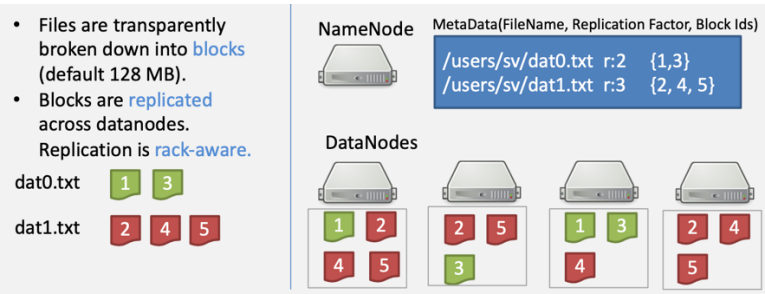
Mean Time To Failure (MTTF) for a disk (1 000 000 hours → 114 years)
Annual Failure Rate (AFR) for a given MTTF = # hours in a year/MTTF = 365 * 24 / 1 000 000 = 0,876% chance of failure a year.
→ if we have 100 000 disks : # disks $ AFR = 100 000 * 0,876% = 2 disks/days that fail.

In order to avoid issues due to the failures, a solution is the redundancy, which implies to store multiple copies of one file. → replicate blocks across datanodes.

HDFS Architecture

HDFS has a master/slave architecture.

- Master = NameNode (NN) that manages the file system and regulate access to files by clients.
- Slaves = DataNodes (DN), usually, one per server in the cluster, manage storage attached to the server that they run on.



• Files are transparently broken down into blocks (default 128 MB).
• Blocks are replicated across datanodes. Replication is rack-aware.

• Optimized for:
  - **Large files**
  - Read throughput
  - Appending writes (files are append-only)
• Replication ensures
  - Durability
  - Availability
  - Throughput

Files are **immutable**, optimized for sequential access.

- New files that are created can only be appended to (no random-access)
- Once a file is created, it is immutable. Changing a file requires re- creating it.

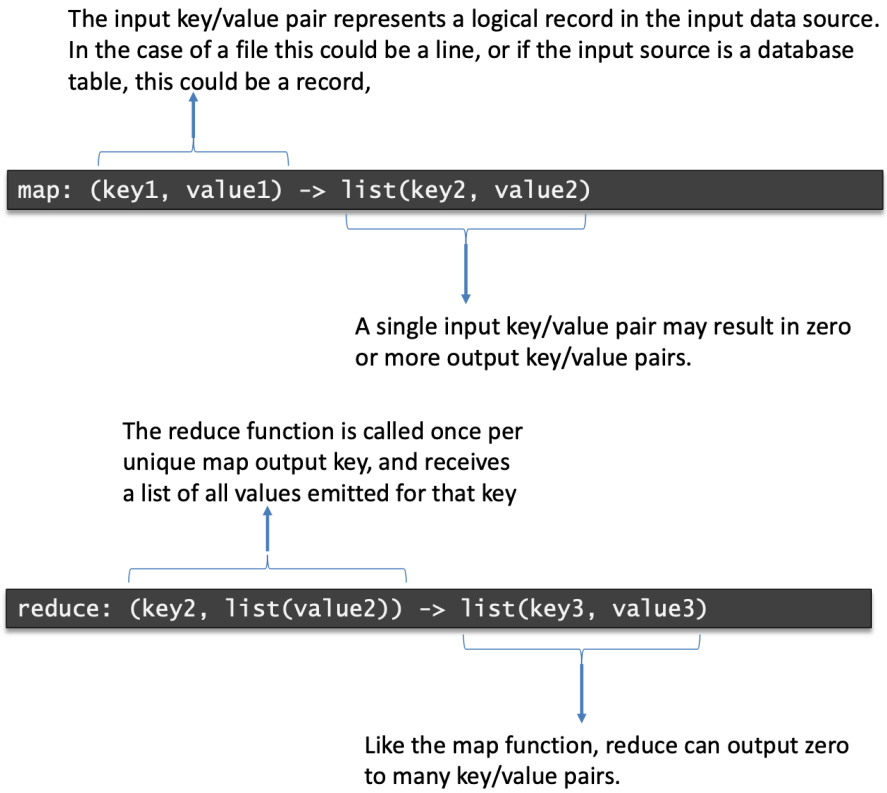
Map – reduce

Map : map operation to each logical record' in our input in order to compute a set of intermediate key/value pairs

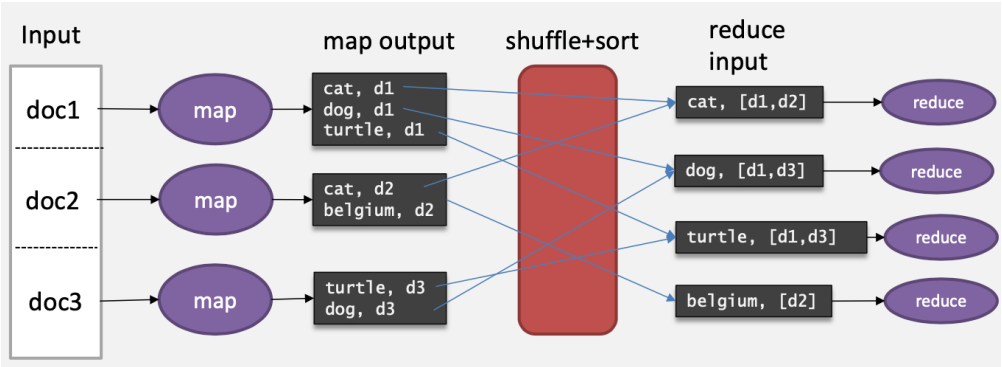
Reduce : applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately

→ It allows to parallelize large computations easily and to use reexecution as the primary mechanism for fault tolerance.

The map-reduce job has two functions, Map and Reduce.



Multiple map functions (map tasks) (at most one for each key-value pair) and reduce functions (reduce tasks) (at most one for each unique key output by the map) can be spawn in parallel !



M/R Execution – Hadoop V1 architecture

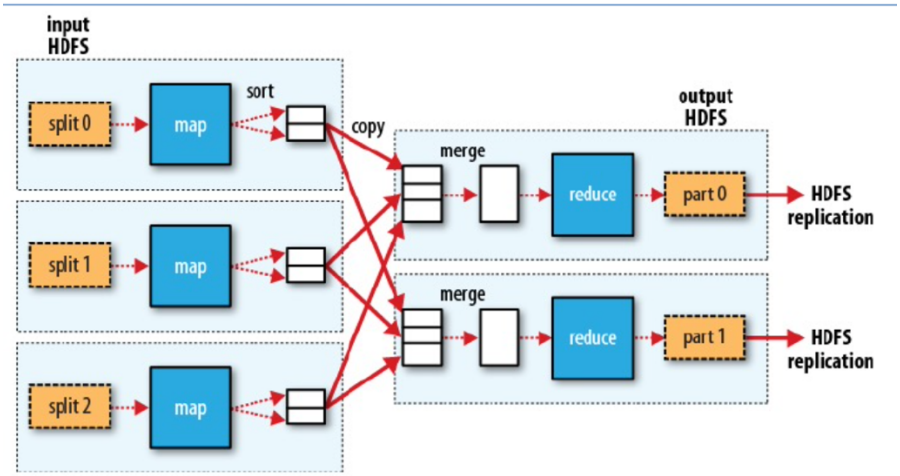
- Master = JobTracker – accepts jobs, decomposes them into map and reduce tasks, and schedules them for remote execution on child nodes.
- Slave = TaskTracker – accepts tasks from Jobtracker and spawns child processes to do the actual work.
- Idea: ship computation to data

Example :

1) The JobTracker accepts M/R jobs.
2) If the input is a HDFS file, a map task is created for each block and sent to the node holding that block for execution.
3) Map output is written to local disk.
4) The JobTracker creates reduce tasks intelligently
5) Reduce tasks read the map outputs over the network and write their output back to HDFS.

- Load balancing: The JobTracker monitors for stragglers and may spawn additional map on datanodes that hold a block replica. Whichever node completes first is allowed to proceed. The other(s) is/are killed.
- Failures: In clusters with 1000s of nodes, hardware failures occur frequently. The same mechanism as for load balancing allows to cope with such failures

Hadoop (job scheduler) tries ideally to perform the map task where the data is located (if not busy). If it is not possible, it will be done in another server of even another rack (inter-rack network transfer)



We have as much “map” as we have “split”. The split is a logical division of the input data while the block (HDFS block) is a physical division of the data.

The size of the split is defined by the user. (if not defined, HDFS uses the default block size). The split can be larger or smaller than the block size.

Each map function will create a key/value pair for each key. Then, for the reduce part, we will merge all the key/value pairs with the same key and reduce them.

We have as many maps than we have splits.

Each map task partitions its output; the number of partitions equals the number of reducers that will run later. Each partition is sorted already on the map-side. (One partition can still have many distinct keys!)

The reducers copy their respective partition from the mappers, and merge them into one big sorted file.

The net result is a distributed merge-sort between the map and reduce phases.

2. Distributed processing with Spark

Apache Spark is a fast, in-**memory** distributed data processing engine.

RDD

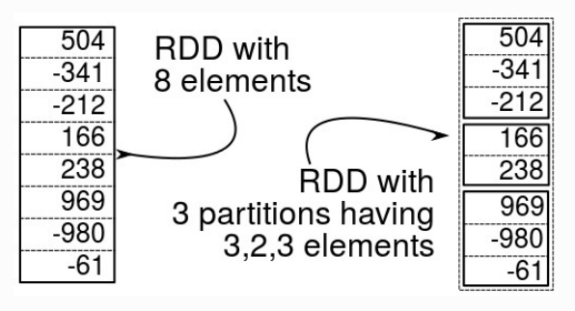
RDD = Resilient Distributed Dataset

- RDD = collection of elements
- RDDs can contain any types of objects as elements , including user-defined classes.

Under the hood, Spark will automatically distribute the data contained in RDDs across your cluster and parallelize the operations you perform on them.

RDDs are distributed

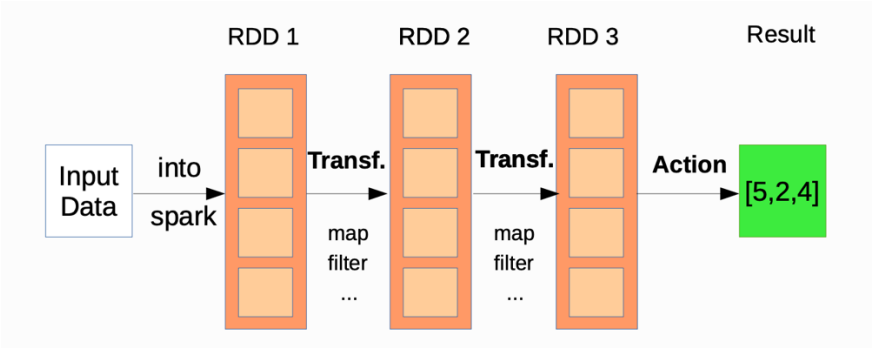
- Each RDD is broken into multiple pieces called partitions, and these partitions are divided across the nodes in the cluster.
- Partitions are operated on in parallel
- The partitioning process is typically done automatically by Spark
- User can (and sometimes should) influence partitioning process.



RDDs are immutable

- They cannot be changed after they are created

- Immutability rules out a significant set of potential problems due to updates from multiple threads at once.



Basic workflow in Spark:

- Generate **initial RDDs** from external data.
- **Transformations** create new RDDs from existing ones
- **Actions** transform RDDs into normal programming language values (lists, numbers, file, ...).
-

RDDs are **Resilient**

- RDDs are a deterministic function of their input. This plus immutability also means the RDDs parts can be recreated at any time.
- In case any node in the cluster goes down, Spark can recover the parts of the RDDs from the input and pick up from where it left off.
- Spark does the heavy lifting for you to make sure that RDDs are fault tolerant.

Operations on RDD

Transformations (operations on RDD that return a new RDD)

- Filter (usefull to clean-up) (keep only the elements from the input RDD that pass the filter function)
- Map (it pass all the elements of the input RDD through the function, we have have a new value for each element in the resulting RDD) (return type could differ from the input type)
- flatMap (it takes each element from the existing RDD and it can produce 0,1 or many outputs for each element !)

- Operations on one RDD that create one new RDD :
  - Sample
  - Distinct (expensive due to shuffling all the data across partitions to ensure that we only received one copy)
    - **Shuffling**: Physical movement of data between partitions is called shuffling. It occurs when data from multiple partitions needs to be combined in order to build partitions for a new RDD .
- Operations performed on two RDDs to produce one RDD
  - Union (can contain duplicates)
  - Intersection (keep common elements)(remove duplicates)(expensive due to shuffling)

- Subtract (only keep element unique to the first RDD) (expensive du to shuffling)
- Cartesian product (returns all possible pairs of a and b (where a and b are the source RDDs)) (great to compare the similarity between all possible pairs)

$$\{1,2,3,4\} \times \{a,b,c\} = \begin{Bmatrix} (1,a), & (1,b), & (1,c), \\ (2,a), & (2,b), & (2,c), \\ (3,a), & (3,b), & (3,c) \end{Bmatrix}$$

Actions

Operations that return a final value to the driver program, it forces the evaluation of the transformations for the RDD that were called on

- Collect
  - Collect operation retrieves the entire RDD and returns it to the driver program in the form of a regular collection or value.
  - If you have a string RDD, when you call collect action on it, you would get a list of strings.
  - This is quite useful if your Spark program has filtered RDDs down to a relatively small size and you want to deal with it locally.
  - The entire dataset must fit in memory on a single machine as it all needs to be copied to the driver when the collect action is called. So collect action should NOT be used on large datasets.
  - Collect operation is widely used in unit tests, to compare the value of our RDD with our expected result, as long as the entire contents of the RDD can fit in memory.
- Count (count how many rows we have in the RDD)
- CountByValue (useful to check if we have duplicate rows, we want to count the unique row value we have)
- Take (could give a biased collection)
- SaveAsTextFile
- Reduce
  - The reduce action takes a function that operates on two elements of the type in the input RDD and returns a new element of the same type. It reduces the elements of this RDD using the specified binary function.
  - With the reduce action, we can perform different types of aggregations.
  -

- **Transformations** are operations on RDDs that return a new RDD, such as map and filter.
- **Actions** are operations that return a result to the driver program or write it to storage, and kick off a computation, such as count and collect.

Pair RDD

- A lot of datasets we see in real life examples are usually key value pairs.

Examples : A dataset which contains passport IDs and the names of the passport holders.

- The typical pattern of this kind of dataset is that is each row is one key mapped to one value or multiple values.
- Spark provides a data structure called Pair RDD instead of regular RDDs, which makes working with this kind of data simpler and more efficient.
- **A Pair RDDs is a particular type of RDD that can store key-value pairs.**

- **Pair RDDs are allowed to use all the transformations available to regular RDDs, and thus support the same functions as regular RDDs.**
- Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.

Map & MapValue (transformations)

Most of the time, when working with pair RDDs, we don't want to modify the keys, we just want to access the value part of our Pair RDD. Since this is a typical pattern, Spark provides the **mapValues** function. The mapValues function will be applied to each key value pair and will convert the values based on mapValues function, but it will not change the keys.

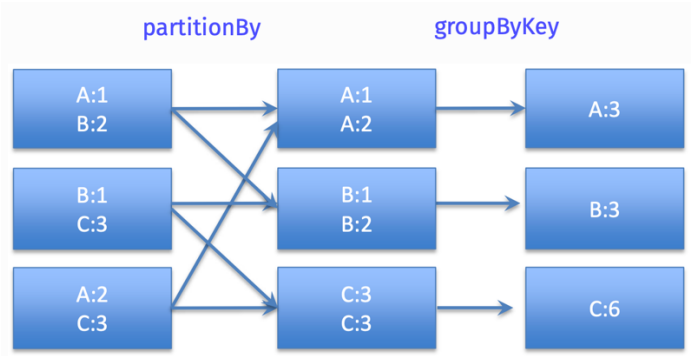
Aggregation

When our dataset is described in the format of key-value pairs, it is quite common to aggregate statistics across all elements with the same key.

We have looked at the reduce actions on regular RDDs, and there is a similar operation for pair RDD, it is called **reduceByKey**. ReduceByKey runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key.

Considering input datasets could have a huge number of keys, **reduceByKey operation is not implemented as an action that returns a value to the driver program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.**

- GroupByKey
- SortByKey, SortBy
- Partition by (it can reduce the amount of shuffle for many operations (join, groupbykey, reducebykey, combinebykey, etc.)

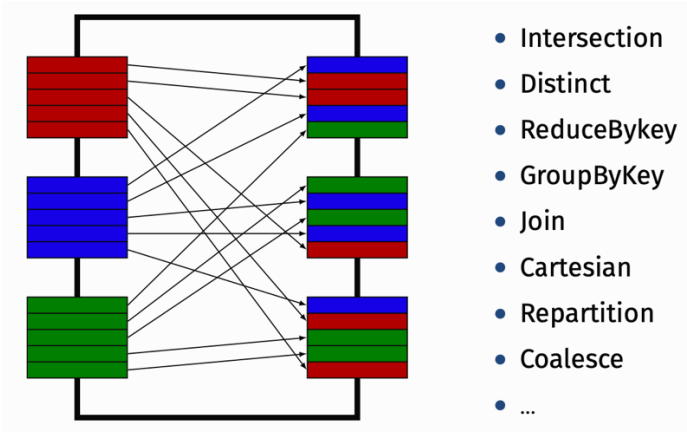


Spark program execution

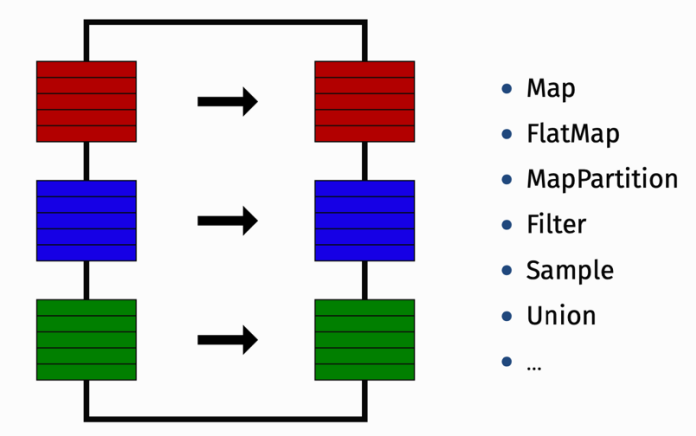
Lazy evaluation : Even though new RDDs can be defined any time, they are computed by Spark in a lazy fashion, which means they are not evaluated until they are needed for the result of an Action. We can apply transformations (filter, map, etc.) but it is the action (reduce, count, take, etc.) that we imply some computation, some evaluation.

Transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action. Rather than thinking of an RDD as containing specific data, it might be better to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations. Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together.

Wide transformations – transformations that require a shuffle phase



Narrow transformations – transformations that do not need a shuffle phase (grouped in a single stage → pipelining)



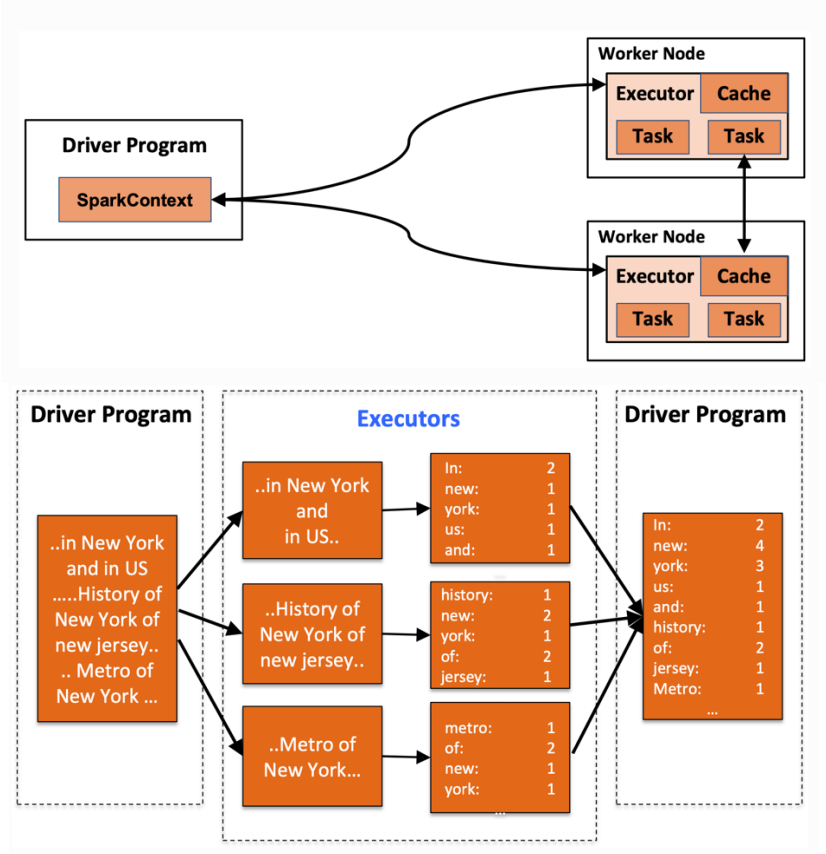
For wide transformation, you need to send data over the network while for narrow you do not need to send data over the network.
Persistence : when we use many actions on the same RDD multiple times, it is better to keep the RDD in memory rather than recompute it every time. With “persist”, we can store an RDD in memory with Spark.

Which storage level should be chosen ?

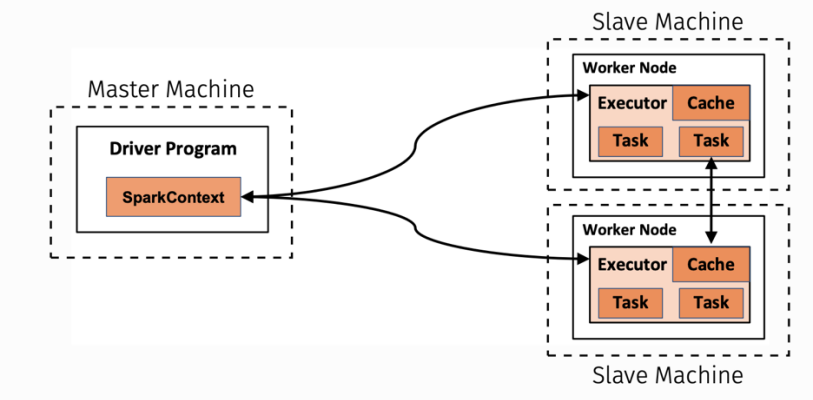
- Spark’s storage levels are meant to provide different trade-offs between memory usage and CPU efficiency.
- If the RDDs can fit comfortably with the default storage level, MEMORY_ONLY is the ideal option. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- If not, try using MEMORY_ONLY_SER to make the objects much more space-efficient, but still reasonably fast to access.
- Don’t save to disk unless the functions that computed your datasets are expensive, or they filter a significant amount of the data.

If we attempt to fit too much data in memory : ◦ Spark will evict old partitions automatically using a Least Recently Used cache policy. (for MEMORY_ONLY, Spark will re-compute the partitions the next time they are needed, for MEMORY_AND_DISK, Spark will write the partitions on the disk)

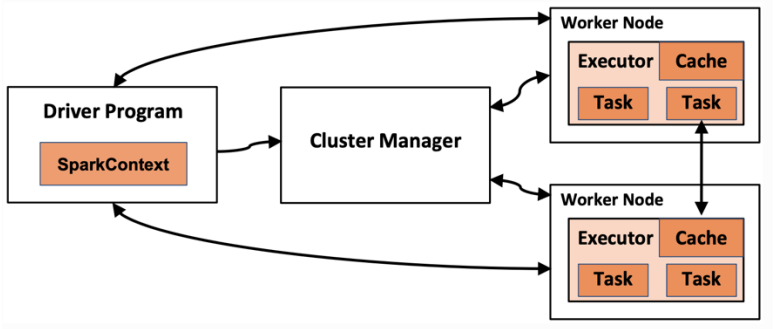
Master-Slave architecture



In spark, all components can run on the same machine but they may run on different machines



Running spark in cluster mode



The cluster manager is a pluggable component in Spark. The cluster manager abstracts away the underlying cluster environment so that you can use the same unified high-level Spark API to write Spark program which can run on different clusters.

1) The user submits an application using the spark-submit script ◦ available in Spark's bin directory
2) Spark-submit launches the driver program and invokes the main method specified by the user.
3) The driver program contacts the cluster manager to ask for resources to start executors.
4) The cluster manager launches executors on behalf of the driver program.
5) The driver process runs through the user application. Based on the RDD or dataframe operations in the program, the driver sends work to executors in the form of tasks.
6) Tasks are run on executor processes to compute and save results.

3. Distributed stream processing

Back to the V's, velocity refers to the speed at which data is being produced, captured or refreshed. Data is not just big, it is also fast (a lot of data are generated every minute worldwide) and it requires (near) real-time analysis (stock market, out-of-stock detection, etc.)

Even if an application does not require **real-time analysis**, **use of data streaming is widespread because of the λ-architecture.**
Big Data Software Frameworks that allow analysis of high-velocity data are called **data streaming frameworks.** We will discuss two, with fundamentally different architectures:

- Tuple-at-a-time
- Mini-batching

The λ-architecture

Issue with mutable data (données modifiables)
Mutable data is corruptible :

- Bugs will be deployed to production software over the lifetime of a data system.
- Humans can err, therefore operational mistakes will be made at some point.
- You must design to safeguard against data corruption.

→ Key observation: as long as an error does not lose or corrupt good data, we can fix what went wrong.

Immutability

Because mutable data is corruptible, we must keep all the master data immutable !

- An immutable data system captures a historical record of events

- Each event happens at a particular time, and remains true forever.
- You can only append new events; never delete or modify existing event records.
- Filter on the timestamp to see what is true at a particular moment in time.

Big Data technologies allow to store all raw data so that it can be used later gain new insights and create new data-driven products, which we haven't yet thought of!

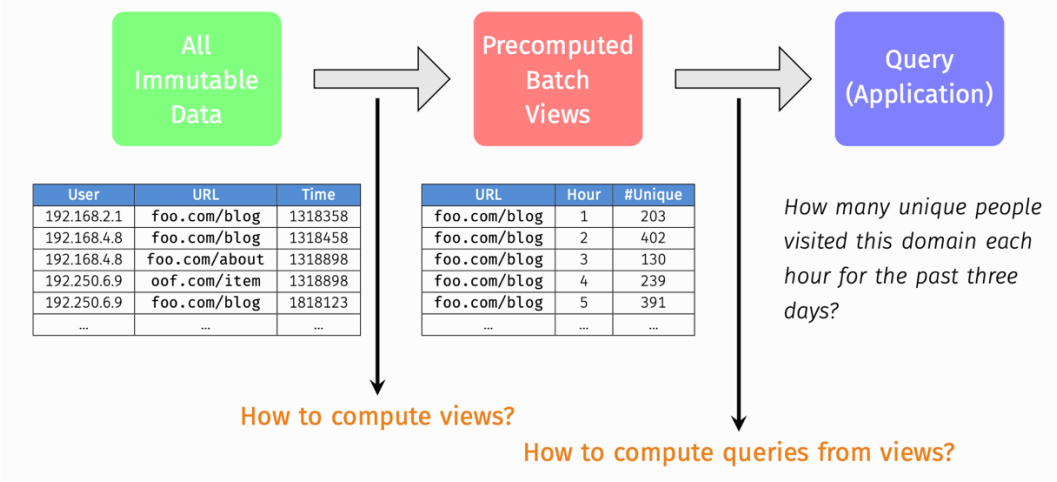
Desired properties of a big data system

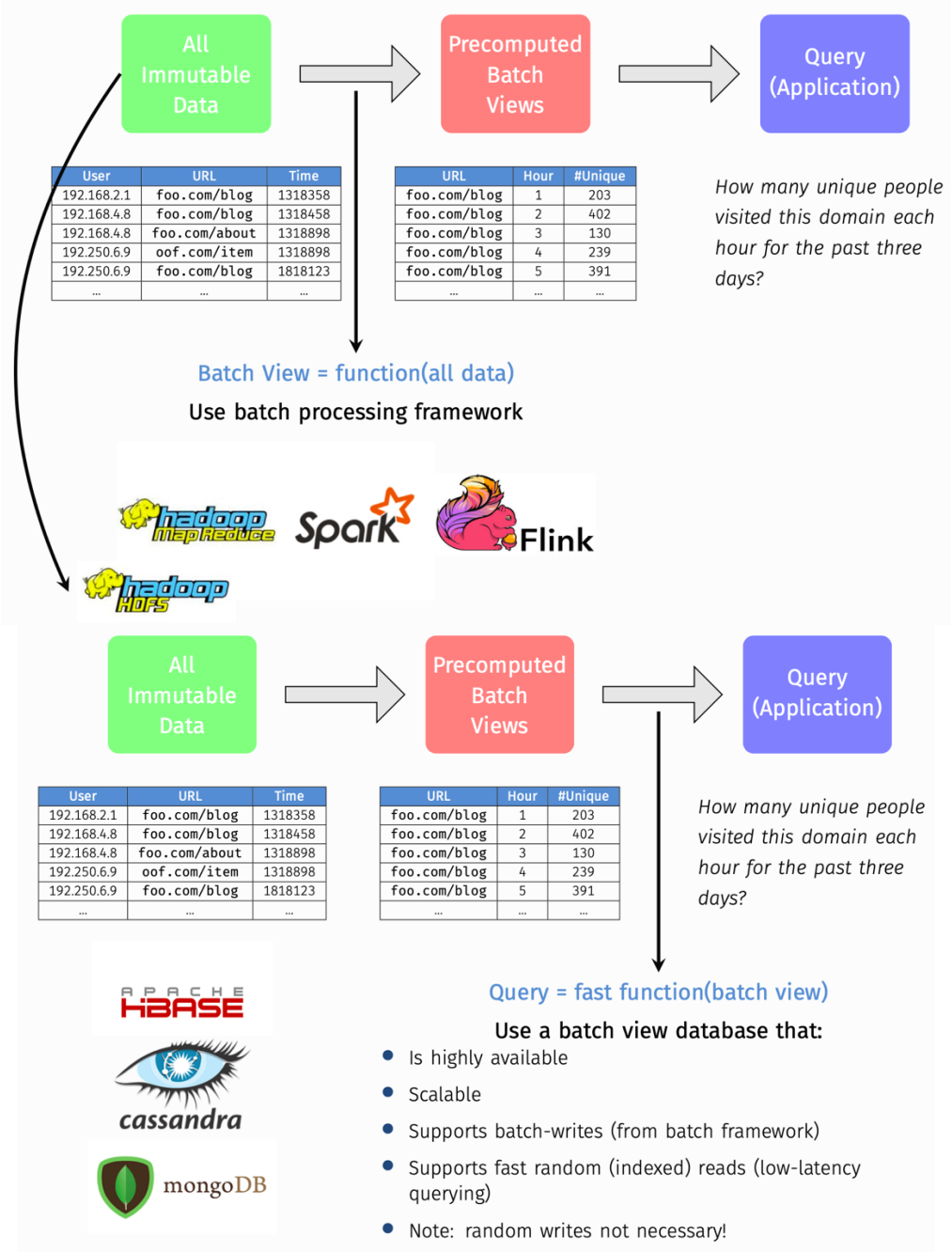
- Robust and fault tolerance (under both machine and human failures)
- Scalable: maintain performance under increasing load by adding resources.
- Answer pre-defined queries with low latency
- Support ad-hoc querying
- Low-latency updates
- Extensible/general

A query is a function applied to all the data. A query hence transforms our immutable raw data into useful information.

However, how can we deal with a large dataset with small latency ?
→ Computing query answer on the fly produces high latency! **The idea is to perform precomputed batch views !** Precompute views from which the answers to pre-defined queries can be read/computed with low latency.

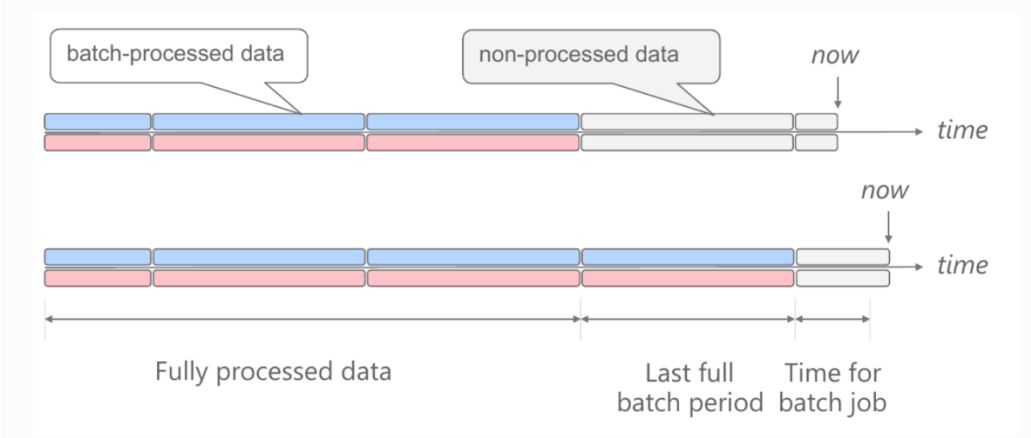
We can do this, prepare the views because we already know the query that we want ! We cannot do this on the fly ! The view is specific for its related query, it cannot be used for another query !





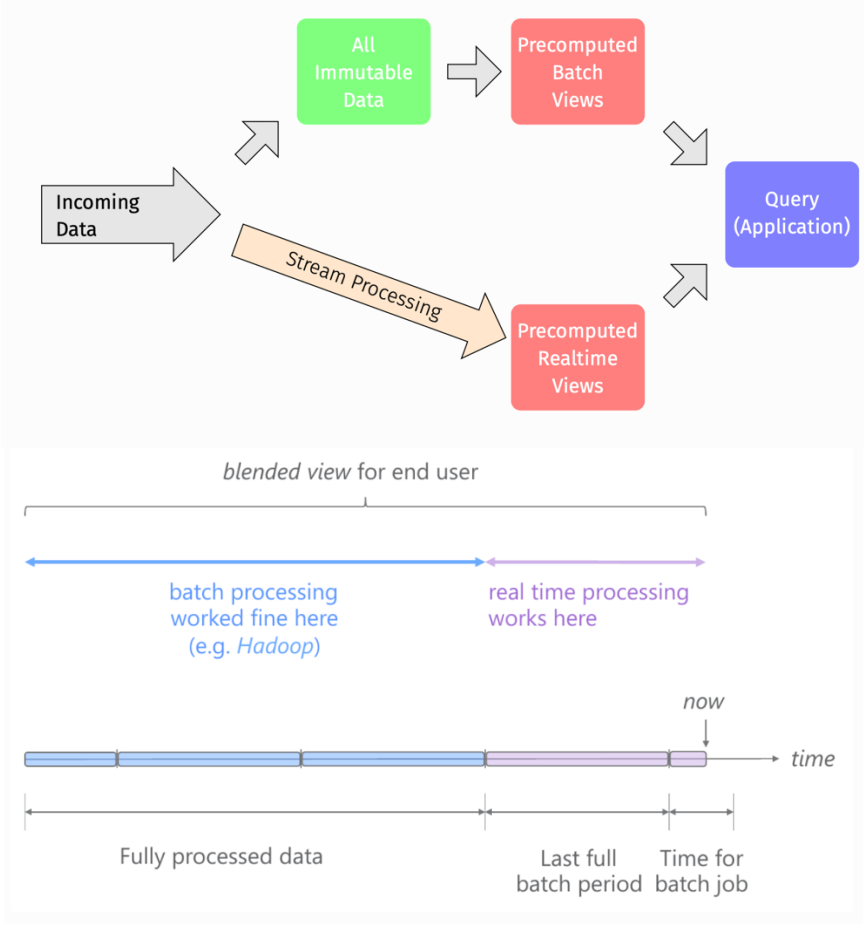
How to keep the batch view up to date ? Incrementalize the batch computation as much as possible is not the right answer because batch computation can be relatively slow.

The solution is to treat the fresh data separately : **real-time stream processing**



Let's add real-time stream processing :
Realtime view = function(realtime view, new data)

- Exact if possible, otherwise approximate (see upcoming lecture)
- Approximate realtime-views become exact batch view over time.
- Realtime view database must support random writes (= more complex).



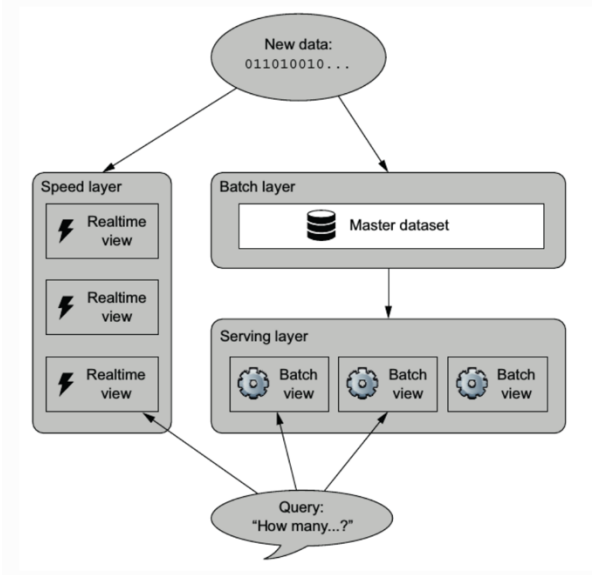
The λ-architecture in summary :

It is a big data architecture that distinguishes between three layers :

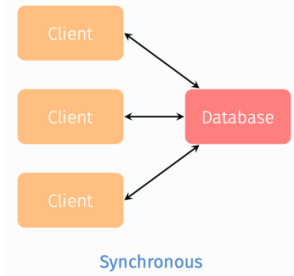
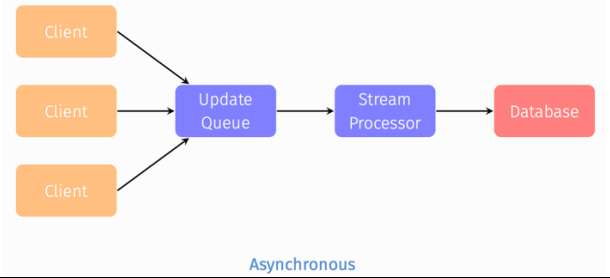
- The batch layer :

    - It keeps an immutable, historical log of all data (it is also known as the data lake)
    - It computes batch views from this master data
    - It allows ad-hoc (but high-latency) querying
- The serving layer :
    - It stores the batch views
    - It uses these views to respond to pre-defined queries
- The speed layer :
    - It computes realtime views on streaming new data
    - It uses these views to augment the batch views for responding to pre-defined queries

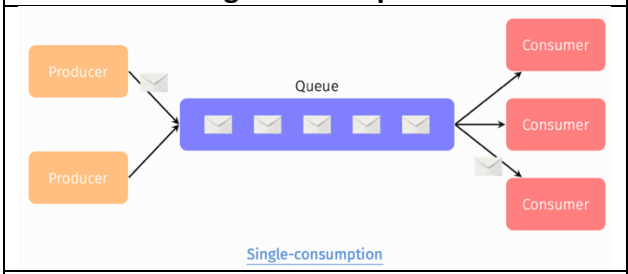
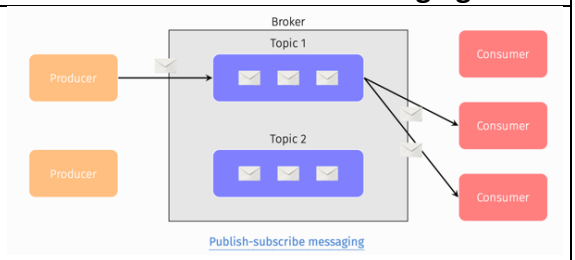
→ The λ-architecture is a meta-architecture: concrete frameworks for each layer can be chosen on a case-by-case basis.



Message Queues: the sources of Fast Data

Two ways of processing updates	
Synchronous	**Asynchronous**
	
Direct connection between client and DB	Client submits update to a queue and continues work without waiting
The client waits until update complete	Updates are processed by stream processor when possible

(+) The client knows when the update is done	(+) Load spikes are easily handled
(-) Spikes can overload the database	(-) Special mechanism required to acknowledge update is processed
	Approved and used approach

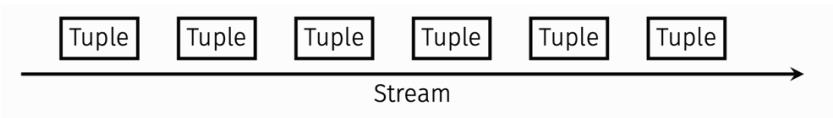
Two types of message queues	
Single-consumption	**Publish-subscribe messaging**
	
Also known as point to point queue or message bus/queue	The queue is organized into topics
Producers (senders) push the message to the queue	A broker is responsible for queue maintenance and message delivery
Consumers (receivers) pop messages from the queues	Producers (publishers/senders) push messages to topics on queue
Parallelism through (round-robin) distribution over consumers	Consumers (subscribers/ receivers) receive messages from the topics that they are subscribed to. The broker decides when old messages can be deleted
(-) Every message is consumed by only one consumer ! By default, a message is deleted from the queue when consumer fetches message (fault-tolerance problem !)	(+) Every message is processed by all consumers subscribed to the message topic !
	(-) Scalability (parallelism) ?

Apache Kafka is a distributed messaging / streaming platform based on partitioned publish-subscribe :

- Topics are partitioned
- Partitions are distributed for scalability and replicated for fault-tolerance.

Tuple-at-a-time processing

Apache Storm & Heron are Distributed and Fault-Tolerant real-time computation



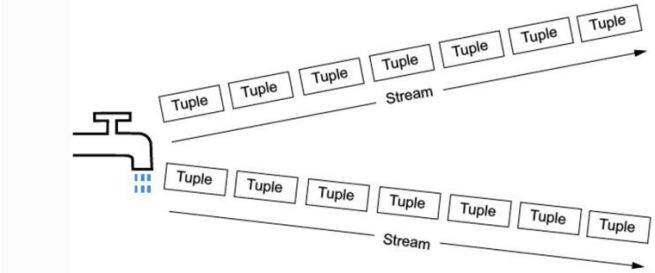
Tuple :

- Core unit of data

- Immutable set of key/value pairs

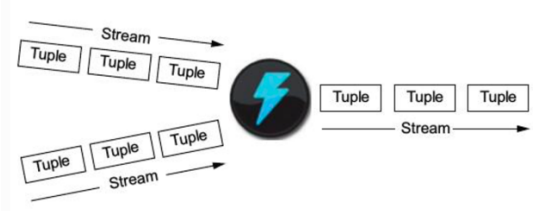
Stream :

- Unbounded sequence of tuples



Spout :

- Data source
- Emits streams
- Example : a kafka spout on a particular topic

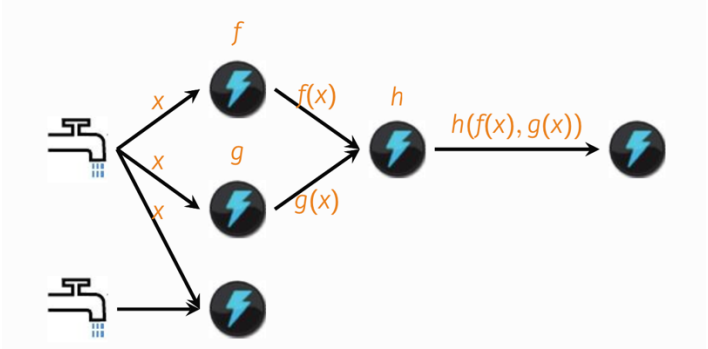


Bolt :

- Core function of streaming computation
- Receive tuples and process them, e.g.:
  - Write to a data store
  - Lookup a tuple in a data store
  - Perform arbitrary computation
  - Emit additional streams

Topology :

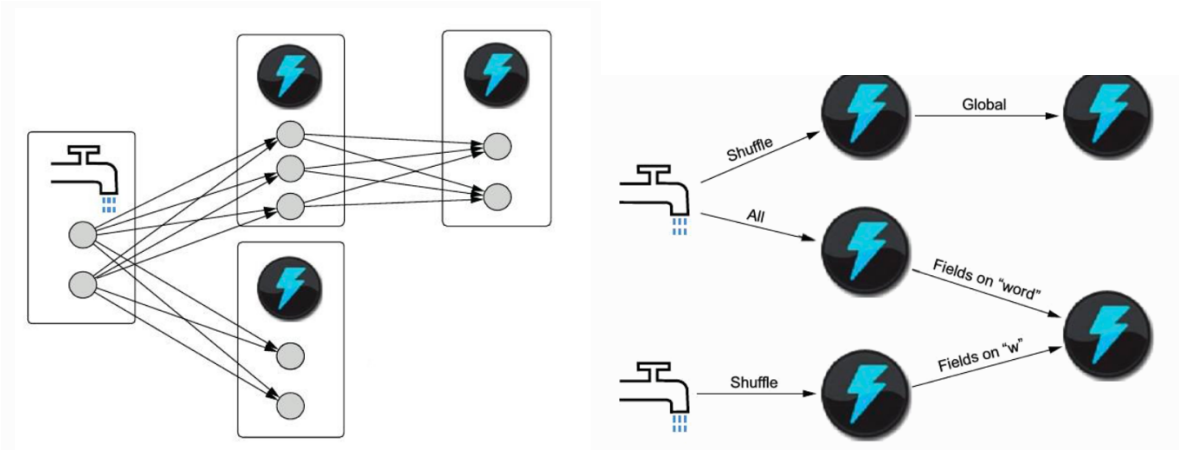
- DAG of spouts, bolts, streams (DAG : A directed acyclic graph (DAG) is a conceptual representation of a series of activities)
- Data flow representation of computation
- Tuples are pushed through the DAG (streaming computation) starting from spouts



When a tuple is emitted, it goes according to the grouping specified in the topology programmer :

Shuffle grouping : pick task in a round-robin fashion (round robins : a tournament in which each competitor plays in turn against every other)

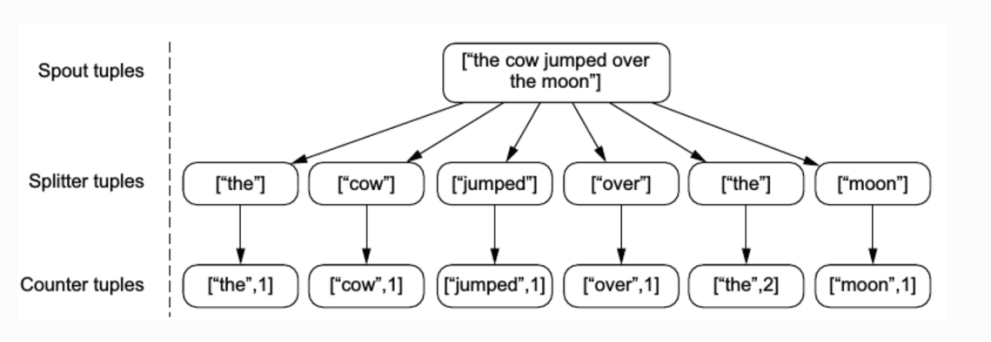
- Field grouping : consistent hashing on a subset of tuple field (only “word” or “w”)
- All grouping : send to all tasks the tuple



Bolt has a smart mechanism that tracks where tuples have been processed and if necessary, it recreates the messages, tuples that were lost in order to guaranty, that every edge of the tree has been followed !

- A spout tuple is not fully processed until all tuples that “descend” from it have been completely processed.
- The descendant-relationship (encoded by emit) is recorded in a tuple tree.
- If the tuple tree is not completed within a specified timeout, the spout tuple is replayed. This is known as at least once processing semantics.

Replaying a tuple always safe ?
→ The programmer should ensure that when we are in a **at least once** set-up, we are not unnecessarily updating a counter if the task is repeated



Apache Storm/Heron
- Distributed and Fault-Tolerant real-time computation
- Tuple-at-a-time processing model

Mini-batching

It is called mini-batches because it is computed online, in real time while batches are computed offline

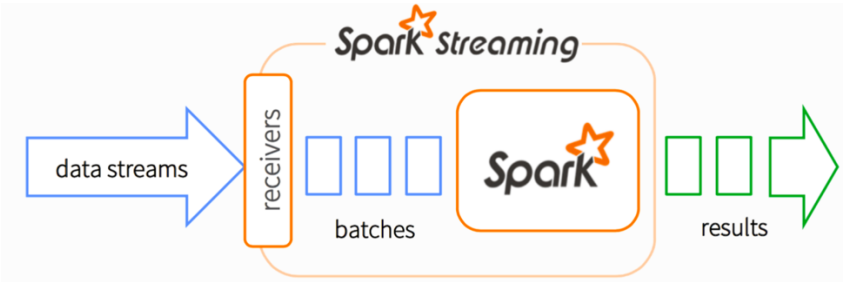
Spark Streaming

- Distributed and Fault-Tolerant (near) real-time computation
- Developed as a separate library on top of spark in 2015
- Re-uses spark concepts and programming model (RDDs, functional transformations, failure recovery through re-execution).

Key difference with Storm:
Processes stream tuples in (mini)-batches :

- Has exactly-once semantics. → **while Storm uses the "at least once"**

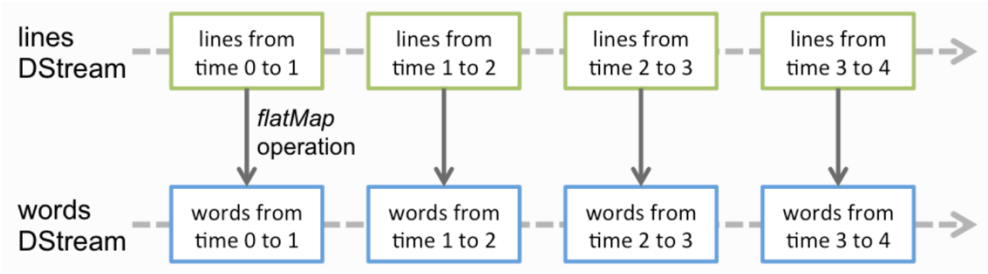
Spark streaming uses RDD and you can work with them likes any normal spark computations.



Mini-batching

- Spark streaming partitions the input stream into disjoint time intervals
- The data in each time interval becomes a (mini-batch) RDD; the stream of mini-batch RDDs is called a Discretized Stream (DStream)
- Mini-batch RDDs are normal RDDs. Therefore, normal spark operators can be used to transform/act on mini-batch RDDs (possibly jointly with normal RDDs)
- Each transformed mini-batch RDD can then be saved to HDFS, or stored in a DB, or communicated to a message queue, ...

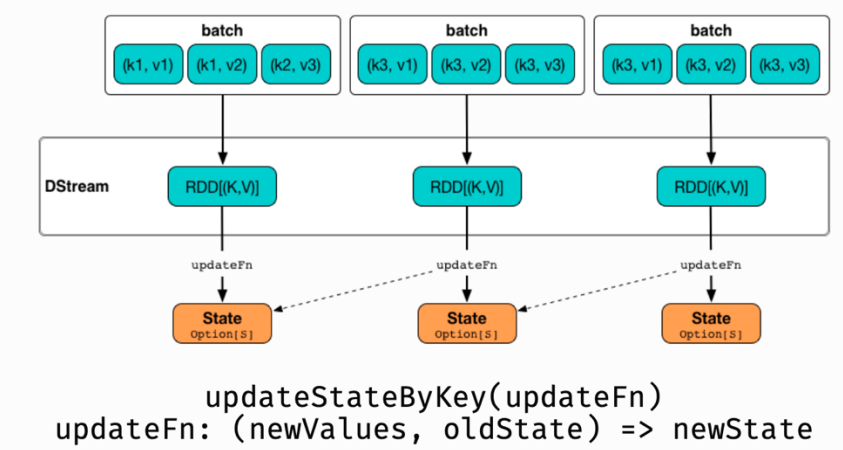
Example – word counting : we start with a sequence of RDD and end up with a sequence of RDD (one to one correspondence)



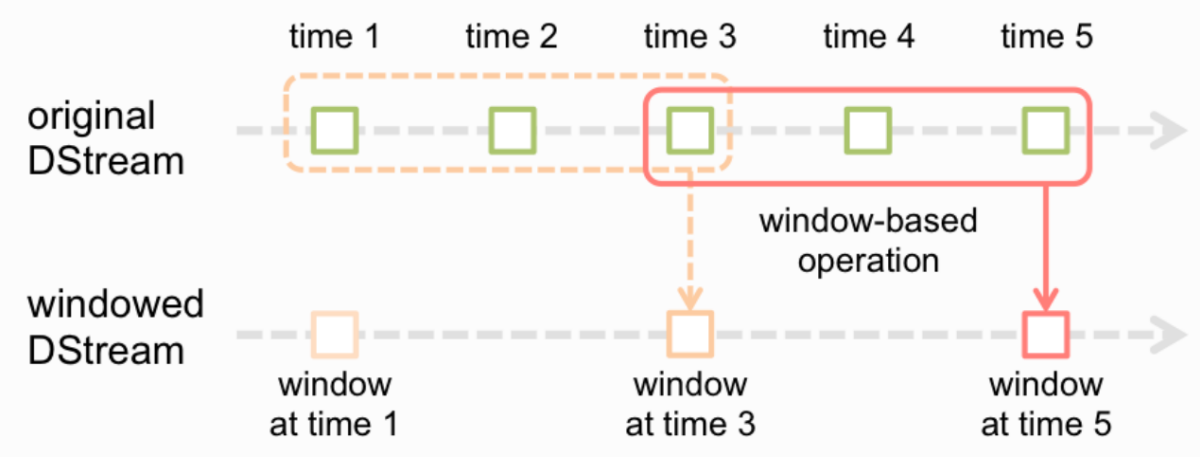
DStream

- Stream of mini-batch RDDs (interval width is configurable)
- Operations on the stream (flatMap, filter, ...) are performed on each mini-batch RDD in the DStream separately.
- The result is a new DStream.

However, until now, we have a stream of result but not a global result ! We need to global state storing the global result ! (orange part)



Windowing :
We want to maintain the function running of the last ten minutes, and we update it every one minute. We will shift the window to the next time unit, mini-batch to keep the last 10 minutes and the result of the time-window.

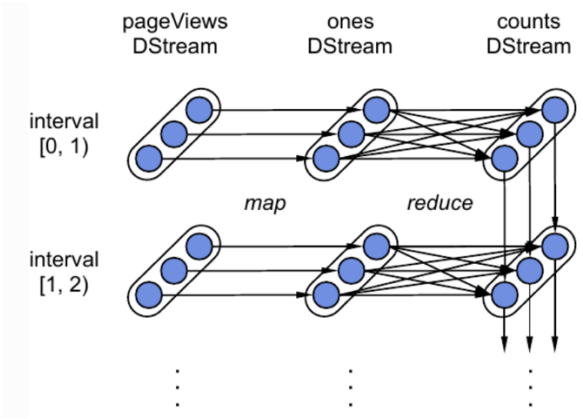


(the original stream is updated every 1 time unit with the window is updated every 2 time unit)

Question: Assume that we want to maintain a running count of every word seen over the past 10 minutes, and update this running count whenever new data arrives. How do we do this ?
Answer: Use windowing operations:

- Every window has a length (3 in figure) and sliding interval (2 in the figure), both must be multiples of the batch interval
- Window length = the duration of the window
- Sliding interval = the interval at which the window operation is performed

Fault-tolerance semantics



Remember

- RDDs are immutable and deterministically re-computable given the RDD's lineage graph (graph of transformations)
- Likewise, if we track lineage of operations on mini-batch RDDs in a DStream, then we can re-compute any such RDD when needed.

Some caveats:

- Lineage graph may become very big, especially when state is maintained.
- Add checkpointing to allow truncating the lineage graph
- Recomputation means that input data must be available. To ensure this, spark replicates input stream data.
- Enable write ahead logging to ensure no data loss.

Possible processing guarantees:

- At most once: Each record will be either processed once or not processed at all.
- At least once: Each record will be processed one or more times. There may be duplicates.
- Exactly once: Each record will be processed exactly once - no data will be lost and no data will be processed multiple times.

4. Stream and Big Data Algorithm

Requirements:

- Results generated on the fly
- Memory should scale sub-linear
- Constant-time processing of incoming data

Heavy Hitters

Karp's solution

Set solution

"Identify all items (heavy hitters)that occur more than Theta N times in a stream S of N items" (Theta in [0,1])

We do not know upfront which combinations will be frequent.

Question: How much space needed for an *exact* solution?
Answer: worst case at least Ω(n log(N/n)) bits where

- n = number of possible symbols and
- N = length of the total stream.

Proof:
Let θ = 50%, and we have already seen N/2 - 1 elements of the stream

Configuration: bag (no order) of symbols seen so far
For any two configurations the system needs to be in a different *state*
Hence, we need at least log(#configurations) space

If we want to have a complexity lower than upward, we need to sacrifice the "exactness" of the solution !

"Given a stream, identify all items that occur more than 20% of the time"

Remove 5 elements of a different color to get S':
If ● was a heavy hitter, it still is!
Hence, removing 5 elements of different color gives us a smaller set, but we keep all heavy hitters.
– Can be done repeatedly
– Until no longer possible to remove 5 with distinct color
Answer is a subset of the remaining (at most 4) colors:

Here the idea is that we have a document of N items, and we randomly remove 5 items of different colors (why 5, because Theta = 20%). Then, the items that were heavy hitters should remain heavy hitters. We continue until we cannot remove 5 different items. We obtain a result of 4 remaining items where probably at least one of them is a heavy hitter BUT we cannot say which one is a heavy hitter.

→ Problem : we therefore have false positive heavy hitters ! One solution could be to make a second run (if it is possible (for document it is OK, for Stream, it is not OK)) to only count the 4 proposed colors, items.

By setting Theta, we put a limit to the max number of heavy hitter, with Theta = 0.2, we can have maximum 5 heavy hitters.

Stream implementation

"Identify all items (*heavy hitters*) that occur more than θN times in a stream S of N items" (θ in [0,1])

– Summary={}
– For each item ● that arrives:
  • If (●, count) is in Summary:
    update count to count + 1
  • Else:
    add (●, 1) to Summary
  • If |Summary| ≥ 1/θ :
    decrease the count of all pairs in Summary
    remove all pairs with count = 0

Algorithm by *Karp et al.*
• Problem:
  – Find all items exceeding frequency θN
• Space:
  – O(1 / θ) counters
  – Counters are log(N) bits each => O(1/ θ log(N)) space
• Time per update: O(1) *
• **Concession:**
  – **False positives or 2-pass**

Karp et al. propose a data structure that allows O(1) in worst case

Here, we are in a situation of a stream, it is the same as the document/set of solution, for the if summary >= 1/theta, we decrease by one each element of the counter in the dictionary and when one pair = 0, we remove it, at the end, we will have the same result as in the previous example, a set of potential heavy hitters.

Lossy Counting of Frequencies

Lossy counting of frequencies is a better algorithm, we want to have the frequencies, we can be OK with false positive as long as we do not have very bad false positive. It is the difference with Karp Heavy Hitters

We have a notion of time, time when the item appears.
The For loop is checking the frequency of appearance of one item regarding our current history.
The frequency will be **relative** to the history, the first item will have a frequency of 100% but after 3 runs, it could have a frequency of only 33% if it does not appeared during the run 2 and 3.

Example: (20%)

	start	# (freq)
●	1	1 (17%)
●	2	1 (20%)
●	3	2 (50%)
○	4	1 (33%)
●	6	1 (100%)

And if it is less than Theta, we remove it. We consider that if it is locally infrequent, it will be globally infrequent so we remove. On the opposite side, if we find that it is greater than Theta, we consider it frequent locally and then globally so we keep it !

- Following algorithm: finds *superset* of θ-frequent items:
  - Initialization: none of the items has a counter
  - Item ● enters at time t:
    - If ● has a counter: count(●) ++
    - Else:
      - count(●) = 1
      - start(●) = t
    - For all other items △ with a counter do:
      - If count(△) / (t – start(△) + 1) < θ :
        - » Delete Counter for △
- Query time: return all items that have a counter

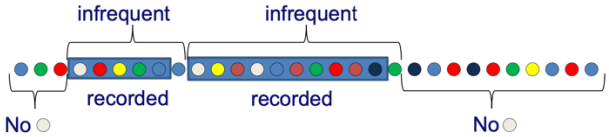
In the end, we return only everything that has a counter !

Why does it work?

- If ○ is not recorded, ○ is not frequent in the stream

Imagine marking when ○ was recorded:

- If ○ occurs, recording starts
- Only stopped if ○ becomes infrequent since start recording



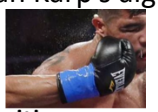
Whole stream can be partitioned into parts in which ○ is not frequent ➔ ○ is not frequent in the whole stream

We can divide the space into frequent and infrequent parts for one item and therefore observe that one item is infrequent locally and therefore globally and we are right to remove it.

In terms of memory, an item is in the summary if it appears 1 among the last k times or 2 among the last 2k times or it appears x times among the last xk times, etc. Worst case is when each item appears only once therefore, we need to keep all of them !

We must notice that our false positive heavy hitters are not that bad because there are in fact frequent but less than required (Theta – **epsilon**). → better than Karp's algorithm.

Karp's algorithm:



- O(1/θ) counters
- No false negatives, but may have false positives

Lossy Counting:

- 1/ε log(Nε) space ***worst case*** *(usually much better!)*
- Maximum error of ε on counts
- No false negatives, only false positives in the range [θ - ε, θ]

- There exist many other algorithms (e.g., CM-sketch)

In the worst case (rare), Lossy counting takes more space, memory than the Karp's algorithm

Example : useful to block malicious IP-addresses.

Heavy hitters can speed-up things while scarifying a little bit of accuracy !

Filtering

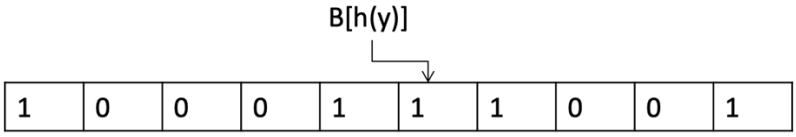
We want to find in a stream, elements that are important.

- Suppose we are not interested in all stream elements, but only in a subset
  - Telecom network: not all calls, but only calls of a group of users having a specific tariff plan
  - Internet packages: traffic to certain suspicious websites
  - Only allow mails to a large whitelist of email addresses
- Stream is possibly captured in a distributed way and selection may be periodically changing
  - Think about our blacklisted pairs of IP addresses

We have a stream of items X, functions key(.) that give the identity of the items and the target set of keys A. And we want to check if X has a key(x) that belongs to A ! (Target could be a spam for e-mails, we want to filter this).

How to do this ? How to identify if an item X has a key(X) that belongs to A ? The idea is to have a vector of bits with 1 and 0 and if it is one we filter it, if 0, else. The bit set will be as long as the domain.
(Sometimes, we cannot afford this because it is too much memory) → We use a hash function to move from the big domain to the small domain of m.

We want to check a new item y, if the bit set (B) at the position h of y is at this position 1 or not ?

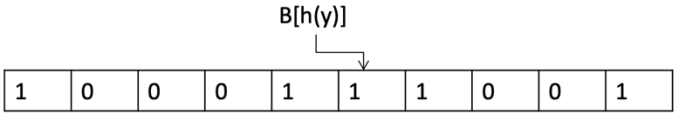


If according to the hash function, we output at 0, we are not part of set A, if we output (based on hash function) at 1, we are part of the set A.

We may have false positive. We can compute the likelihood to get 1.

- No false negatives
  - If y is in A, the bit h(y) will be 1
- There may be false positives, though:
  - if $\boldsymbol{x \in A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the false positive probability we reason as follows.
Let m = length of bitstring, n = number of elements in A



Probability that a single element a in A leaves B[h(y)] untouched

$$\frac{m-1}{m}$$

The probability to not be at the right place is (m-1)/m (with the hypothesis that the hash function is well balanced). Because we have m elements on the bitstring. One element of A has a probability of (m-1)/m to not go to the right place.

Probability that every element a in A leaves B[h(y)] untouched

$$\left(\frac{m-1}{m}\right)^n$$

Probability to get 1 :

Probability that some element in A sets B[h(y)] to 1:

$$1 - \left(\frac{m-1}{m}\right)^n$$

Probability to get false positive

Probability that some element in A sets B[h(y)] to 1:

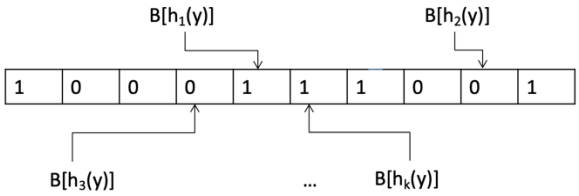
$$P[FP] = 1 - \left(\frac{m-1}{m}\right)^n = 1 - \left(1 - \frac{1}{m}\right)^{m\frac{n}{m}} \approx 1 - e^{-\frac{n}{m}}$$

How to reduce this probability to get false positive ?
We can control m and how big it is ! We can fix a level of acceptable probability to get a false positive by defining m as :

Hence, if we want to reduce the probability of a false positive to $\boldsymbol{\varepsilon}$, we will need m= $-\frac{\boldsymbol{n}}{\boldsymbol{ln(1-\varepsilon)}}$ bits in our bitstring .

Bloom filter is an improvement of the current method, trying to improve the performance of the technic.
We can have collusion/collisions when we move to a big domain to a small domain m. To tackle this issue, we can do multiple times the experiment, with different hash functions ! (because the randomness comes from the hash function.

– Using multiple (k) hash functions; an element $\mathbf{a} \in \boldsymbol{A}$ will set all bits B[h_j(a)]



Membership test for $\boldsymbol{y} \in \boldsymbol{A}$:
Only return true if *all* bits are 1.

We will use k hash function from the initial set A, we set not 1 bit but k bits. Each hash function gives 1 bit. We have 4 hash function and we get different positions to 1. We want all our k hash functions (4 here) to get to 1 and if not all of them get 1, we say that we are not "there", "we are not 1".

→ we repeat the experiment k times.

For k=1 we had: $\boldsymbol{P[FP|k=1] \approx 1 - e}$
We need 1 in every of the k positions:

$$\boldsymbol{P[FP] \approx \left(1 - e^{-\frac{kn}{m}}\right)^k}$$

Assumes independent probabilities and k different hash values (realistic if k<<m)

(k must be much smaller than m !)

$$\boldsymbol{m = \frac{-n\,ln(\varepsilon)}{ln(2)^2}} \textbf{ and } \mathbf{k} = -\frac{\boldsymbol{ln(\varepsilon)}}{\boldsymbol{ln(2)}} = \log_2\left(\frac{\mathbf{1}}{\boldsymbol{\varepsilon}}\right)$$

We have optimal value for m and for k now ! → with 12 bits, we can have almost 100% of true positive !

Bloom filter can be used to check if a key exists in storage or not ! (to avoid useless disk look-up and have a warranty that something exists)

Extreme Counting

The problem : finding the number of unique items in a stream (for example, unique visitors in a website)

Neither heavy hitters nor filters are suited for counting
the number of unique items in a stream

S being a set that store the people, we have a counter N and if P is not seen, we increment N and put in S.
We can apply a hash function, every person is hashed to some value and the value is stored. It works as long as the hash function is perfect, (no collision)

S={}
N=0
Whenever a person P enters the room:
    if h(P) **not in** S:
        S = S ∪ { h(P) }
        N+=1
h(P) denotes hash of identifier of P

How many people attend my presentation?

Near exact algorithm if range(h) large enough
Complexity:
    Space O(N log(N))
    Time per person entry O(log(N))

The first solution is not satisfactory, we will explore a simplified version of hyperloglog sketch which is Flajolet-Martin sketche !

This solution is not satisfactory at all:

- Space N log(N) is completely unacceptable
- Time log(N) per entry is barely acceptable

We will introduce an alternative: Hyperloglog sketch

- Space log(log(N))
- Constant update time
- But approximate

HLL is based on the idea of Flajolet-Martin sketches

Flajolet-Martin sketch (counting)

It does not depend on the stream, only on the number of elements !

The idea is that if we see a large number of people, we are more likely to see an even larger number of people. (N/(N+1)) while is we only have one person, we are less likely to see a second person !

- Compute the number for everyone entering the room
- Maintain the maximum over all numbers seen, call this S_{53}

What do we know about this largest number S?

It only depends on the *number* of elements, not on how many times they entered:

$\max\{h(P_1),h(P_1),h(P_1),h(P_2)\} = \max\{h(P_1), h(P_2)\}$

The higher the number of elements N, the higher S will be *in expectation*

$$P(S \leq x) = x^N$$

$$E[S] = \int_0^1 x\, N\, x^{N-1}\, dx = \frac{N}{N+1}$$

We can reverse-engineer:

$$N = \frac{E[S]}{1-E[S]}$$

To have a good estimator E[S], we can repeat the experiment k times to have a smaller variance (statistical trick).

There are still a number of issues, though:

First issue: S can be far away from E[S]

- Accidentally having one person with a high hash number may lead to huge overestimations
- Use multiple hash functions instead
  - k independent hash functions $h_1, \ldots, h_k$ give $S_1, \ldots, S_k$

$$Var\left(\frac{\sum_{i=1}^{k} S_i}{k}\right) = \frac{Var(S)}{k}$$

Similarity search

We want to find items that are similar to each other (without a 1 to 1 comparison)

Locality sensitive hashing is the technic we will use. It is based on the Jaccard Coefficient.

Indicates how similar the sets A and B are.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Example:

J({a,b,c},{c,d}) = 1/4

J({a,b,c},{b,c,d}) = 2/4

For text documents, we check the n-grams, n-gram is a sequence of n words. We compare sequence of words between two documents to check similarity.

It is relevant for product recommendation for a user. → We therefore compute the jaccard between each item and all the items of the DB. The complexity is therefore the cardinality of the DB. We can set a threshold of similarity and for each jaccard index > threshold, we can return the item/product that are good candidates.

Let A, B be subsets of universal set V
h is a function mapping elements of V to {1,2,…,|V|}
Example: d → 1, c→ 2, a → 3, b → 4

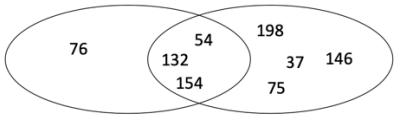
Let $\min_h(A) := \min_{a \in A} h(a)$

$Pr[\min_h(A) = \min_h(B)]$

$= |A \cap B| / |A \cup B|$

$= J(A,B)$

* We implicitly assume range(h)>>|A|, |B|



With a hash function, we convert items to unique values (d → 1). Each set will have a minimum "$\min_h$ (min hash)", so what is the probability to two sets to have the minimum min hash ? It is the probability that they agree on the min hash, on the element. It is the jaccard index/coefficient (in our situation 3/8). → it is the probability over the different hash functions.

If we have m hash functions, we have m hash values. And the new check for how many hash functions, we have the same min hash and we count them. (And we can repeat the experiment to have a lower variance, we could have 4 different hash functions to give 4 different hash values to each item).

Independent functions $h_1, \ldots, h_m$

"signature" of set A:

|A| and vector ($\min_{h1}(A), \min_{h2}(A), \ldots, \min_{hm}(A)$)

Estimating J(A,B):

$(a_1, \ldots, a_m)$ vector for A $(b_1, \ldots, b_m)$ vector for B

Let $e = \#\{ i \mid a_i = b_i \}$

e / m is an estimator for J(A,B)

For example we have set A with for hash functions, a result from each function, it is the signature. We do the same for B and C and we compare the similarity between the min value of each hash function between each set to compute the jaccard coefficient ! It gives us an estimation (estimation because we use a hash function)

Example: V = { a, b, c, d, e }

A = { a, b }
B = { b, c, d }
C = { a, b, c, e }

A	1	2	2	2
B	2	1	1	3
C	1	1	2	1

	h_1	h_2	h_3	h_4
a	1	2	5	2
b	2	5	2	4
c	3	1	4	5
d	4	4	1	3
e	5	3	3	1

J(A,B) = 1/4 ; estimate: 0
J(A,C) = 1/2 ; estimate: 1/2
J(B,C) = 2/5 ; estimate: 1/4

For AB jaccard index, the reality is ¼ (because b is matching) but with hashing function, we get 0 (we are doing a mistake). (A being a set of items that is processed through hash functions to get a numerical value)

With this, we are building the **Locality-sensitivity hashing.**

Every set of items item, set is a column (we transposed compared to jaccard coefficient & signature. We can split, it in bands, the matrix of sets. The bands can be hashed into bucket and we do this for every band. If two columns falls into different buckets, they are different. If they go into the same bucket, they are probably similar and we can consider them as candidate pairs! Same bucket → identical.

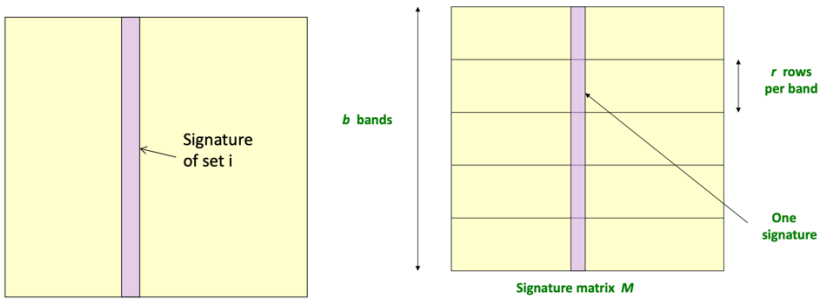
We will first illustrate the principle for Jaccard Index

MinHashing to create signatures of sets

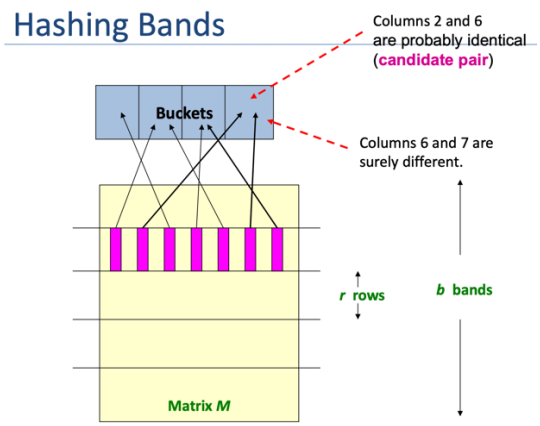
A → [123, 235, 576, 67, 56]
B → [123, 3456, 56, 67, 867]

J(A,B) estimated by number of entries on which their signature corresponds

Signature matrix



- Candidate column pairs are those that hash to the same bucket for ≥ 1 band
- Tune b and r to catch most similar pairs, but few non-similar pairs



With the example, we have for 80% similarity, we have a probability of 32,8% to be identical in one band (for 2 sets of items) and a probability of not being similar in all of the 20 bands of 0.035% → low proba to have false negative.

BUT for a similarity of 30%, we have a probability to be identical in one particular band of 0.243% while we have a probability to be identical in at least 1/20 bands of 4.74% which is a high level of false positive ! We must remove items with a similarity which is low.

- **Pick:**
  - The number of Min-Hashes (rows of ***M***)
  - The number of bands ***b***, and
  - The number of rows ***r*** per band

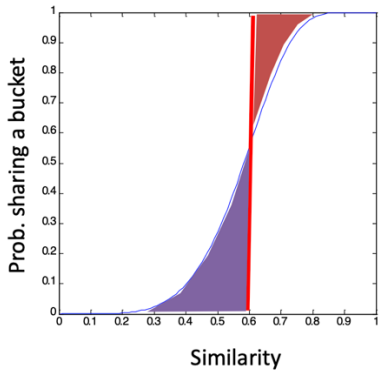
  to balance false positives/negatives

- **Example:** If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

Low similarity threshold → issue with false positive, high similarity threshold → issue with false negative.

S-curve – picking the right r and b values. The value will depends on the application with have.

- **Picking *r* and *b* to get the best S-curve**
  - 50 hash-functions (r=5, b=10)



Red area: False Negative rate
Blue area: False Positive rate

Behind LSH, we use min hash technic.

- LSH is an indexing technique for similarity search particularly useful for high-dimensional data
- Can be extended to other similarity/distance measures
- Key ingredient: there must exist a hash-functions h such that P[h(x)=h(y)] increases with sim(x,y). These hash-functions can be combined to reach the needed sensitivity

Among all the technics saw in this chapter (heavy hitter, filtering, extreme counting or similarity search), if we want to do exact answer for those computation, it takes a lot of time and resources. We solve this problem by sacrificing the exactness (FP, FN depending on the case) but we are able to control this.

5. NoSQL Databases

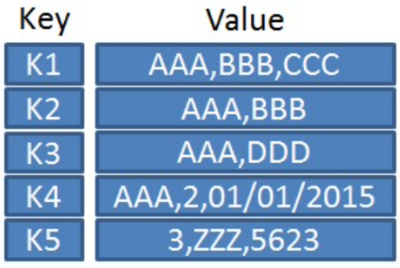
Overview of NoSQL

RDBMS

Pros	Cons
Simple and intuitive model	Impedance mismatch (offer tables that needed to be joined to get all the information but they want integrated views)
Convenient for real-life data	Not built for distributed data management (hard to deploy cluster) (large scale clusters)
Elegant mathematical foundation	Single point of failure
Allows efficient algorithms	Performance (not tailored for specialized applications)
ACID	Scaling: - Can scale up (grow the power/capacity of a machine) - Cannot scale out (adding new machines)
Industrial strength implementations are available	

There are 4 types essentially :

- Key-value stores
    - Simple data model (key-value pair, everything is related to a unique key

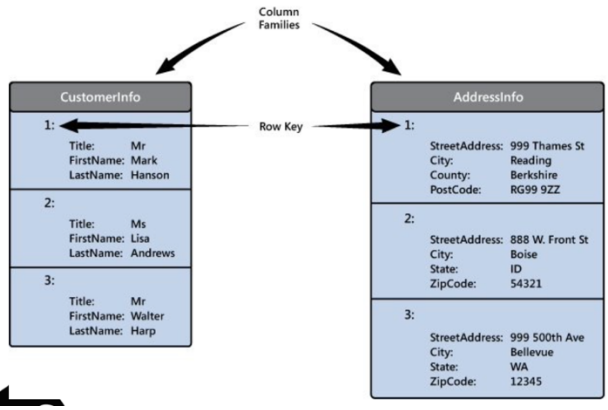


- Documents stores
    - Document oriented data model (store content of a document or even a simple json file)

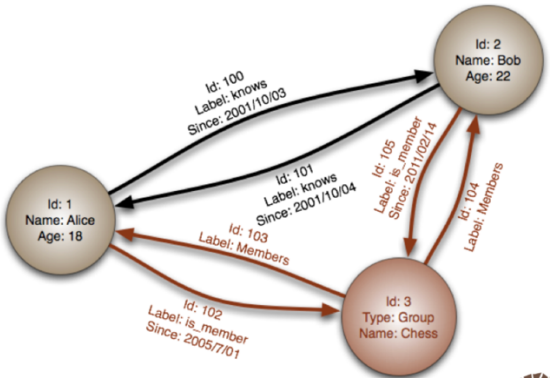
```
{"id" : 1001,
  "customer_id" : 7231,
  "items" : [{"product_id" : 4555, "quantity" : 10},
             {"product_id" : 1213, "quantity" : 1}]
}

{"id" : 1002,
  "customer_id" : 231,
  "items" : [{"product_id" : 55, "quantity" : 34}]
}
```

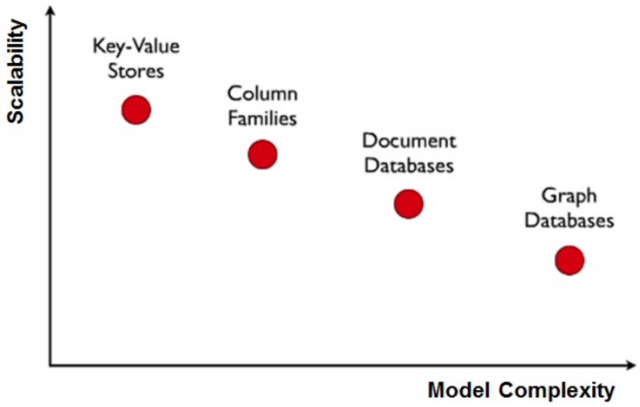
- Column families
  - Uses tables, rows and columns (almost like relational database)
  - BUT names and format can vary from row to row on the same table (country >< state, zip code >< postcode)
  - Each row between column is linked to another with a unique key !



- Graph databases



Scalability and complexity tradeoff



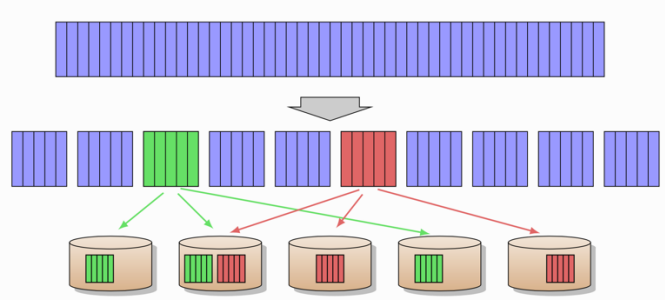
Main RDBMS problem is that it could not scale beyond to 10s machines ! The problem was not so much read operations, but large workloads of many small updates Consistency - Availability – Partition.

Many solutions were offered by Google, Amazon and Facebook !

Consistency – availability – partition

If we want high availability → it implies **replication** through many disks

If we want consistency (everyone should see the same thing) → it implies **update propagation** through all the disks where it is saved



(distribution and replication)

ACID transaction (standard for transaction update)

The golden standard for transaction management in Relational DBMSs. • Recall what ACID stands for:

- Atomicity: Either all effects of the transaction are recorded in the database, or none are.
- Consistency: After the transaction, the database must satisfy all validity rules.
- Isolation: A transaction must have the impression it is the only one running, in terms of values it has read and the resulting state of the database. (like there were not distribution)
    - This is also referred to as serializability, i.e., the answers to queries and the final state of the database must be as if the transactions were serially executed.
- Durability: Once a transaction has been committed, it stays committed.

How to achieve this for large dataset ?

- ACID properties are always desirable (it is a must but not a necessity)
- But, web applications have different needs from applications that RDBMS were designed for :
    - Low and predictable response time (latency)
    - Scalability & elasticity (at low cost!)
    - High availability (always respond to user request)
    - Flexible schemas and semi-structured data Geographic distribution (multiple data centers)
- Web applications can (usually) do without

- Transactions, strong consistency, integrity
- Complex queries

CAP theorem

Consistency

- The system is in a consistent state after an operation
- All clients (asking for information) see the same data
- Strong consistency (ACID) vs eventual consistency (BASE) (relaxed version of consistency)

Availability

- The system is "always on", no down time (and respond to queries)
- It can tolerate Node failure– all clients can find some available replica, it should provide information to client requests
- Software/hardware upgrade tolerance

Partition tolerance

- The system continues to function even when split into disconnected subsets, e.g., due to network errors or addition/removal of nodes (when the cluster is made of disconnected components (due to errors connections or geographical disconnection) it should continue to work)
- Not only for reads, but writes as well!

→ CAP theorem : in a shared data system, at most 2 out of 3 of those properties can be achieved at any given moment in time.

Talking about distributed system, we WANT a partition tolerance system (so it is like an obligation for us and then we can pick one of the two others and after try to optimize the remaining one)

CA

- Single site clusters (easier to ensure all nodes are always in contact) e.g., 2PC
- When a partition (failure ?) occurs, the system blocks

→ we are not really distributed system, we have a single cluster.

CP

- Some data may be inaccessible (availability sacrificed), but the rest is still consistent/accurate
- E.g., sharded database

→ Some data can be not accessible but what is accessible is accurate

AP

- System is still available under partitioning, but some of the data returned may be inaccurate
- I.e., availability and partition tolerance are more important than strict consistency
- E.g., DNS, caches, Master/Slave replication

- Need some conflict resolution strategy

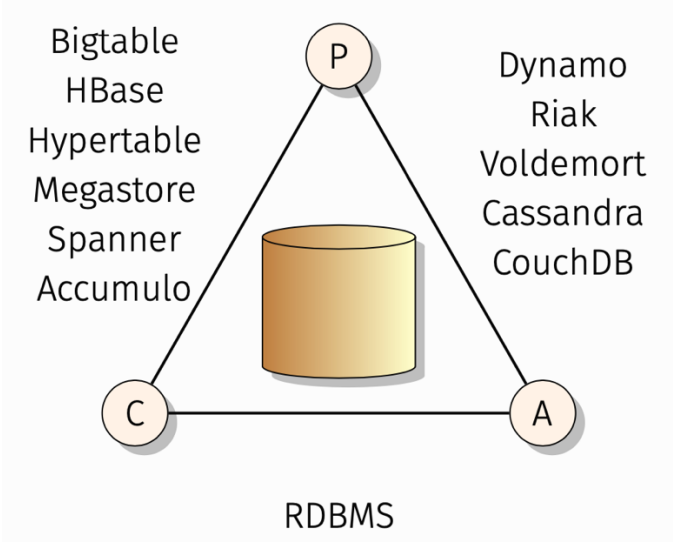
→ Very popular, we want to be able to respond to queries and we do not care about strict consistency (eventual consistency)

Eventual consistency :

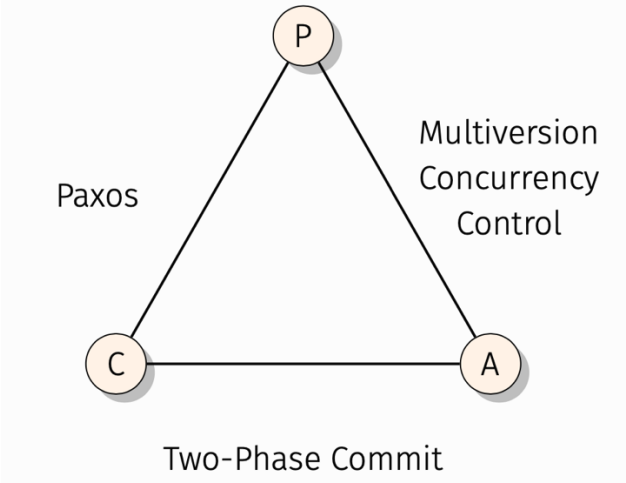
If there are no more updates happening, there is a way that we have replicas that converge into a consistence state, identical copies , but in the meantime, we can have state that is not exactly consistent and the developer will have to fix the issue to reach consistency.

The meaning:

- In absence of updates, all replicas converge towards identical copies.
- What the application sees in the meantime is sensitive to replication mechanics and difficult to predict.
- Contrast with RDBMS: Immediate (or "strong") consistency, but there may be deadlocks.



Consistency protocols



The two-phase commit protocol (2PC)

We tolerate that we are not partition tolerant

Designed for RDBMS when it is distributed

- **Phase 1:**
  - Coordinator sends Prepare to Commit
  - Subordinates make sure they can do so, no matter what
    - and write the decision to the local log to tolerate failure after this point
  - Subordinates reply Ready to Commit or Not able to Commit

If everyone say "ready to commit" we can go, instead, we block the commit but end in a consistent state.

Phase 2:
- If all subordinates have sent Ready to Commit,
  - then the coordinator sends Commit to all
  - else it sends Abort to all
- In both cases the coordinator writes the decision to the local log.
- If a subordinate receives Commit or Abort, then it
  - writes the received message to the local log and
  - does the update, if it received Commit, and releases the resources

(if not ready to commit) 2PC can freeze parts of your database for a long time if the Coordinator goes down for a while after "ready to commit". (can take times)

Multiversion Currency Control (MVCC) (tolerate partition failure)

We tolerate data that are a little bit outdated

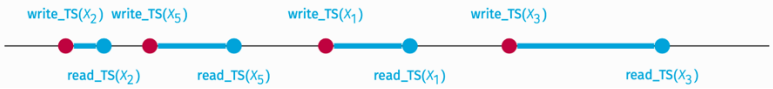
To maintain version of the data and be consistent with these versions. We can respond to a outdated version of the data (but consistent !). It increases the availability.

Because we have versions of the data, we need to keep all the versions of the data → more storage (at one time, we can remove the garbage versions of the data)

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction.
- Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effect:
  - Significantly more storage (RAM and disk) is required to maintain multiple versions.
  - To check unlimited growth of versions, a garbage collection is run when certain criteria are satisfied.

Write rule

- Every transaction T is assumed to have a timestamp $\mathrm{TS}(T)$
  - We will pretend this is when the transaction was **atomically** executed.
- Assume $X_1, X_2, \ldots, X_n$ are the versions of a data item X created by a write operations of transactions.
- With each X_i we associate **two timestamps**:
  - $\mathrm{read\_TS}(X_i)$: the largest of the timestamps of transactions that have successfully read version X_i.
  - $\mathrm{write\_TS}(X_i)$: the timestamp of the transaction that wrote the value of version X_i.
- A new version of X_i is created only by a write operation.

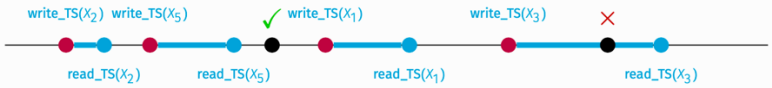


To ensure isolation / serialization, avoid overlapping in the writing, we have rules.
Why it is bad to overlap, read a version replaced by another version ?
→ We must identify which version has been red, to which version it is refer ! The read at one point in time, should be related to one exact version/moment in time.

We should not change a version of the data if someone already red this ! Some of the write can be rejected !

To ensure serializability, the following rule for **writing** is used:

- Assume transaction T attempts to write X.
- Let X_i be the version of X with the highest $\mathrm{write\_TS}(X_i)$ that is also less than or equal to $\mathrm{TS}(T)$.
- If $\mathrm{read\_TS}(X_i) > \mathrm{TS}(T)$ then **reject**
- Else, create a new version X_j and $\mathrm{read\_TS}(X_j) = \mathrm{write\_TS}(X_j) = \mathrm{TS}(T)$.



We can write the at the black point because no one already read it BUT the cannot write at the black point because someone already read it. (read validity duration)

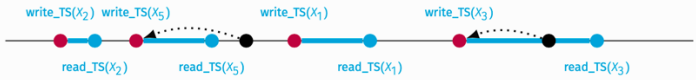
Read rule

We should be assign version to read.

To ensure serializability, the following rule for **reading** is used:

- Assume transaction T attempts to read X.
- Let X_i be the version of X with the highest **write_TS**(X_i) that is also less than or equal to **TS**(T).
- Return the value of X_i to T.
- Set **read_TS**(X_i) to the largest of **TS**(T) and **read_TS**(X_i).

Note: This rule **never rejects** a read operation.



With the black point on left, we need to extend the read duration. With the black point on the right, we do not need to change anything.

→ We sacrifice consistency, we have multiple versions, no single truth in order to ensure availability

The Paxos Protocols

→ Sacrifice availability for consistency.

Idea : we want to have a majority of nodes agreeing on specific value. (can come and leave (due to network failures)

Participants are the nodes and they can play many roles at the same time :

- Proposers : it initiate some updates (proposer some new value)
- Acceptor (everybody else) : analyse the value proposed by the proposer and come back with a response (accept or reject)
- Learner : when a decision is made, it should learn the decided value (not very relevant)

Two phases :

Phase 1 : the proposer send a prepared message tio the accepter and it can either accept the message or ignore the message.
Each message as a generation number (a counter) (every participant has a local counter) and it increases with the time.

- Phase 1a, Prepare: A Proposer sends a Prepare message with a proposal number n to a majority of the Acceptors.
    - The number n must increase with each round.
- Phase 1b, Promise: The Acceptor checks if n is higher than any proposal number received from any Proposer. If so, the Acceptor returns a Promise message with m and u, where m and u are the proposal number and value of the highest-numbered proposal accepted by it so far.
    - This implies a promise to ignore any proposal with a number < n

- Phase 2a, Accept Request: The Proposer waits until it has received promises from a majority of the Acceptors, and then checks if any of them already accepted a value earlier. If so, then v is set to the u of the Promise message with the highest m. Then, an Accept request message with n and v is sent to the Acceptors that responded.

- Phase 2b, Accept: If an Acceptor receives an Accept Request message with n and v, it checks if it promised to ignore all proposals with proposal number n. If not, it accepts the proposal and sends an Accept message to the Proposer and all the Learners.

Conclude or Retry: If a Learner gets Accept messages from a majority of the Acceptors, a decision has been reached. If a Proposer notices that this is not happening, it may initiate a new round with higher n.

Key-Value Stores

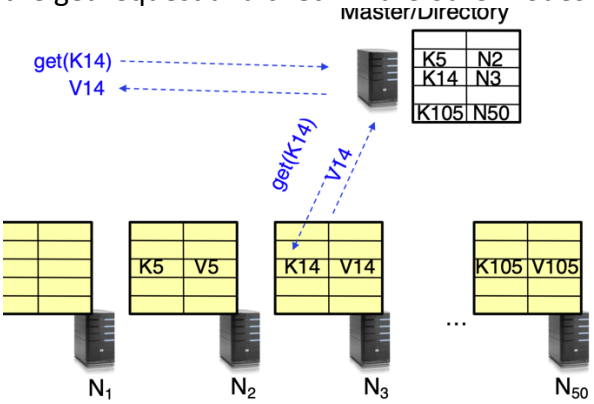
Everything looks like key pair/values. Easy, straight forward.

How to distribute key-value around the machines ?
Where to put, store a new key ?
Where to get a new key ?
How to do this to ensure scalability, consistency and fault tolerance ?

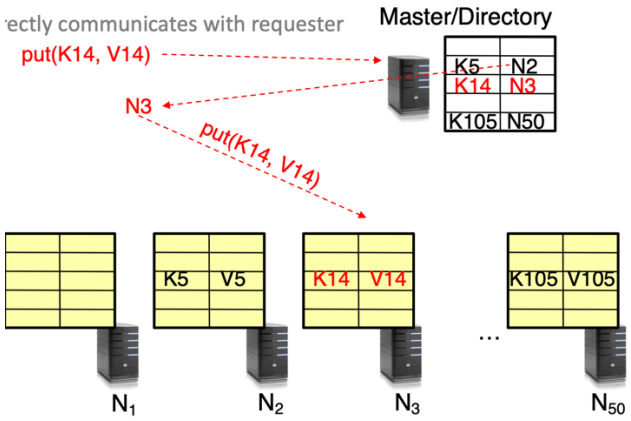
Directory based architecture

Master/directory node keeps the directories, it is a mapping between keys and files.

→ everything goes through the master → **recursive access**
The master will receive the get request and check in the other nodes



Iterative access :
The Master is only here for a look-up to check in which note is the value and it is the machine from the request than will check within the node. (direct communication)

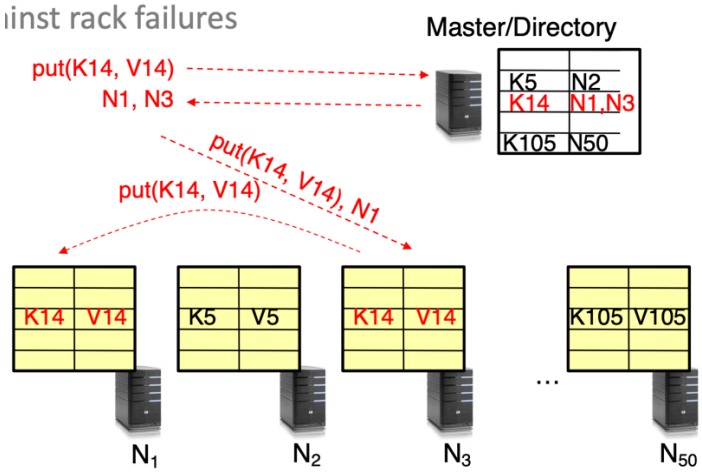


Recursive access	Iterative access
(+) faster, typically master is closer to the nodes	(+) more scalable
(+) easier to maintain consistency as the master/directory serialize put / get (everything goes via a single point)	
(-) scalability bottleneck as all values go through master node	(-) slower, harder to enforce data consistency

How to handle fault tolerance ?

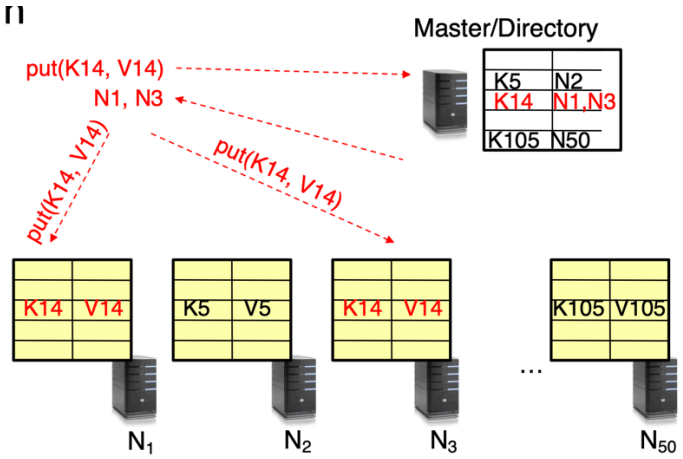
→ We keep replicas of the values, in potentially different racks or data centers.

Recursive



We have two nodes here in the directory and the put, in order to propagate the value !

Iterative



We can have recursive access but the replication is iterative

How to handle scalability ?

→ We use more nodes (storage)
→ Can serve requests from all nodes on which a value is stored in parallel. Master can replicate a popular value on more nodes (in terms of number of requests)
→ Replicate directory (ok idea), Partition directory, so different keys are served by different masters/directories (better idea) (master/directory scalability)

How to handle consistency (how values replicates correctly)

Atomic consistency (ACID)
Eventual consistency

How to scale up the directory ?

Directory contains a number of entries equal to number of (key, value) tuples in the system.
→ If we distribute the data, we must also distribute the directory.

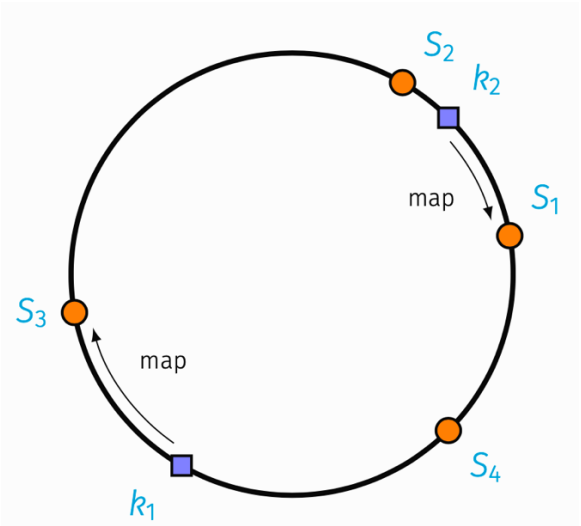
It can be done via **consistent hashing !**

One way to partition key value pairs over the servers is to use a hash function; if we have n servers, we get the modulo of the key with n and we pick the modulo as the server ! BUT if N changes (the number of servers), we need to repartition all the pairs !

Solution : consistent hashing

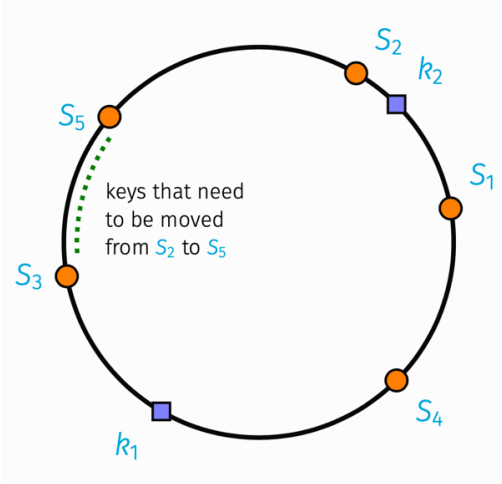
We have a fixed hash space.

- a fixed hash function h maps both the keys and the servers IPs to a large address space A (e.g., $[0, 2^{64} - 1]$);
- A is organized as a ring, scanned in clockwise order;
- if server S is followed directly by S' on the ring, then all the keys in range $]h(S), h(S')]$ are mapped to S'.

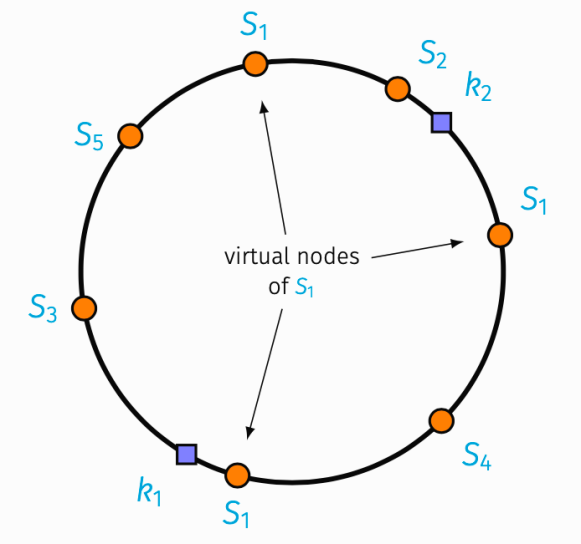


The ring is our ensemble of servers, 4 servers S in total. Any key k between S2 and S1 is mapped to server S1.

When we had a new server, we need to move some keys. Some keys will move from S2 to S5



If somebody leaves, the keys will be taken over by the previous server on the ring. And to guaranty that nothing is lost, we repeat this process to replicate thing with different hash functions. The keys from one server can be found on another ring probably.



Server failure : With the image upward, we can have a replication factor of 3, with virtual nodes of S1 which are making the copy of some parts of the ring to deal with potential server failures.

Balance the load map of a server by creating virtual nodes on the ring.

Until now, we do not have a master node to find where a key is store ?

- On a specific ("Master") node, acting as a load balancer. Example: caching systems. ⇒ raises scalability issues. (we store the all ring, not scalable ! to avoid !)
- Each node records its successor on the ring. ⇒ may require O(N) messages for routing queries – not resilient to failures. (only store the next one on the right "I do not know, go to the next one on the right, etc, implies a loooot of messages to search on the ring)
- Each node records position of 1st, 2nd, 4th, 8th, 16th, .. follower in ring. ⇒ ensures O(log N) messages for routing queries – convenient trade-off for highly dynamic networks (e.g., P2P) (record the 1^{st}, 2^{nd}, 4^{th}, etc. neighbors on the ring to find in which interval the key is stored on the ring, very nice approach, requires only O(log(N)) messages)
- Full duplication of the hash directory at each node. ⇒ ensures 1 message for routing – heavy maintenance protocol which can be achieved through gossiping (broadcast of any event affecting the network topology).

NoSQL Rebuttal

To handle modern requirements of web applications, we must sacrifice some things : the relational idea, that we have a fixed schema and all data must be organized and standardized into different tables BUT NOW more flexibly schema. We sacrifice complex ways likes joins for example.

NoSQL solves performance and flexibility issues → explain popularity (even more for applications).

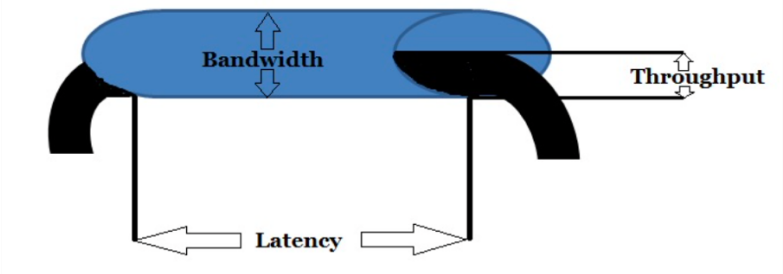
No high level query languages

No standard (complex to move from one to another)

6. Parallel Processing

It is possible to analyze huge data sets by exploiting data parallelism:

- partition and distribute the data over a cluster consisting of many machines
- machines operate in parallel on part of the data and communicate over a network to compute the final result

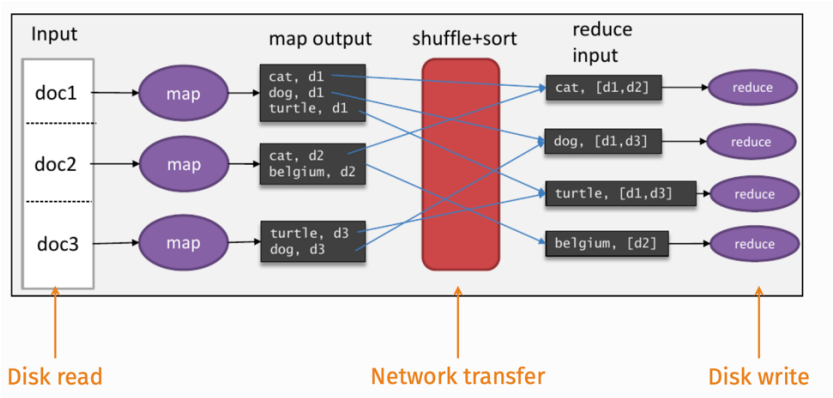


Where is the bottleneck ?

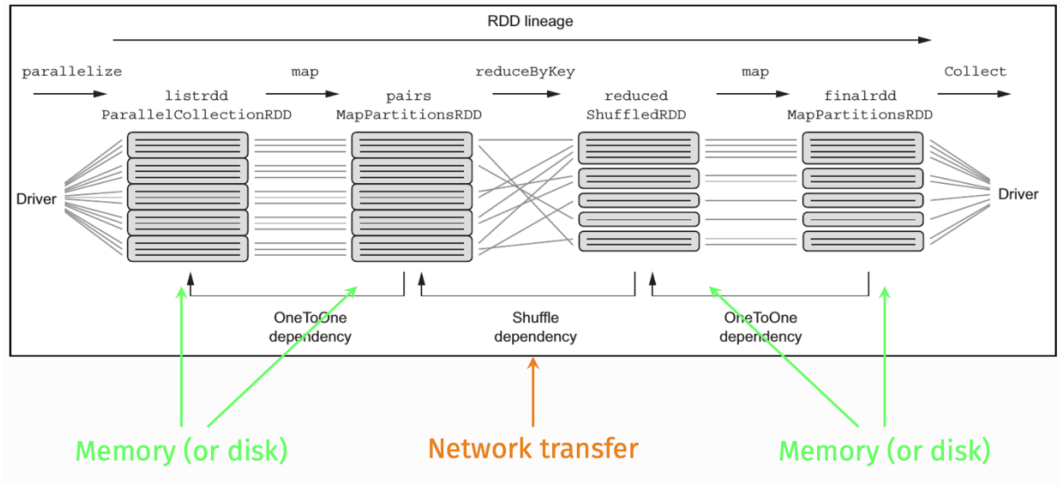
Throughput is the total amount of work done in a given time.
Example : 100MB copied from disk A to disk B in 10s = 10Mb/s throughput
Latency, also known as response time is the time between the start and completion of an event
Example : If it takes 1h to analyze 1 terabyte completely, the latency is 1h.
Bandwidth is the maximum throughput that we can have.

In terms of speed, Memory >> disk >> network

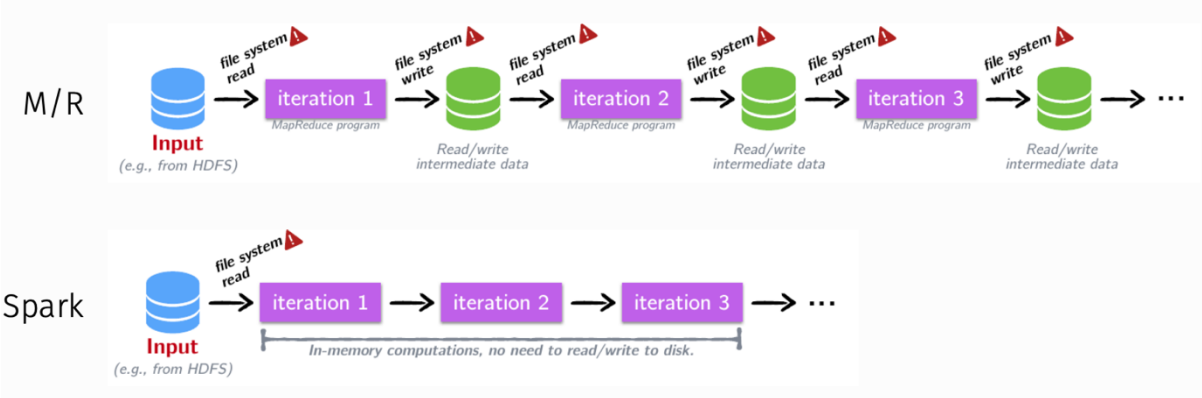
Latency in M/R



Latency in Spark



Comparison Spark & M/R



- Spark is better for iterative algorithms (typical in Machine Learning). Spark eliminates the access to memory and therefore eliminate this source of latency. BUT it cannot avoid shuffling, you still need to move data !
- Try and avoid operations that cause a shuffle in both M/R and Spark. For instance, prefer map-only M/R jobs; be careful how you partition in Spark.

How to improve shuffling in Spark or Map Reduce ? Reduce the effect ? Have a better partitioning strategy !

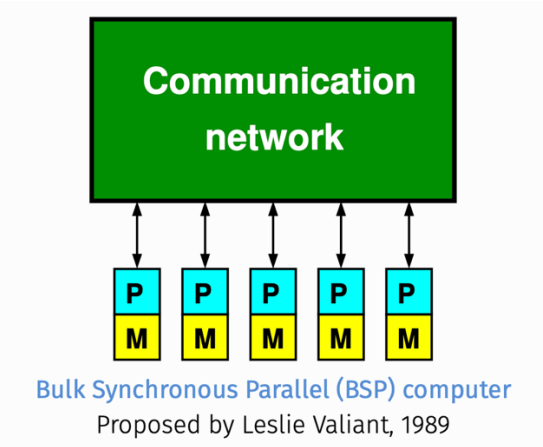
The bulk synchronous parallel (BSP) model

A parallel computer consists of a set of processors (such as a cluster of servers) that work together to solve a computational problem.

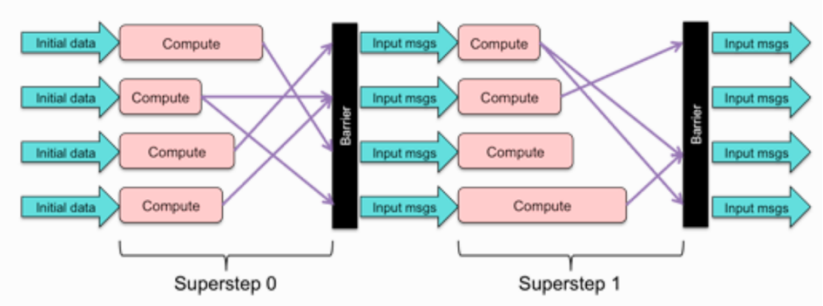
Two types:

- Shared memory parallel computers (e.g., multi-core computer, a super-computer) → the memory is shared → scale-up configuration
- Shared-nothing cluster of machines, a.k.a. distributed-memory parallel computer (e.g., a Big Data compute cluster). → memory is not shared → scale-out configuration → relevant in Big Data.

→ We want to analyze the complexity of a model.



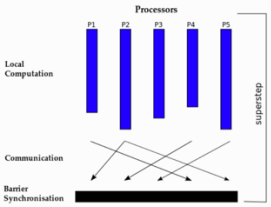
The BSP computer has a collection of processors with each its own memory (share-nothing cluster). It is a distributed memory computer.
The communication network box can be seen as a black box.



BSP algorithm execution consists of a sequence of supersteps, each superstep start with :

- A **computation phase** : (pink boxes) each processor works alone/asynchronously on data in local memory. It is making its own task
- Then we have the **communication phase**, where each processor when it has finished its work, will send data to another processor
- Finally, we have the **synchronization barrier** where our processor is waiting the computation and communication from all the other processors, to put in the right order, synchronization the data. Then, the new superstep can begin.

Now, how to measure the latency, the cost of using an algorithm ?



- Basic arithmetic operations and local memory accesses have unit cost (e.g., 1 time unit).
- Cost C_i of a superstep i:

$$C_i := w_i + h_i \cdot g + l$$

where:

- w_i is the maximum number of local operations performed by a processor in superstep i;
- h_i is the maximum number of data units sent or received by a processor during superstep i;
- g is the throughput ratio (data units/time unit); and
- l is the communication latency[1].

We assume a unit cost associated to each local computation !

- Wi is the maximum amount of work done by a processor for the step I (here it will probably be the value of P2 or P5). → computation phase → computation cost
- Hi is the max amount of data sent during the communication phase → communication cost
- L is some inherent latency to initialize/prepare the communication channel (a constant)

→ All of this is the cost of a superstep ! It is an uperbound (so it never underestimate the cost) (because we take the max of h and w)

Cost C of a computation consisting of S supersteps:

$$\begin{aligned} C &= \sum_{i=1}^{S} (w_i + h_i \cdot g + l) \\ &= (\sum_{i=1}^{S} w_i) + (\sum_{i=1}^{S} h_i) \cdot g + S \cdot l \\ &= W + H \cdot g + S \cdot l \end{aligned}$$

In general, with W,H,S we can describe the cost of an algorithm, cost model.

When designing a BSP algorithm, there is often a trade-off between communication and synchronization. Synchronization, is the number of supersteps that we need.

- Method 1: broadcast the value to all processors.

$$H = O(p) \quad S = O(1)$$

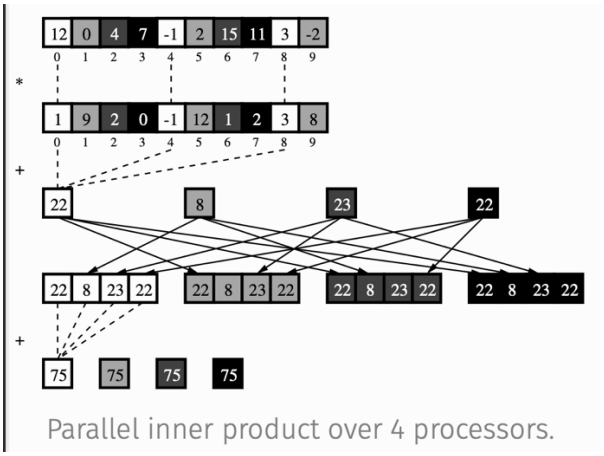
→ Here, we have a high communication cost but only one superstep. → Extremely parallel

- Method 2: organize the p processors in a balanced binary tree.

$$H = S = O(\log p)$$

→ Here we are more balanced → more sequential, do a lot of supersteps !

→ What is the most sequential ? The trade-off doing things in parallel or serialized. Parallel means, we have few supersteps, → more parallel means few supersteps, more sequential means more supersteps.



Parallel inner product over 4 processors.

We do the inner product locally (computation part) (22,8,23,22) (third row)
Then, we do the communication phase by sending the results to all the nodes, all the processors (fourth row). It end a super step ! Now we can do another super step. (for example, everybody aggregate locally the result (fifth row)(75). No need of communication for this superstep.

Cost analysis:

- Superstep 1:
  - Local work to multiply and sum: $w_1 = \left\lceil \frac{n}{p} \right\rceil$
  - Broadcast: $h_1 = p - 1$

Cost analysis:

- Superstep 2:
  - Local work to sum partial results: $w_2 = p - 1$
  - No communication: $h_2 = 0$

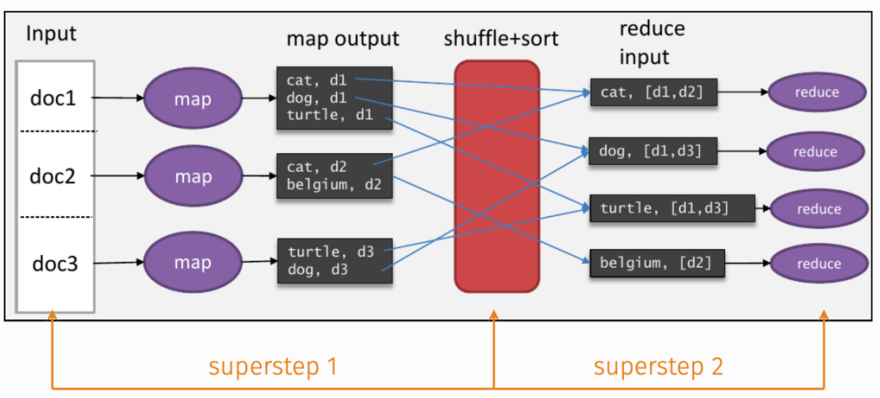
Cost analysis:

- Total cost = $\underbrace{\left(\left\lceil \frac{n}{p} \right\rceil + p - 1\right)}_{W} + \underbrace{(p-1)}_{H} g + \underbrace{2}_{S} l$

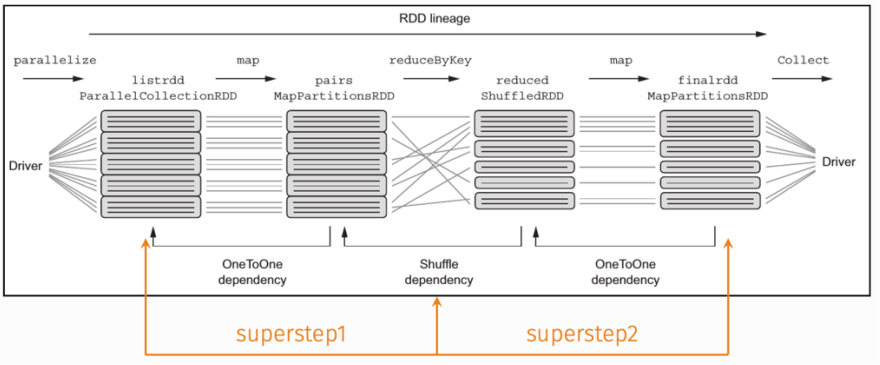
N being the number of color box at row 1, p the number of processors (4)
We are back with W,H,S.

<u>Transpose to spark or M/R :</u>

One M/R job consists of two supersteps: 1 for the map phase; 1 for the reduce phase. Sequences of M/R jobs hence give you a way to implement a BSP algorithm.



→ map reduce is 2 steps



→ reducebykey create a new step

The separation between two supersteps happens when we do the shuffle, the communication between the job done and the processor to synchronize everything.

BSP is a very generic model that can be applied on any case where we want to do parallel computing and it helps us to consider the cost, what would be the effect of an partition, modification for the cost, the complexity.

It allows analysis of parallel algorithms, taking into account the cost of (parallel) local computation; communication; and synchronization.

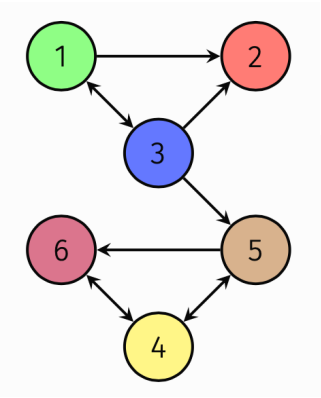
BSP application. : Think like a vertex

We will focus on the PageRank (what made Google famous) for this part.

We need a way to find how good is a node. The idea is that we are at least as good as a node pointing to us. We get the influence from those pointing to us !

It depends on the number of people pointing at us BUT also how good they are themselves.

The page rank for every node in a graph is a score saying that we are as good as the good pointing to me. BUT we do not get all the “influence” from the point pointing to us, we take a fraction based on the number of all the others pointing from the node (if we are 2, we get 1/3 of node 3 and ½ of node 1)



The PageRank p_i of a page i is given by

$$p_i = \sum_{j \in B_i} \frac{p_j}{N_j}$$

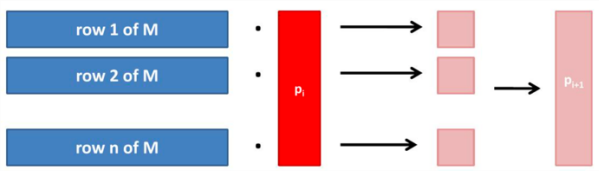
where B_i is the set of pages linking to i and N_j is the number of links on page j.

$$p_1 = \frac{1}{3}p_3 \qquad p_2 = \frac{1}{2}p_1 + \frac{1}{3}p_3$$
$$p_3 = \frac{1}{2}p_1 \qquad p_4 = \frac{1}{2}p_5 + \frac{1}{2}p_6$$
$$p_5 = \frac{1}{3}p_3 + \frac{1}{2}p_4 \qquad p_6 = \frac{1}{2}p_4 + \frac{1}{2}p_5$$

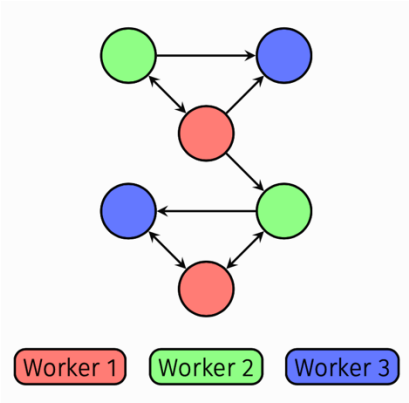
$$\rightarrow \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} \cdot \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{bmatrix}$$

Textbook approach to PageRank in M/R or Spark:

- From the web hyperlink graph one can construct a matrix M that essentially captures the transition probabilities $M_{i,j} = \frac{1}{N_j}$ from node j to node i.[2]
- The pagerank can then be obtained by multiplying an initial PageRank vector by M (power iteration): $\vec{p} = M^k \cdot \vec{p_0}$



Because we have more vertices, nodes than processors (workers), we can allocate many nodes to one worker.

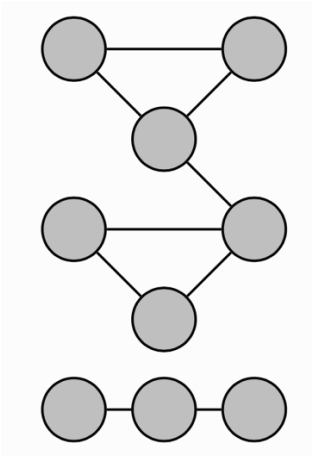


- Assign each vertex to a real processor (worker). E.g. by hash-partitioning, or some more clever form of partitioning.

- In each superstep, the real workers accumulate the messages sent by the vertices to other vertices, and communicate these to the corresponding workers.

Vertex-Centric BSP:

- The majority of graph algorithms are iterative, and traverse the graph in some way.
- Vertex-centric BSP gives a natural way of expressing these algorithms in a parallel fashion by “thinking like a vertex”.



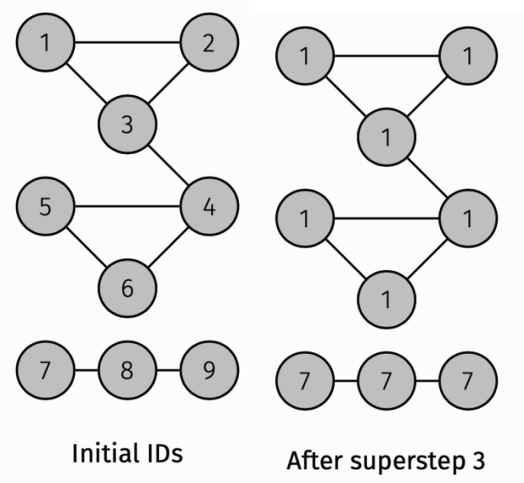
We have some components that are not connected, to detect this, we will firstly assign one unique ID to each node and try to propagate the minimum value through the network. When ID does not changes any more, it means that we can identify the different parts.

The number of supersteps is defined by the length to reach each opposite side of a part

Connected Components in Vertex-BSP.

(For undirected graphs)

1) Initially, each node has a distinct label (id)
2) In each superstep, nodes communicate their label to neighboring nodes, and keep the minimum of their current label, and the labels in received messages.
3) Keep doing new supersteps until no label changes anymore.
4) Upon convergence, each node in a connected component has the same label.



Speedup and scaleup

Speed-up

Ideal example of speed-up :

Let us consider the maximal aggregate bandwidth: the speed by which we can analyze data in parallel assuming ideal data distribution over servers & disks

- Scanning 400TB hence takes 138 secs ≈ 2.3 minutes
- Scanning 400TB sequentially at 100 MB/sec takes ≈ 46.29 days

- Parallelism hence gives a speed-up of 28 800 x
- It is not a coincidence that the number of parallel resources we are using is:

12 hard disks x 80 servers x 30 racks = 28 800

Speedup is the ratio of the latency of two systems, A and B, when run on the same problem (of the same size).
This is **linear speedup**: adding 28800 times the resources gets the job done 28800 times quicker.

Example:

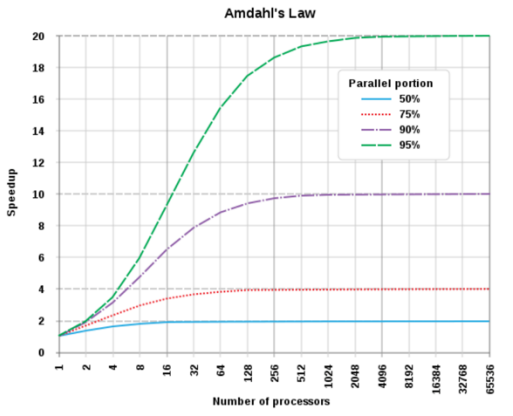
- *A*: Sequential scan of 400 TB with 1 hard drive takes ≈ 46.29 days
- *B*: Parallel scan of 400 TB with 28 800 hard drives takes ≈ 2.3 minutes

$$\text{Speedup of } B \text{ w.r.t. } A = \frac{A}{B} = 28800$$

Bottom line:

- More processors → higher speed.
- Linear speedup ideal (but not always possible). In practice, it is pretty far from that !

Amdahl's law is a formula that gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.



Amdahl's law:

$$\text{Speedup}(P) = \frac{1}{(1-\alpha) + \frac{\alpha}{P}}$$

where

- P denotes the number of parallel processors
- α denotes the fraction of the task that benefits from parallelism;
- $1-\alpha$ denotes the fraction of the task that runs inherently sequential.

Observe:

$$\lim_{P \to \infty} \text{Speedup}(P) = \frac{1}{(1-\alpha)}$$

Some part of the task can be parallelize (alpha part) but another part can just be serialized (1-alpha)

Scale-up

Before, we were comparing the same workload, now we are comparing the same amount of time !

Scalability is the capacity of a system to handle a growing amount of work by adding a growing amount of resources.
Scaleup = is the ratio between the amount of work that two systems, A and B, process when running for the same amount of time.

Example:

- *A*: in 10 minutes, 10 servers process 100 TB of data.
- *B*: in 10 minutes, 22 servers process 200 TB of data.

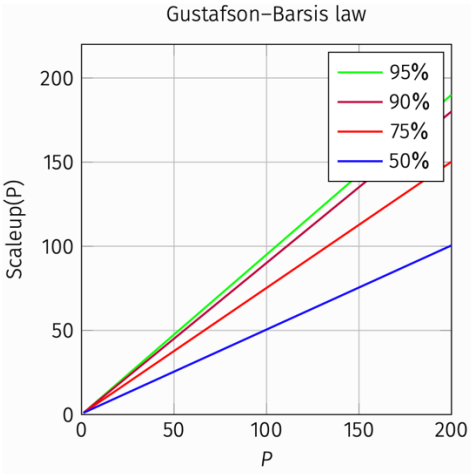
$$\text{Scaleup of } B \text{ w.r.t. } A = \frac{200}{100} = 2$$

This is **sub-linear** scaleup: we require 2.2 times more resources to perform 2 times the amount of work.

<u>Bottom line:</u>

- More processors → can process more data (in the same time)
- Linear scaleup ideal (but not always possible).

The **Gustafson-Barsis law** is a formula that gives the theoretical scaleup that can be expected of a system whose resources are improved.



Gustafson–Barsis's law:

$$\text{Scaleup}(P) = (1 - \alpha) + \alpha \cdot P$$

where

- P denotes the number of parallel processors
- α denotes the fraction of the task that benefits from parallelism;
- $1 - \alpha$ denotes the fraction of the task that runs inherently sequential.

→ this is what matter when we talk about big data algorithms !

Embarrassingly parallels requires very few supersteps !

In other words:

- An **embarrassingly parallel** problem is a problem where α ≈ 100%, i.e., there is little or no inherent sequential computation involved (e.g.. “count occurrences of Belgium”).
-

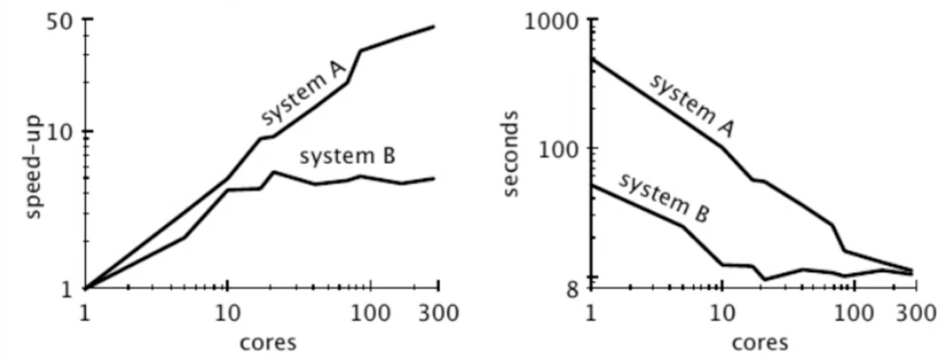
In other words:

- When phrased as a BSP algorithm, the number of supersteps S is fixed (does not depend on problem size), and the total work W and total communication H are inversely proportional to the number of processors P.

Scalability, but at what cost ?

We should not only look at **speed-up**, the speed improvement to compare two algorithms but also at the time it takes by adding more resources , the **latency** !

→ Before spending resources, we should first check why the algorithm is slow, try to improve it and then you can use resources. Because, when you speed-up, you could speed the not efficient part of the algorithm …



Bottom line:
Speedup and scaleup don’t mean anything if they are because of system inefficiencies (overheads) that are parallelizable.

Define the **COST** of a system (on a given workload/dataset) to be the Configuration at which it Outperforms a Single Threaded optimized algorithm.

The COST hence quantifies when it becomes useful to use a distributed processing system for a given workload

- If the COST is high, a Single-Threaded implementation may actually be more preferable from an economic viewpoint.
- Some systems have unbounded COST on certain graph-related problems.

Conclusion

- Big-data programming frameworks such as M/R and Spark can allow analysis of huge datasets.
- These frameworks introduce their own overheads; avoid network communication if possible.
- The BSP model is natural theoretical model for the formulation and analysis of distributed parallel algorithms.
- There are limits to speedup; scaleup is more favorable.
- Big-data programming frameworks are great for “embarrassingly parallel” problems.
- For problems that are more difficult to parallelize (e.g. graph problems), it may not always make sense to use distributed processing. A good central algorithm can go a long way.

Example of exam question :

No details, general understanding !

- what is the difference between speed-up and scale-up
- algorithm in the BSP model, we should be able to understand the complexity and the cost of this

We should apply what we have seen to real problems