09.Distributed Platforms for Data Analytics

Data Science

Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from data in various forms, both structured and unstructured
We have a huge amount of data and we want to extract information from them. Data science is made possible by the increasing availability of large volumes of data(Big data).
Examples:

- Recommender algorithms (Netflix)
  - Based on historical data
- Google translate
  - Based on snippets of books in different languages, websites written in different languages
- Genomic data
  - Correlation between genes and diseases
- Medicine
  - Lifestyle and environment variables monitored through smart devices (smartphones, watches, bands, …)

Technical Perspective

Collect all the data

- The more the better → statistical relevance
- Keeping all is cheaper than deciding what to keep
  Decide independently what to do with data
- Run experiments on data when question arises
  Huge difference with respect to traditional information systems
- Decide upfront what data to keep and why
  Some assumptions on the characteristic of the data and the environment make the calculation useless in some cases. Store a lot of data and run a lot of queries on the data to extract patterns from them, don't discard anything.

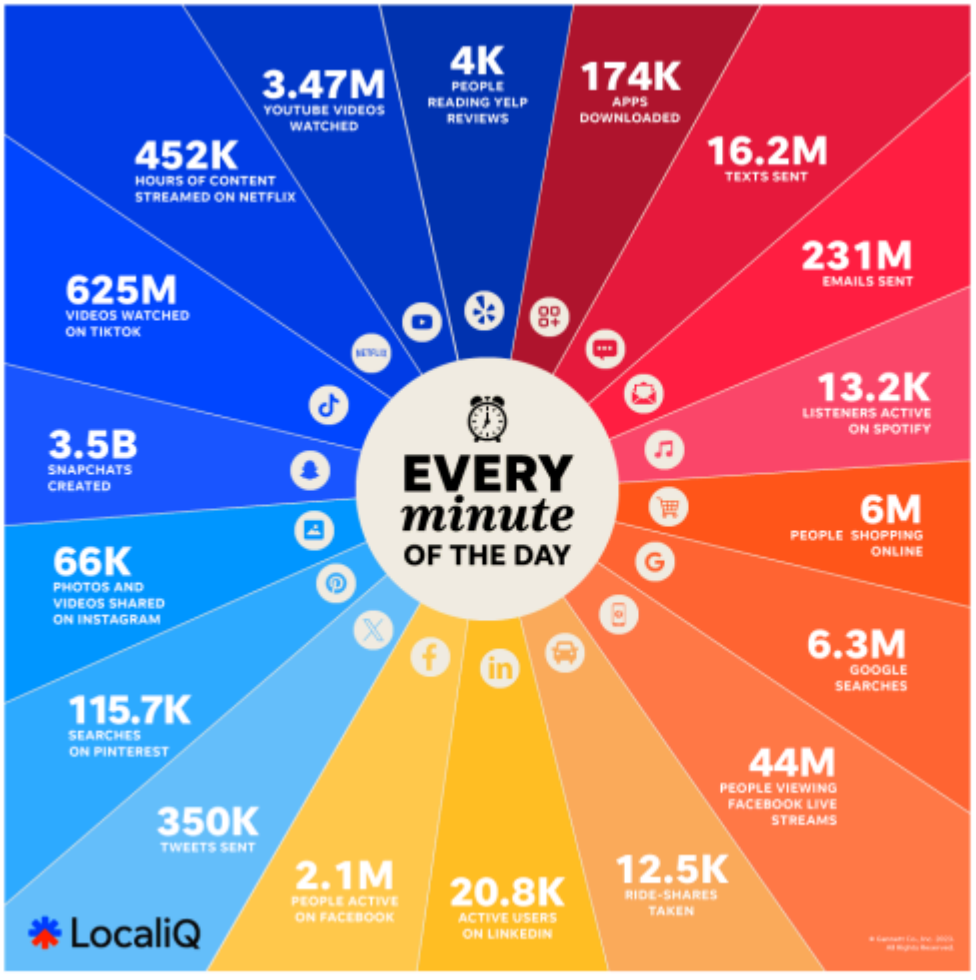
Multiple Vs

- Volume: data at rest(it is going to be a lot of data!)
- Velocity: data in motion, data changes really fast(It is going to arrive fast). It also means that, depending on the scenario, you need to react quickly.
- Variety: many different formats(Different versions, different sources). Not possible to use typical assumptions on structured data, you need to integrate different type of data.
- Veracity: not always correct. You need to collect a lot of data to discriminate between things that are correct and things that are statistical outlier or incorrect.

Volume

Back in 2008, Google was processing 20 PB (20k TB) of data every day
Exponential grows over the years.

Velocity



Internet minute [2024]

In many domains, information is relevant when it’s fresh and loses value as it becomes old
We need to process data and extract valuable knowledge as soon as possible!

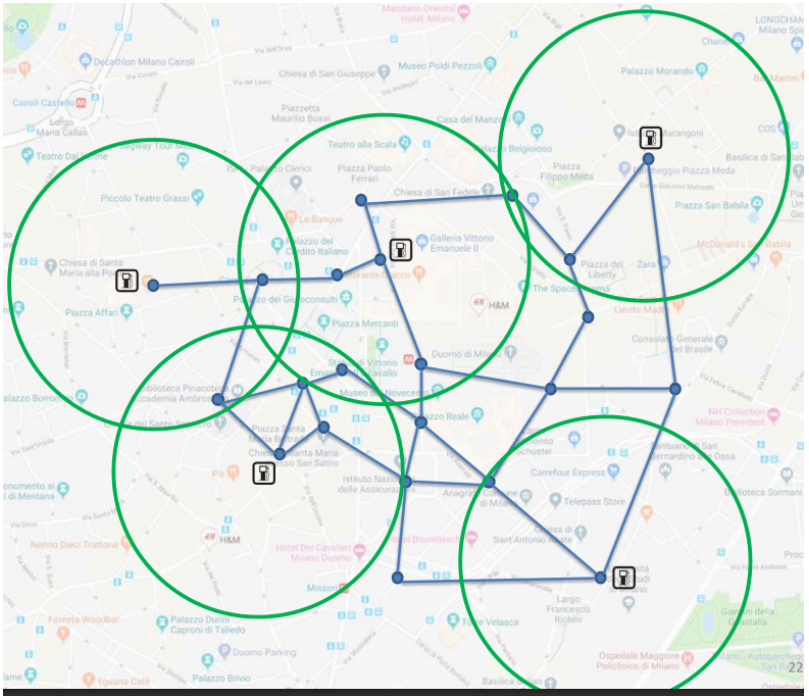
Problem Scope

Scalability to large data volumes
Support for quickly changing data
Required functions:

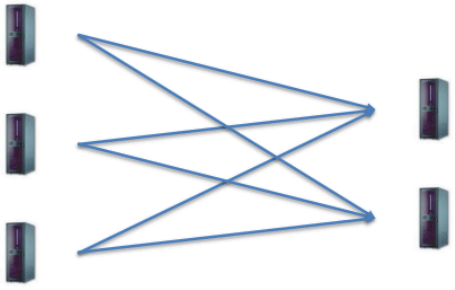
- Automatic parallelization & distribution
- Fault-tolerance
- Status and monitoring tools
- A clean abstraction for programmers
  You want to run really complex queries with a high level really simple abstraction for data science

MapReduce(Early 2000s)

Problem: we have a map represented as a graphs with nodes are points of interest and edges are roads. We want to compute the shortest path from each point to the nearest gas station.
The idea is to start from a gas station and propagate to each near nodes and save the distance from the gas station(shortest path tree). Then compute the shortest path for each node.
We know the solution but doing so much reading/writing on disk can become a bottleneck.
We need to have a way to distribute the load of the algorithm to have the bottleneck not prohibitively expensive(distribute reads and writing on disk). We can exploit domain knowledge to partition data to perform computations on the data independently on different geographical part of the system.



The first step of the computation is parallel at this point.
We cannot move in parallel on the second part of the algorithm as we need to distribute information of multiple nodes that are the intersection of multiple geographical area(requires minimum of the intersections). We want to reorganize our data in a way that it is organized by points of interest. We can resume reasoning in parallel way.
The solution is to use some mutable data/data structure

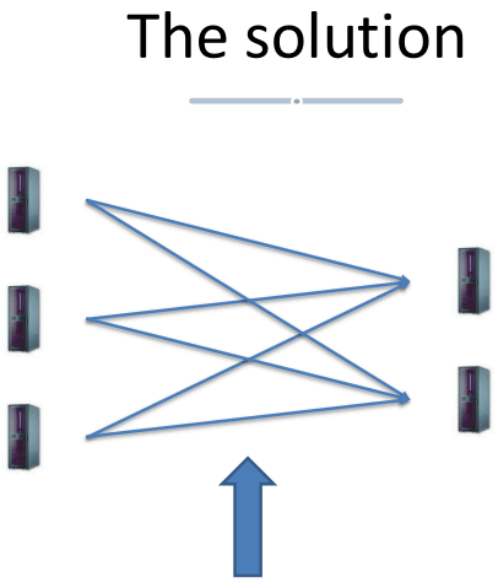


Computation of paths
- Data split by gas station
- Each gas station can be processed in parallel
- Nodes can read input blocks in parallel

Computation of shortest paths
- Data split by node
- Each node can be processed in parallel

You need to asset the data, you need to store the partial result, you need to reason on where are the data. You partition the problem to have all the little problem independent from each others and then compute them in parallel. Then you make aggregation of partial results and then make sure that the aggregation is available to the process that is computing the minimum.
You need a cluster of not so expensive machines and they aren't uniform. You take multiple machines and tries to use the parallel power of these machines instead of

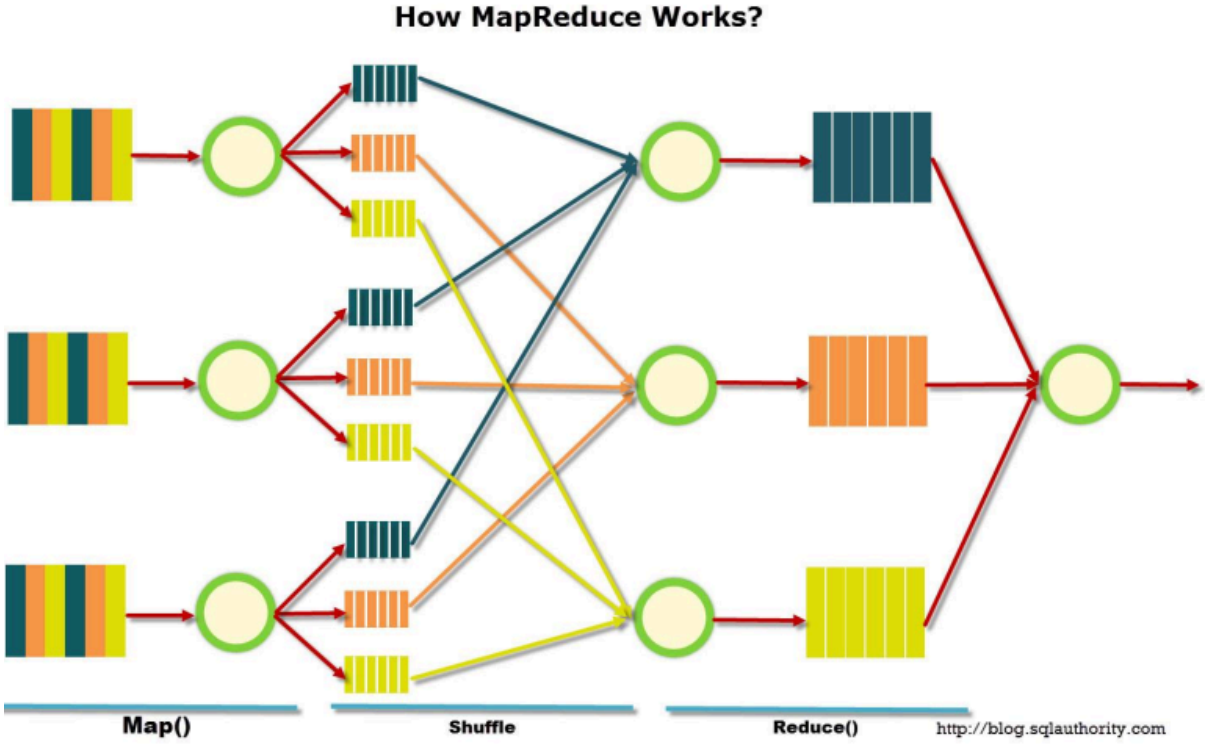
having a small number of really powerful dedicated HW. This is advantageous as you can scale it really easily, don't need to upgrade it all the time to the maximum and can only upgrade the part of the system that is lagging behind and is better fault tolerance as failures happens anyway.
The disadvantage is that having a lot of heterogeneous machines you need a good abstraction layer to permit the intercommunication and you will have a lot more machines down in the same time.



From global and *mutable state* to transformations of *immutable data*

The previous example exemplifies MapReduce
Programming model introduced by Google in 2004
Enables application programs to be written in terms of high-level operations on immutable data
The runtime system controls scheduling, load balancing, communication, fault tolerance, …
The computation is split into two phases: Map and Reduce

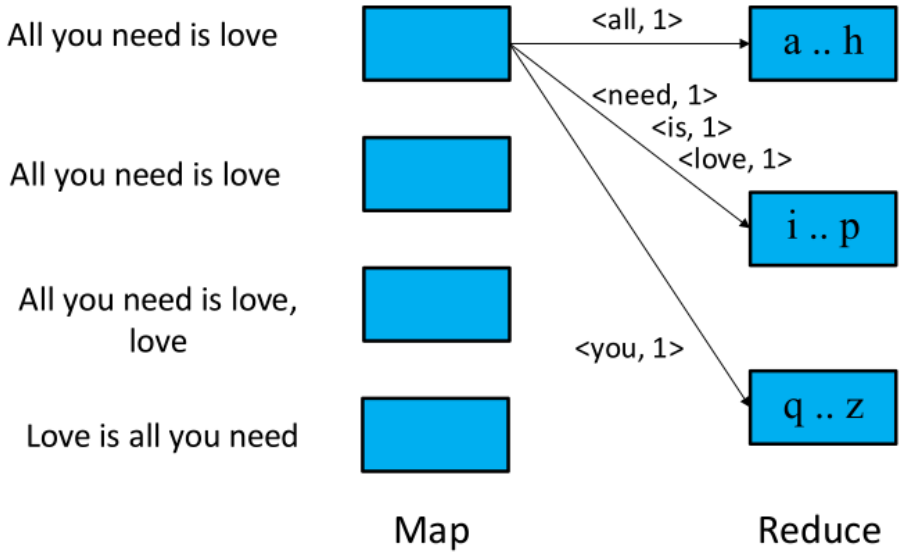
- Map processes individual elements
  - For each of them outputs one or more <key, value> pairs
- Reduce processes all the values with the same key and outputs a value
  The developers need only specify the behaviour of these two functions, only need to say how to compute the shortest path of a single gas station and how to aggregate the results of a single gas station.



Word Count

For each word w in its part of the document, the map function outputs the tuple <w, c>:

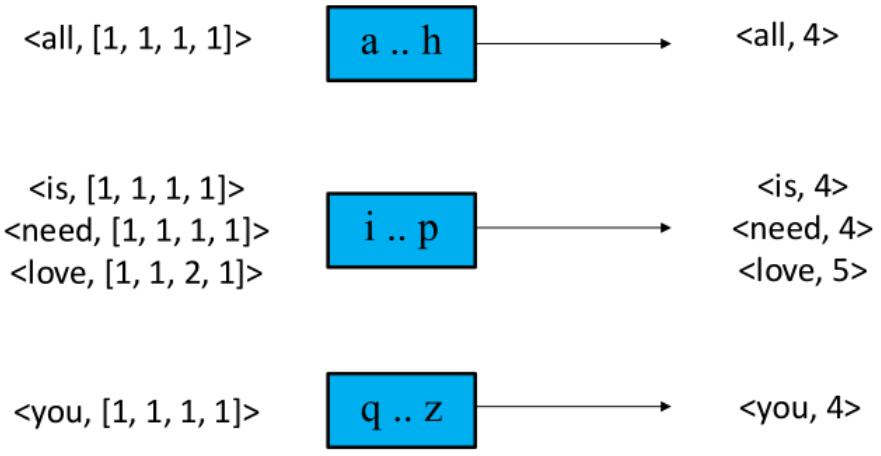
- w is the key
- c is the count
  The map function is stateless: its output depends only on the specific word that it receives in input
  Tuples with the same key are guaranteed to be received by the same receiver



We are moving from having a global space for the result of data to a single local partial results from all the processes. Need to give the same key to the same process to permit a global view of the key.

Reducers receive an immutable list (actually, an iterator) over the values associated to each key

- Why? To ensure that the data is read sequentially, from the disk, only once
  The “reduce” function iterates over the list and outputs one or more values for each key
  In our case, it outputs the sum of the elements in the list



Reduce

What does the programmer need to write? For the map function, what a mapper has to do
with its part of the document. For the reduce function, how to process the list of values associated to a given key.

```
// key: document name
// value: document contents
def map(key: String, value: String):
count = []
for w in value:
        if not w in count:
                count[w] = 1
        else:
        count[w] = count[w] + 1
for w in count:
        emit(w, count[w])

// key: a word
// values: a list of counts
def reduce(key: String, values: Iterator):
        result = 0
        for v in values:
```

```
                result += v
        emit(result)
```

You use an iterator to force the programmer to read a value at a time without the possibility to go back to the other values. Maximise performance on loading values from disk.
What does the platform do?

- Scheduling: allocates resources for mappers and reducers.
- Data distribution: moves data from mappers to reducers
- Fault tolerance: transparently handles the crash of one or more nodes

MapReduce scheduling

One master, many workers

- Input data split into M map tasks (typically 64 MB)
- Reduce phase partitioned into R reduce tasks (hash(k) mod R)
- Tasks are assigned to workers dynamically
- Data are scattered to the node in the system in a replicated way(usually at least three time)
  Master assigns each map task to a free worker
- Considers locality of data to worker when assigning a task
- Worker reads task input (often from local disk)
- Worker produces R local files containing intermediate k/v pairs
  Master assigns each reduce task to a free worker
- Worker reads intermediate k/v pairs from map workers
- Worker sorts & applies user's reduce operation to produce the output
  Data is replicated in multiple places. There is a leader-worker schema, there is a leader that knows where the data is stored. This is important as the scheduler is responsible to dynamically spawn process with the map function it spawns them the nearer possible to where the data is.
  The engine also spawn reducer using the same principles.
  We have transformation of immutable data $\Rightarrow$ we don't have state as we can discard the previous data as they are useless. Problem is how to handle struggler nodes(nodes that cannot give you the result in an acceptable time): you resolve it scheduling the computation on multiple nodes and use the result of the first one to complete and discard the results from the others.
  MapReduce is good when you can run data parallel computation.

Conclusions

Strengths

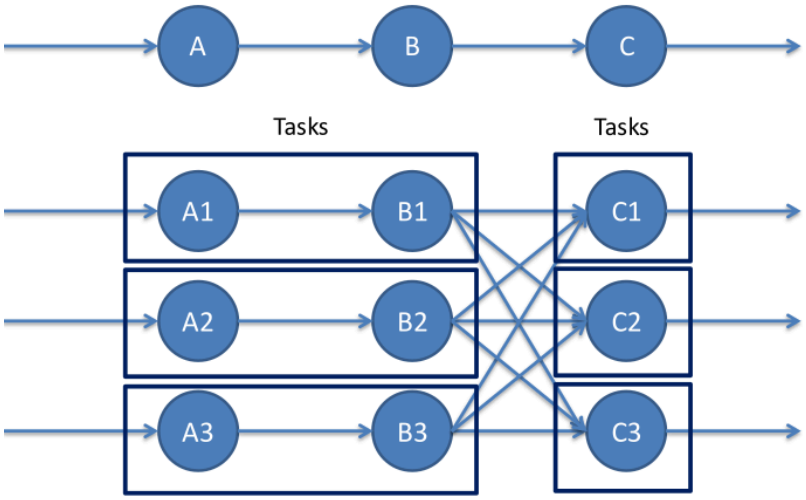
- The developers write simple functions
- The system manages complexities of allocation, synchronization, communication, fault tolerance, stragglers, …
- Very general
- Good for large-scale data analysis
  Limitations
- High overhead
- Lower raw performance than HPC
- Very fixed paradigm
  - Each MapReduce step must complete before the next one can start

BEYOND MAP REDUCE

In the last decade, many systems extended and improved the MapReduce abstraction in many
ways

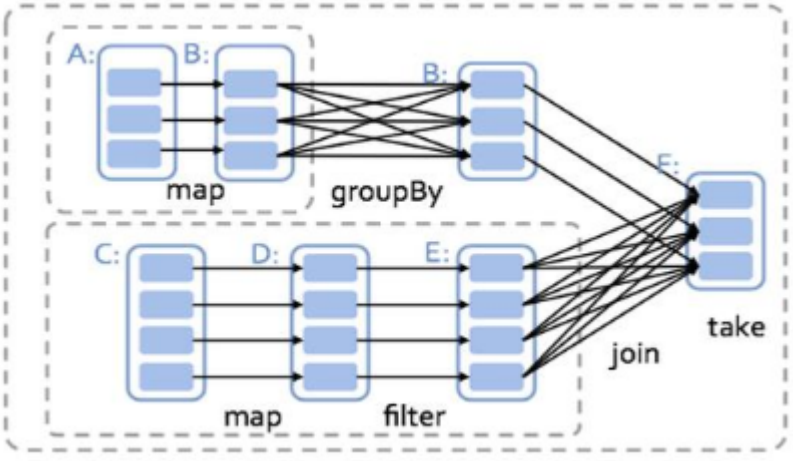
- From two processing steps to arbitrary acyclic graphs of transformations
  - Dataflow model
  - In some cases, support for iterative computations
- From batch processing to stream processing
  - Not only process large datasets with high throughput …
  - … also, low latency
  - From disk to main-memory or hybrid approaches

Dataflow programming model



Nodes are the operators in which the data are transformed, then you have multiple replication of data in the structure to perform several step of computation without changing the way data is partitioned.
You can do transformation of data if there aren't cycle on the nodes, use the parallelism of the structure as you assign computation on multiple nodes in parallel.
Exploit main memory: common to have 64 or more Gb of RAM so instead of using disk to store intermediate results between two stages(Caching memory in RAM other than the disk). DOing so when you schedule operations that use the data in memory the operations become much faster.
Two types of architecture:

- Scheduling architecture: (Ex is Apache Spark) similar to MapReduce, the idea is that stores the data also in memory and it split the data and schedule the computation on different nodes as close as possible to the nodes. Tries to maintain the same partition of the data. Typically developers have a initial structure on where you can do the map and reduce functions and then a set of libraries to do more complex operations. For each partition there is a task(unit of work of Spark). Each stage consist of multiple tasks run in parallel(the tasks are scheduled on the free nodes in the system).



- Pipelining: the tasks are active at a certain point in time, data enter the system, enter in the first task and the results, once computated, flows to the next task that needs it(No intermediate store, there are channels that connect tasks between them). Ideal for stream processing. Data flows into the system without waiting for the operators to be scheduled. Lower latency. Same approach used for batch processing. By “streaming” the entire batch through the operators

The advantage of a pipelining approach is that you can exploit all the parallelism that your HW can support, scheduling need to wait for all the upstream tasks to complete before starting the downstream tasks. In pipelining the scheduled tasks remain active until taken down and you can input them data and receive results continuously. You don't need to schedule process as the machine that compute on data is already there. It provides no latency. Can also work for static data you just need to create a stream of data from this static data.
Scheduling approach need static data. We can batch the data and create a stream of data with these batch.
Problems: if you schedule in batch you still need to compute the mean on the previous data so these batch aren't independent from the previous data. You can resolve this problem if you store the previous state of the system produced by the previous micro-batches.

Comparison: latency

The pipeline approach of Flink provides lower latency

- Relevant for stream processing scenarios where new data is continuously generated
- New data elements are ingested into the network of processing operators as they become available
- No need to accumulate (micro-)batches
- No need to schedule tasks

Comparison: throughput

A scheduling approach offers more opportunities to optimize throughput

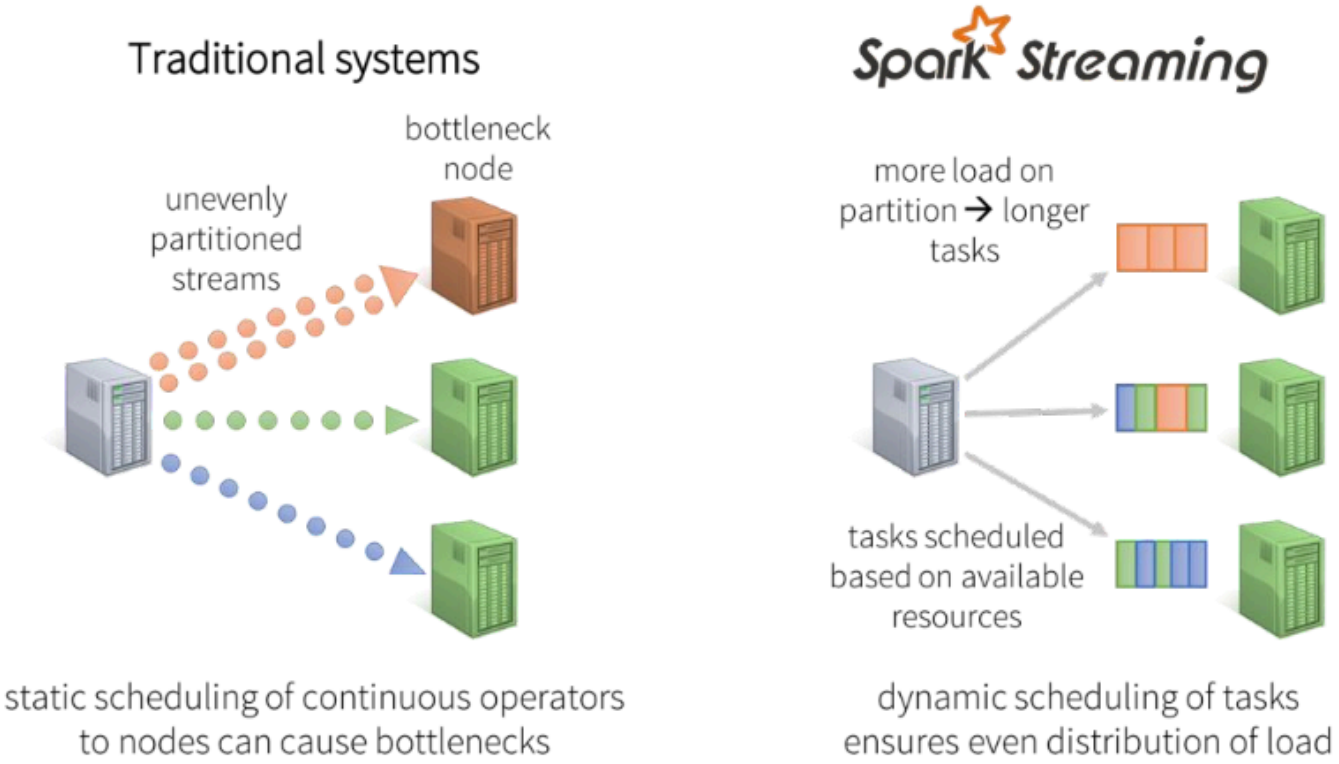
- Moving larger data blocks can be more efficient
  - Smaller overhead from the network protocols
  - More opportunities for data compression
- Scheduling decisions can consider data distribution

- See the comparison on load balancing below

Comparison: load balancing

The scheduling approach of Spark simplifies load balancing

- Dynamic scheduling decisions can consider data distribution
- This is not possible in the case of a pipelined approach
  - Allocation of tasks to operators is decided statically when the job is deployed
- Also relevant for streaming workloads
  - The distribution of data may change over time



Comparison: elasticity

We are considering repartition of task in base of the workload in the system, we want to pay only for what we are using. We need to allocate more/less nodes in base of the quantity of data that are arriving.
Big data processing platforms are often offered as a service
The actual cost of the service depends on the amount of resources being used
In the case of continuous/streaming jobs, the load can change over time
Elasticity indicates the possibility of a system to dynamically adapt resource usage to the load of the system
Scheduling approaches are better for elasticity

- Scheduling decisions take place dynamically at runtime
- It is possible to dynamically change the set of physical resources being used for deployment
  • This may require moving intermediate state in the case of stream processing jobs
  Elasticity is simply not possible in pipelined approaches
- Only opportunity: take a snapshot of the system and restart it on a different set of physical node
  We can implement it using snapshots to know how the system was before shutting it down to add more resources(this for pipelining)

Comparison: fault-tolerance

What happens if a node fails?

- Scheduled processing
  - Re-schedule the task
- Apache Spark relies on the lineage
  - No replication of intermediate results
  - If a data element is lost simply recompute it
  - If the data it depends on is lost simply recompute it
  In pipelined processing, periodically checkpoint to (distributed) file system
  In the case of failure, replay from the last checkpoint
  Apache Flink relies on checkpointing
- You have seen the Chandy Lamport fault tolerance algorithm in detail in previous lectures
  Scheduling recovery mechanism is done recursively as it need to have the minimum recomputed data from the host to resume operation after the crash.
  In pipelining the recovery process works on snapshots(similar to Lamport)