



POLITECNICO  
MILANO 1863

# Big Data Platforms

Alessandro Margara

[alessandro.margara@polimi.it](mailto:alessandro.margara@polimi.it)

# License

This work is licensed under the Creative Commons Attribution-ShareAlike International Public License



# 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”

Wikipedia

# Data science

- 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
  - Genomic data
    - Correlation between genes and diseases
  - Medicine
    - Lifestyle and environment variables monitored through smart devices (smartphones, watches, bands, ...)

# Business perspective

- It is a business model
- Users pay with data
  - Google, Facebook, ...
  - Use service → give data
- Google and co. sell your data to advertisers
  - The user pays the advertisers indirectly

# 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

# Consequences

- Volume: data at rest
  - It is going to be a lot of data!
- Velocity: data in motion
  - It is going to arrive fast
- Variety: many different formats
  - Different versions, different sources
- Veracity: not always correct

# Volume

- How Big is Big Data?
- Apparently Google in 2009 processed around 24 petabytes of data every day!!!
- Is it a lot? Let's see...

# Volume

- Let's imagine that a byte is represented by a single grain of rice
  - A kilobyte = a bowl of rice
  - A megabyte = a sack of rice
  - A gigabyte = two containers full of rice
  - A terabyte = 2048 containers = enough for two meals for everyone in the European Union (about 500 million people)
  - A petabyte = 135 of the largest ever container ships = feed the entire planet with 146 bowl of rice or cover New York City with one meter of rice

# Volume

- That is one petabyte
- Google in 2009 got 24 petabytes per day
  - Or 1920 bowls of rice for everyone on the planet
  - Or covering New York City with 24 meters of rice
- Thanks to:  
[http://noiseinmyhead.wordpress.com/2008/06/26/  
visualizing-huge-amounts-of-data/](http://noiseinmyhead.wordpress.com/2008/06/26/visualizing-huge-amounts-of-data/)

# Volume

- And the volume is increasing!
- “Every two days we create as much data as much we did from dawn of humanity to 2003”
  - Eric Schmidt, Google

# Velocity



# Velocity

- 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!

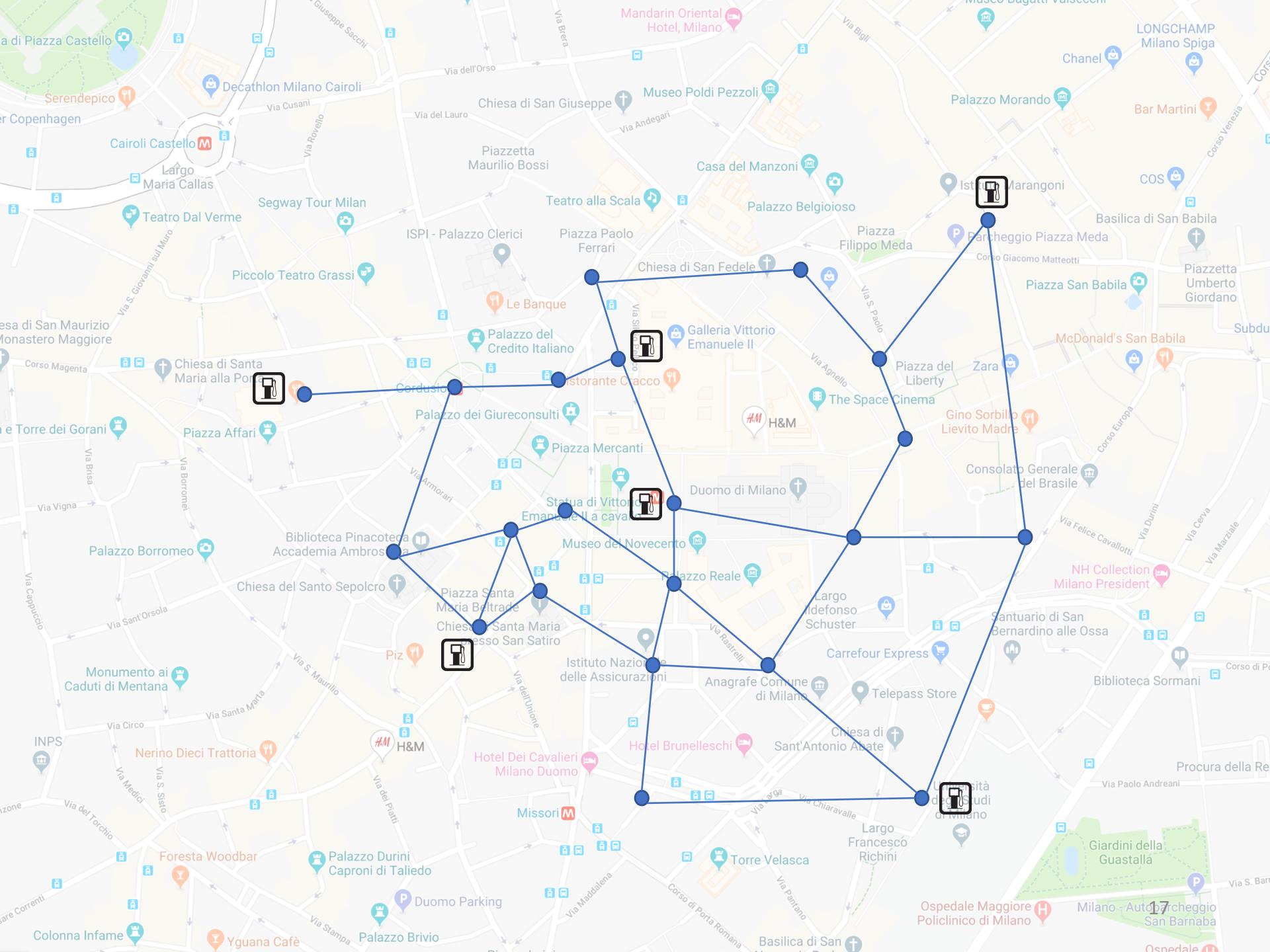
# Problem scope

- 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

# Map Reduce

# Problem

- We have a map represented as a graph
  - Nodes are points of interest
  - Edges are roads
- We want to compute the shortest path from each point to the nearest gas station



# Solutions?

- Compute the shortest path between each gas station and each node
  - This is the single-source shortest path problem
  - Well known algorithms (Dijkstra, Johnson, ...)
  - If you are curious about the complexity: the best known algorithm works in  $O((N + E) \log N)$ , where  $N$  is the number of nodes and  $E$  is the number of edges, and we need to do that for each gas station
- Then, for each node, we need to compare the distance computed from each gas station and keep only the minimum
  - E.g., we can store the best known solution for each node and update it as we discover better paths

# Solutions?

- Possible optimization: we can discard nodes that are too far away
  - E.g., we assume that the maximum distance between any node and the nearest gas station is 50 km and we discard paths that are longer than that

# Solutions?

- Did we ask ourselves the right questions?
- We implicitly made some strong assumptions
  - We can easily access the entire input data
  - We can easily store and update the state of the computation
    - Discovered paths, current minimum for each node, ...
  - They fit in memory ...
  - ... or at least on the disk of a single node
    - Random access can then become a problem

# Scalability

- In general, some problems only appear “at scale”
- Reading input and reading/storing the intermediate state of the computation can easily become the bottleneck
- If input/state do not fit in one machine, we need distributed solutions ...
- ... and then the coordination, communication can become the bottleneck

# Scalability

- We need to consider
  - The scale of the problem
    - For now, let's focus on the volume of data only
  - The computing infrastructure we have
- Our story takes place in a datacenter of Google in the early 2000s

# Assumptions: scale of the problem

- Terabytes or more
- Does not fit into a single disk
  - Or it is too expensive to read from a single disk
  - Sadly, there were no SSDs at that time ...
- For sure does not fit in memory

# Assumptions: computing infrastructure

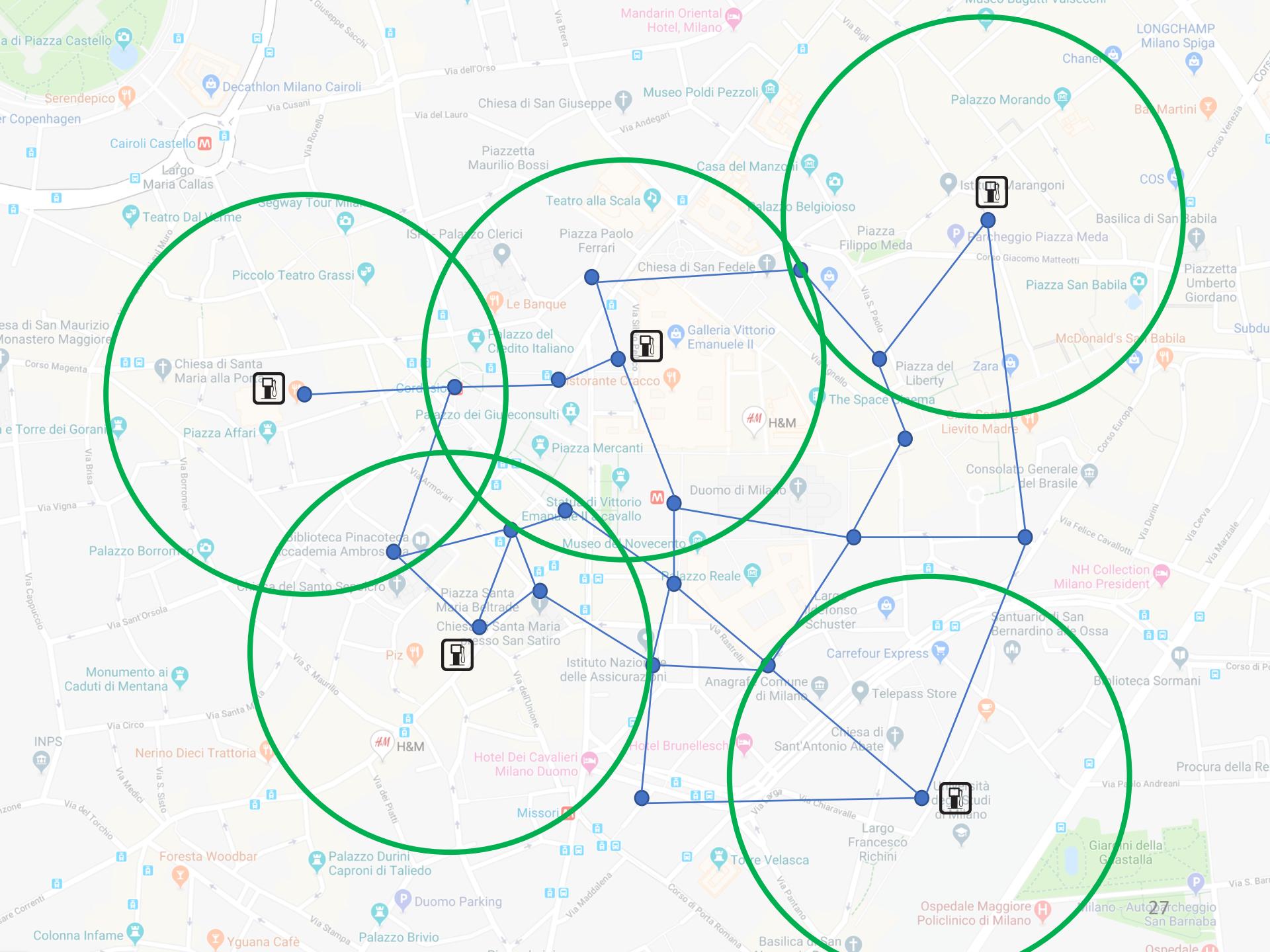
- Cluster of “normal” computers
- Hundreds to thousands of nodes
- No dedicated hardware
  - Less expensive
  - Easy to update frequently and incrementally
  - **Not reliable! Hardware failures are common!**

# Back to the problem

- Now we have a clear picture of the assumptions ...
- ... better solutions?

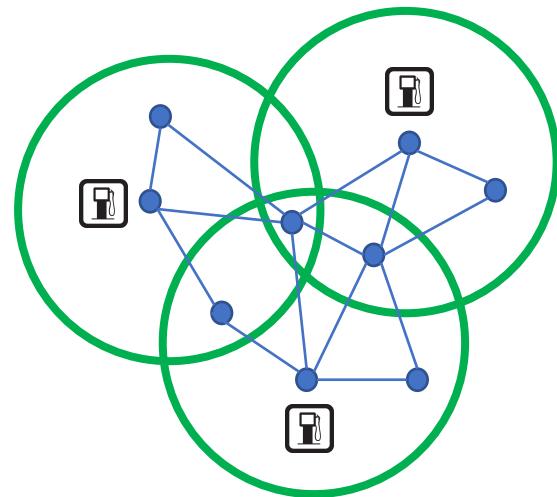
# Back to the problem

- Assuming that the maximum distance between any point and the nearest gas station is 50 km ...
- ... we can split the dataset into blocks
  - Each block  $b_i$  centered around gas station  $i$
  - Different blocks can be stored on different computers
  - For each block  $b_i$ , we can compute the distance between gas station  $i$  and any node in the block
    - The computation is independent for each block
    - Blocks can be processed entirely in parallel



# Back to the problem

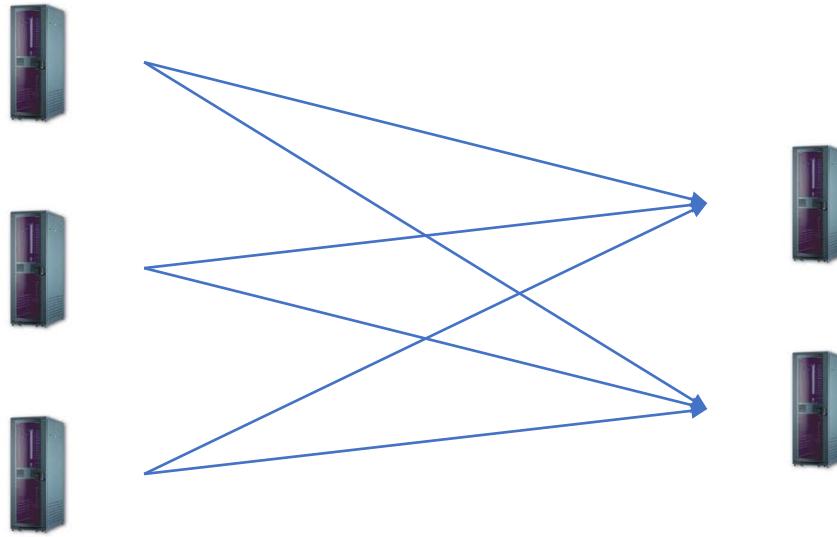
- Next, we need to determine the closest gas station and the shortest path for each node
- This requires communication, because a point can be at the intersection of several blocks



# Back to the problem

- We can repartition the data by node
- Again, the shortest path (minimum distance) can be computed independently for each node
  - Each node can be processed in parallel on a different machine

# The solution



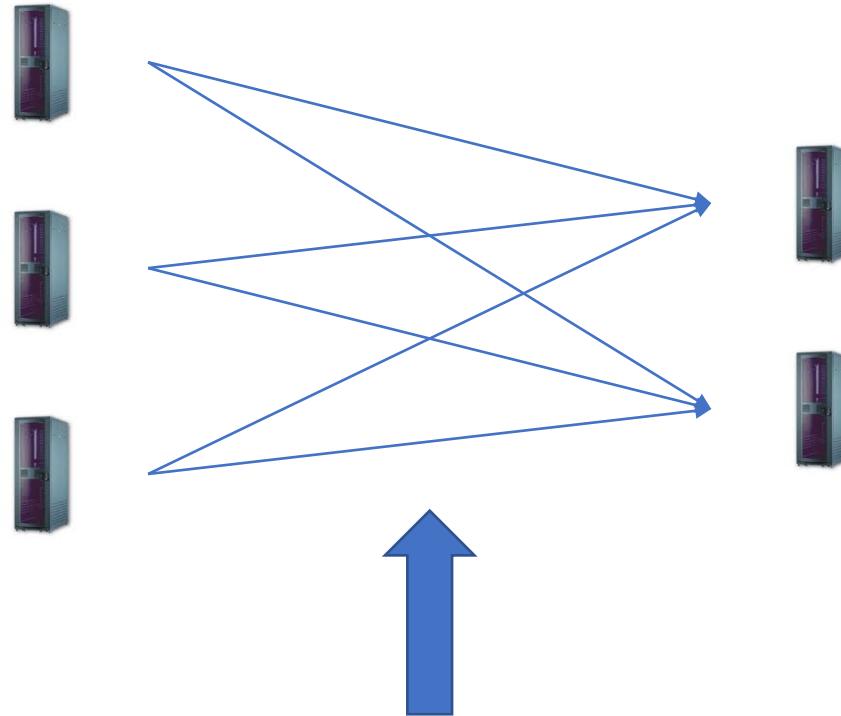
## Computation of paths

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

## Computation of shortest paths

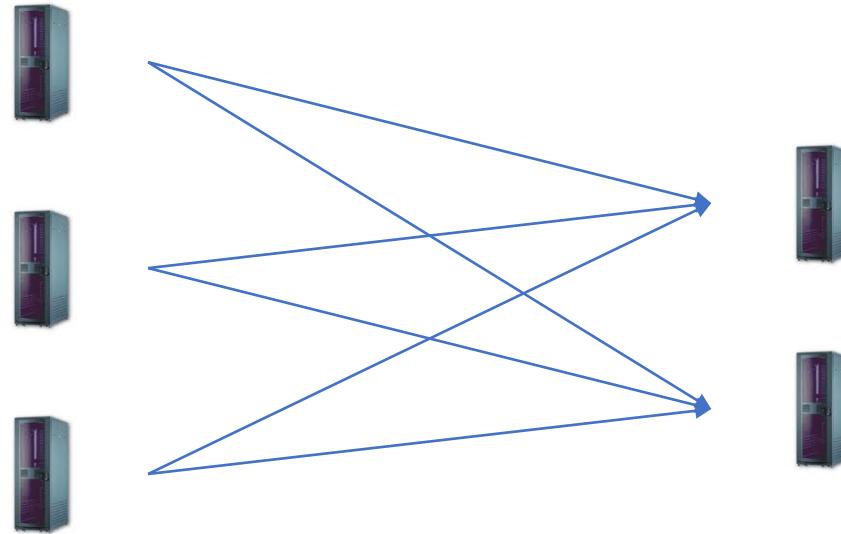
- Data split by node
- Each node can be processed in parallel

# The solution



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

# The solution



## Different channels

- Distributed filesystem
- Messages (e.g., TCP)
- Queues (e.g., Kafka)

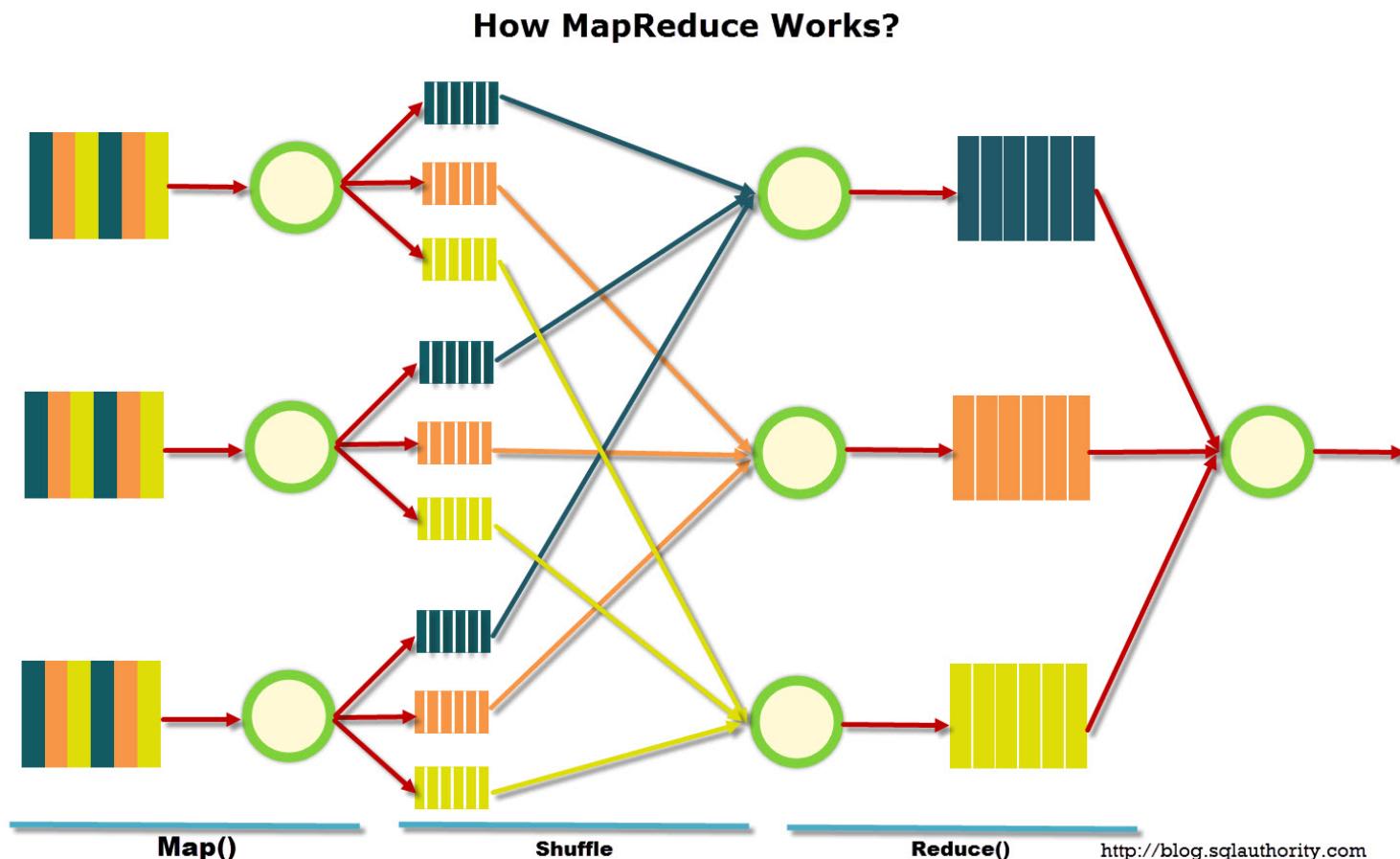
# MapReduce

- 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, ...

# MapReduce

- 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 behavior of these two functions

# MapReduce



# Another example: word count

- We want to count the number of occurrences of each word in a large document
- Example: “All you need is love”, The Beatles
  - All you need is love
  - All you need is love
  - All you need is love, love
  - Love is all you need
- The document is split into blocks, and each block is given to a different “Map” node

# Another example: word count

All you need is love



All you need is love



All you need is love,  
love



Love is all you need

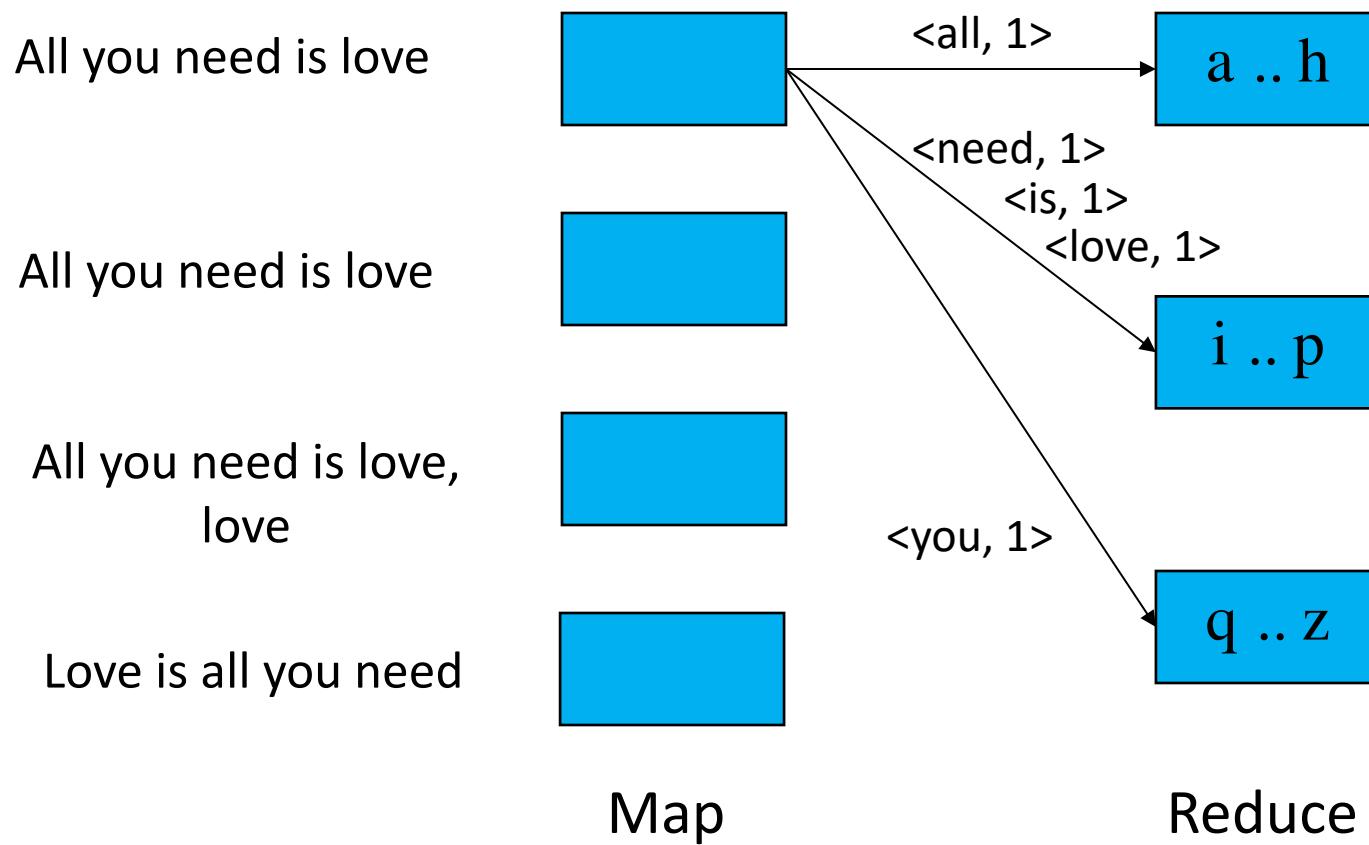


Map

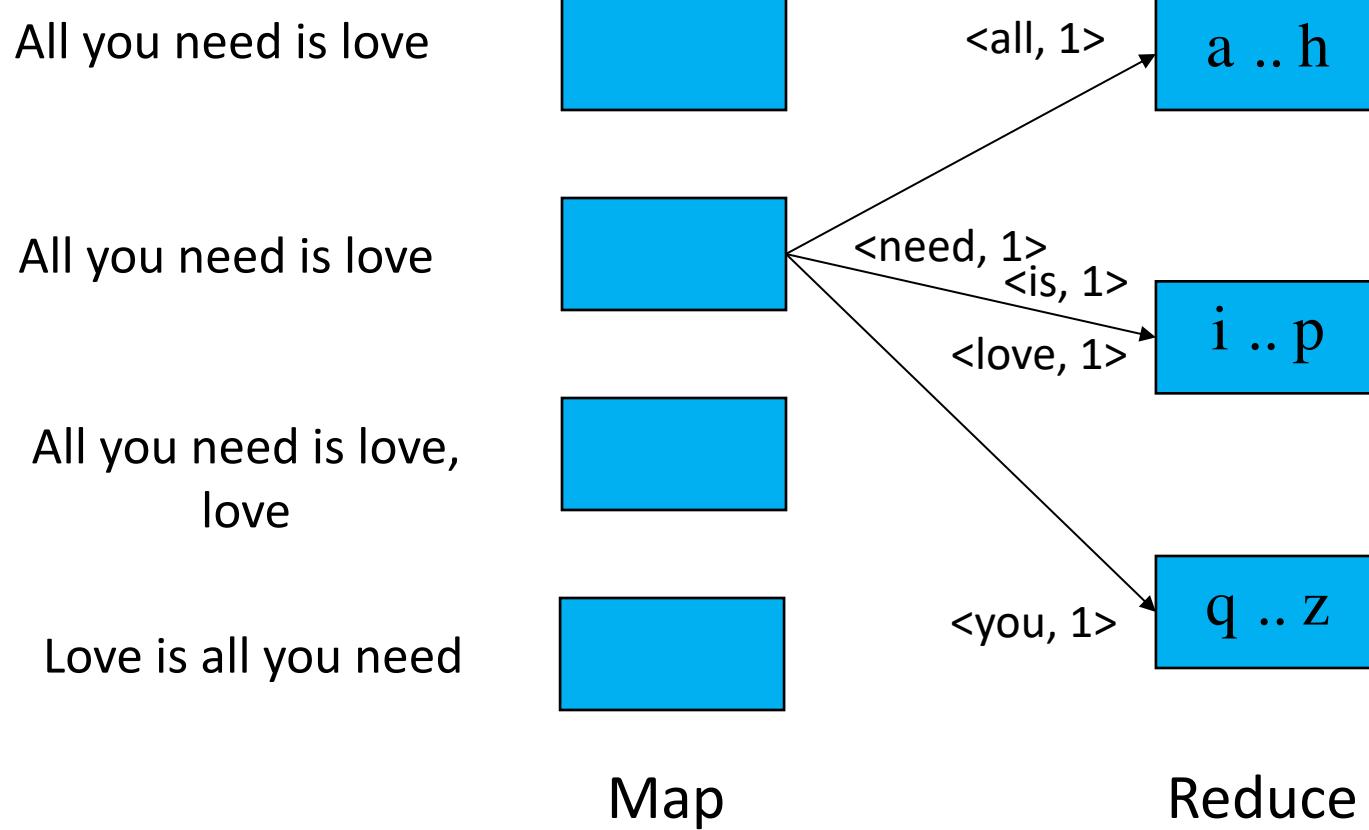
# Another example: word count

- For each word  $w$  in its part of the document, the map function outputs the tuple  $\langle w, c \rangle$ 
  - $w$  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

# Another example: word count



# Another example: word count



# Another example: word count

All you need is love



All you need is love



All you need is love,  
love



Love is all you need



Map

<all, 1>

a .. h

<need, 1>  
<is, 1>

i .. p

<love, 2>

q .. z

<you, 1>

Reduce

# Another example: word count

All you need is love



All you need is love



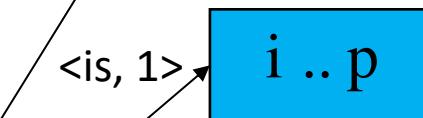
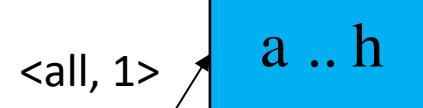
All you need is love,  
love



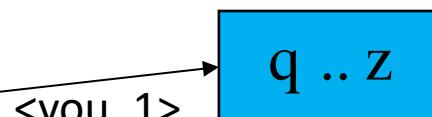
Love is all you need



Map



<love, 1>  
<need, 1>



Reduce

# Another example: word count

- 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 paradigm encourages the definition of streaming algorithms
- 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

# Another example: word count

All you need is love



<all, [1, 1, 1, 1]>

a .. h

All you need is love



<is, [1, 1, 1, 1]>

<need, [1, 1, 1, 1]>

<love, [1, 1, 2, 1]>

i .. p

All you need is love,  
love



<you, [1, 1, 1, 1]>

q .. z

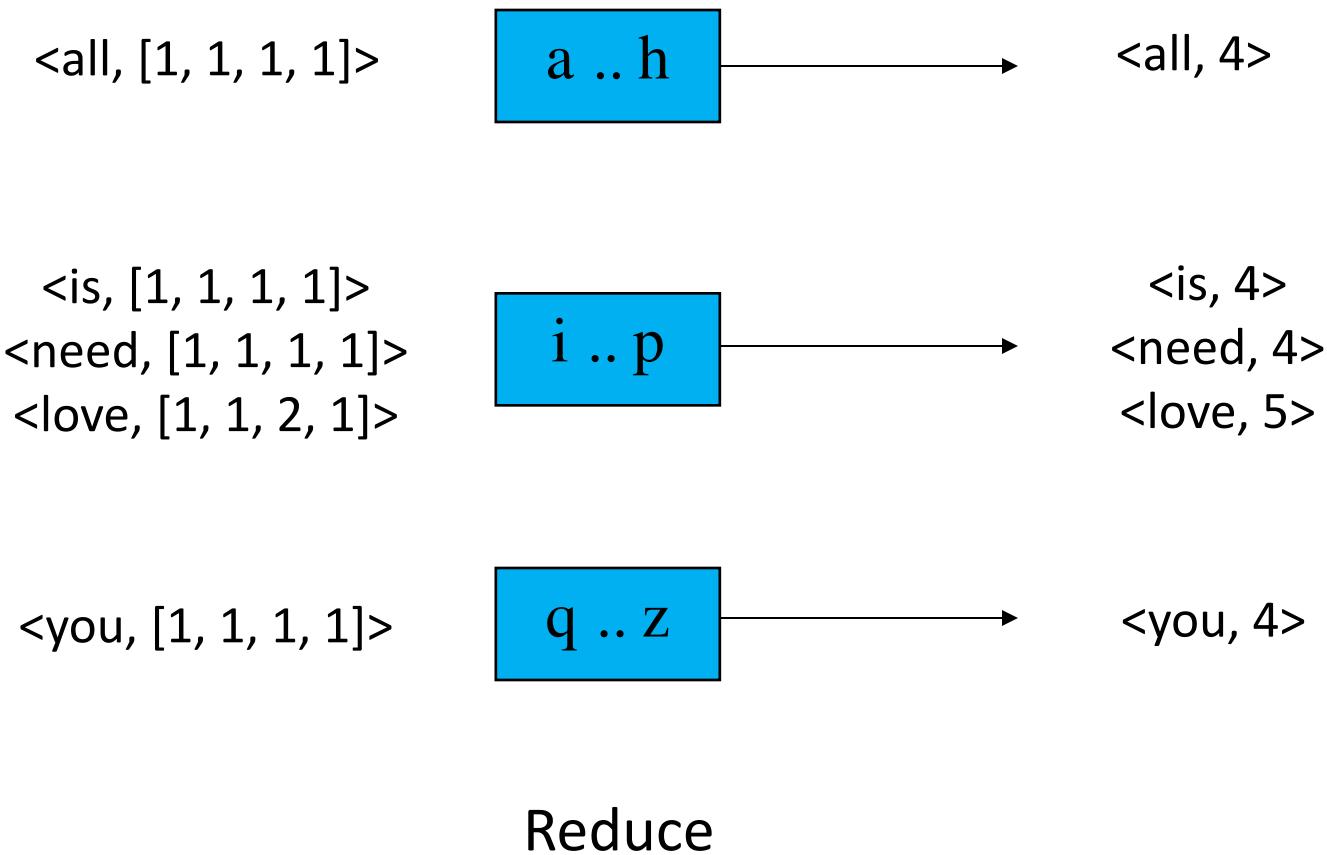
Love is all you need



Map

Reduce

# Another example: word count



# Another example: word count

- 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

# Another example: word count

```
// 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])
```

# Another example: word count

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

emit(result)
```

# MapReduce

- 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 ( $\text{hash}(k) \bmod R$ )
  - Tasks are assigned to workers dynamically
- 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

# MapReduce data locality

- Goal: conserve network bandwidth
- In GFS, data files are divided into 64MB blocks and 3 copies of each are stored on different machines
- Master program schedules map() tasks based on the location of these replicas
  - Put map() tasks physically on the same machine as one of the input replicas (or, at least on the same rack / network switch)
- This way, thousands of machines can read input at local disk speed
  - Otherwise, rack switches would limit read rate

# MapReduce fault tolerance

- On worker failure
  - Master detects failure via periodic heartbeats
  - Both completed and in-progress map tasks on that worker should be re-executed
    - Output stored on local disk
  - Only in-progress reduce tasks on that worker should be re-executed
    - Output stored in global file system
  - All reduce workers will be notified about any map re-executions
- On master failure
  - State is check-pointed to GFS: new master recovers & continues
- Robustness

# MapReduce: stragglers

- Stragglers = tasks that take long time to execute
  - Slow hardware, poor partitioning, bug, ...
- When done with most tasks ...
- ... reschedule any remaining executing task
  - Keep track of redundant executions
  - Significantly reduces overall run time

# MapReduce: conclusions

- Map and reduce function are the “core building blocks” to build large scale data analytics application
  - Transform data
  - Partition data (by key)
  - Aggregate data (with the same key)
- Typical MapReduce application
  - Sequence of steps, each requiring map & reduce
  - Series of data transformations
  - Iterating until reach convergence
    - E.g., Google PageRank

# MapReduce: conclusions

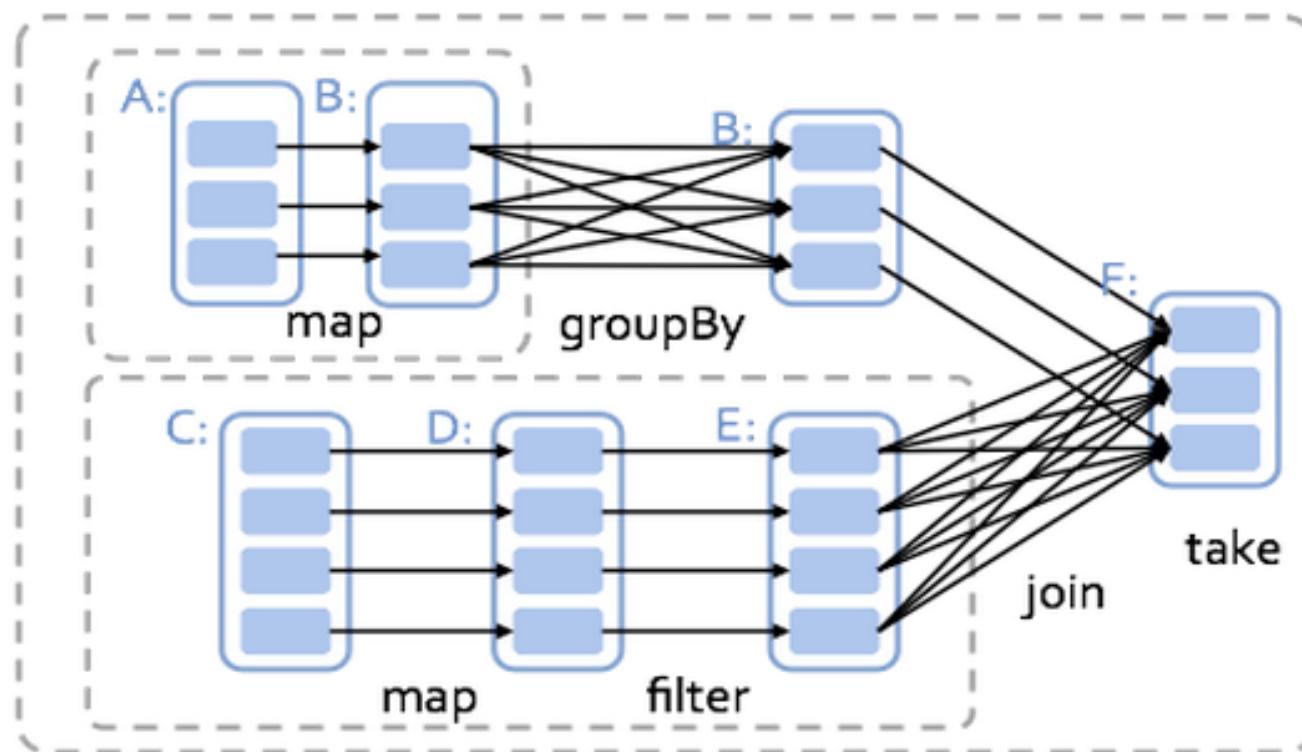
- Strengths
  - The developers write simple functions
  - The system manages complexities of allocation, synchronization, communication, fault tolerance, stragglers, ...
  - Very general
  - Good for large-scale data analysis
- Limitations
  - High overhead
  - Lower raw performance than HPC
  - Very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

# Beyond MapReduce

# Beyond MapReduce

- 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
    - In some cases, support for iterative computations
  - From batch processing to stream processing
    - Not only process large datasets with high throughput...
    - ... but also low latency
  - From disk to main-memory or hybrid approaches

# Beyond MapReduce



# Beyond MapReduce

- We consider two systems as representatives of two different architectural approaches
  - Apache Spark
    - Batch (or micro-batch) processing
    - Scheduling of tasks
  - Apache Flink
    - Streaming or continuous processing
    - Pipelining of tasks

# Apache Spark

- Similar to MapReduce
  - Instead of only two stages (map and reduce) ...
  - ... arbitrary number of stages
- Intermediate results can be cached in main memory if they are reused multiple times
- Scheduling of tasks (stages) ensures that the computation takes place close to the data

# Apache Spark

- Support for streaming data through the micro-batch approach
- The input data stream is split into small batches of data
  - Processed independently ...
  - ... but some state can persist across batches
    - For example, in a streaming word count, the count of each word is persisted and updated by each new micro-batch

# Apache Flink

- A job is not split into stages that are scheduled
- Instead, all the operators are instantiated as soon as the job is submitted
  - They communicate using TCP channels
  - An operator can start processing as soon as it has some data available from the previous ones
    - Pipeline architecture where multiple operators are simultaneously running

# Apache Flink

- 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

# 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 operators

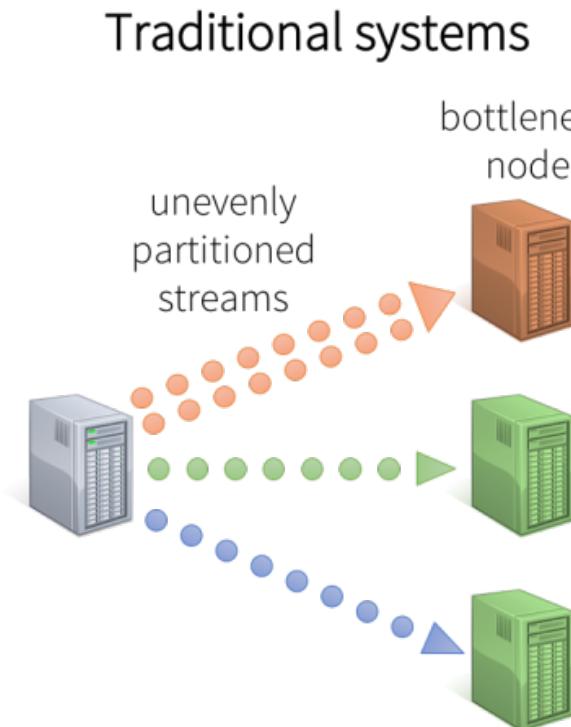
# Comparison: throughput

- 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 take into account data distribution
    - See the comparison on load balancing below

# Comparison: load balancing

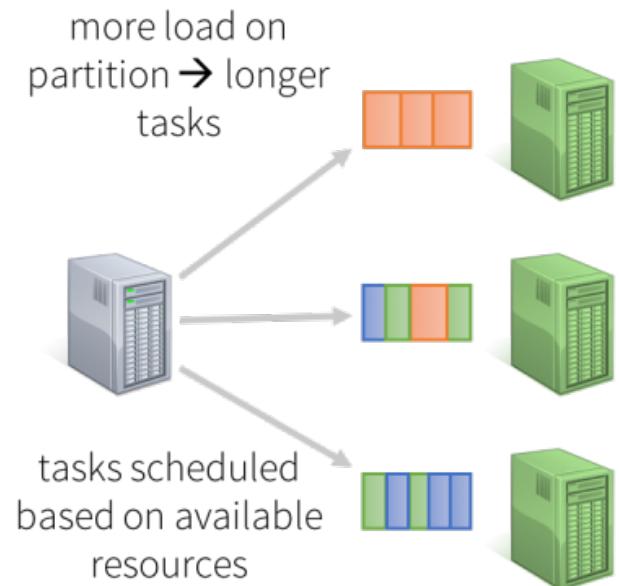
- The scheduling approach of Spark simplifies load balancing
  - Dynamic scheduling decisions can take into account data distribution
  - This is not possible in the case of a pipelined approach
    - Allocation of tasks to operators is decided statically when the job is deployed

# Comparison: load balancing



static scheduling of continuous operators  
to nodes can cause bottlenecks

## Spark Streaming



dynamic scheduling of tasks  
ensures even distribution of load

# Comparison: elasticity

- 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

# Comparison: elasticity

- 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 might 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 nodes

# 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
  - ... and so on ...

# Comparison: fault-tolerance

- In pipelined processing, periodically checkpoint to (distributed) file system
- In the case of failure, reply from the last checkpoint
- Apache Flink relies on checkpointing
  - You have seen the Chandy Lamport fault tolerance algorithm in detail in previous lectures

# Questions?

