Designing Distributed Geospatial Data-Intensive Applications

Ph.D. Course, 2022

Instructors:

Prof. Luca Foschini, Associate Professor &

Dr. Isam Mashhour Al Jawarneh, Postdoctoral Research Fellow

{isam.aljawarneh3, Luca.foschini}@unibo.it

Department of Computer Science and Engineering (DISI), Università di Bologna

Part 1

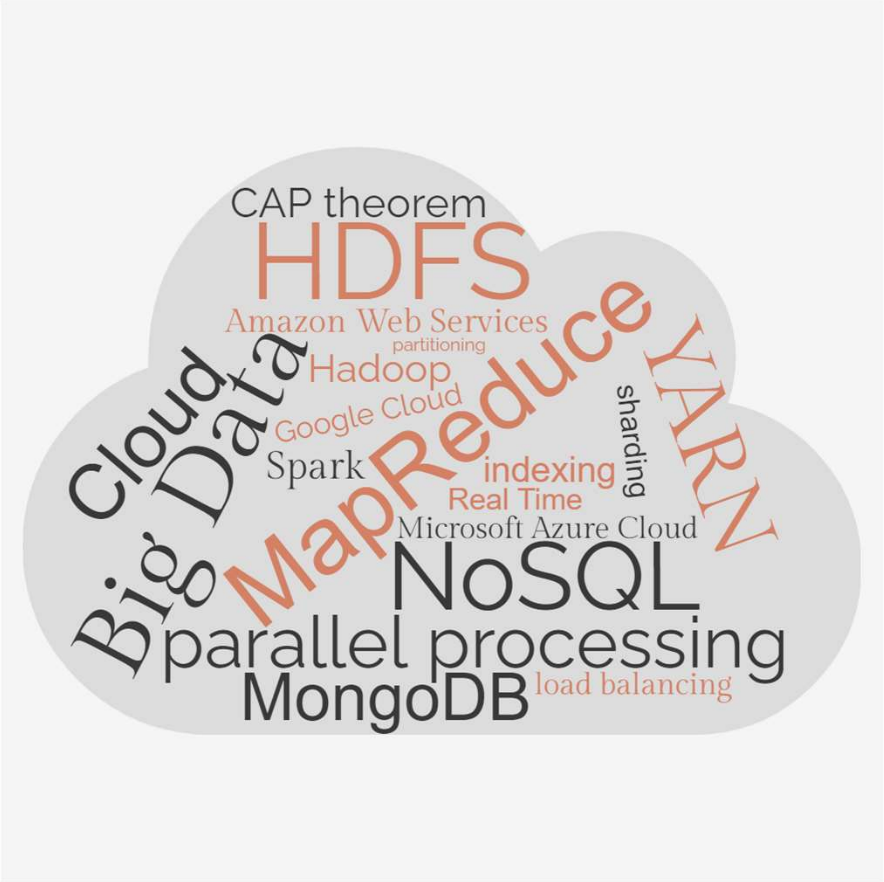
Introduction

18th July 2022

What makes an application data-intensive

Big data management

- What all those are about?
  - Big data management in distributed systems



Driving forces for distributed data management

- Unprecedented voluminous amounts of big data are generated by big tech companies such as Google, Amazon, Twitter
  - They need new tools, beyond the traditional server-based deployments, that enable management of such data, at scale
- Mature **open-source projects** are preferred over in-house counterparts
- Network transfer capabilities are becoming faster, enabling **parallelism** to become the de facto standard

Data-intensive applications

- What makes an application data-intensive
  - Data is its primary challenge
    - Data **volume**, **complexity**, **speed** of arrival & change
- Novel distributed computing tools have emerged for the **storage** and **processing** of such data
  - Scalable **distributed storage systems** (e.g., MongoDB) and **data processing** (e.g., Apache Spark & Hadoop)
  - Related technologies: message queues, caches, search indexes, frameworks for **batch** and **stream** processing

This course

- We need a deep technical understanding of the big data technologies and
  - The trade-offs of design choices for domain-specific applications
  - In this course, we are focusing on **georeferenced big data** management in **distributed computing** deployments
- It is true that the technology is rapidly changing
  - However, enduring **principles** remain valid for all tools
  - Understanding those **principles** helps us choose the right tool and add custom tools to improve its performance in a domain-specific direction
- A technological view of the landscape of tools for big data management
  - With a domain-specific focus (**spatial**)
  - With examples of successful frameworks and systems
  - A deep preview of the internal building blocks
    - It is not about how systems work; it is more about why they work in a specific way
  - Fundamental principles and trade-offs
    - Design decisions
    - Always in the scope of **spatial big data**

What makes an application data-intensive

- Data is the main challenge (the dominating factor)
  - Data **size**
  - **Complexity**
  - **Uncertainty** (speed at which data is changing)

Data size

- To give you a sense of possible data sizes

SI-prefix	Name	Scale	Status (2011)
k kilo	thousand	10^3	Count on fingers
M mega	million	10^6	Trivial
G giga	billion	10^9	Small
T Tera	trillion	10^{12}	Real
P Peta	quadrillion	10^{15}	Challenging
	(multi-PB)	10^{16-17}	Possible
E exa	quintillion	10^{18}	Aspirational
Z zetta	sextillion	10^{21}	Wacko
Y yotta	septillion	10^{24}	Science Fiction

From an orginal table by Stuart Feldman, Google

Challenging = Just about feasible for Google …
Far too easy to say "peta" and "exa" …

Data-intensive examples

GB = Sorted on Google and Bing
BG = Sorted on Bing and Google

1) Searching the WWW

- As of May 2022, the estimated number of Web pages indexed by Google is circa 60 billion.
  - Almost 70 petabytes (PBs) of data in only one Google BigTable
- To manage such a huge amount of data (storage & searching)
  - Google built a custom file system and indexing methods
  - Running in **distributed deployments (computing clusters)** consisting of thousands of machines

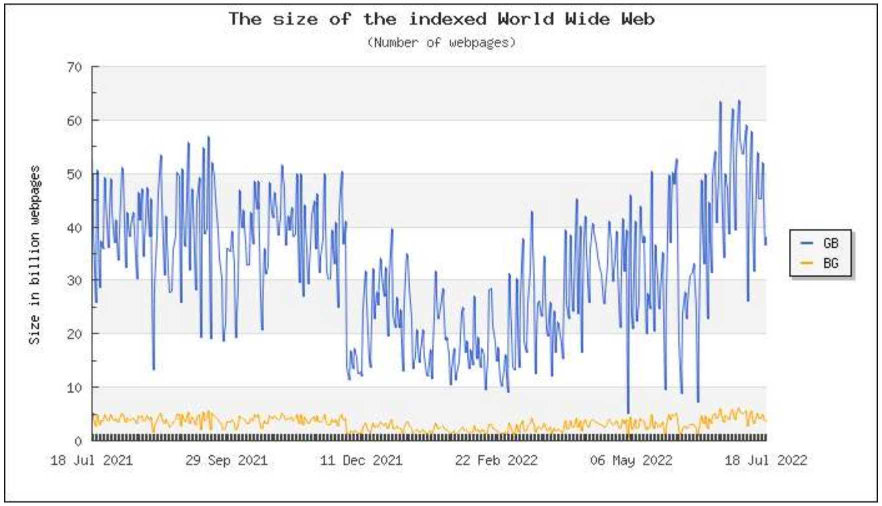


Image source

Data-intensive examples (cont.)

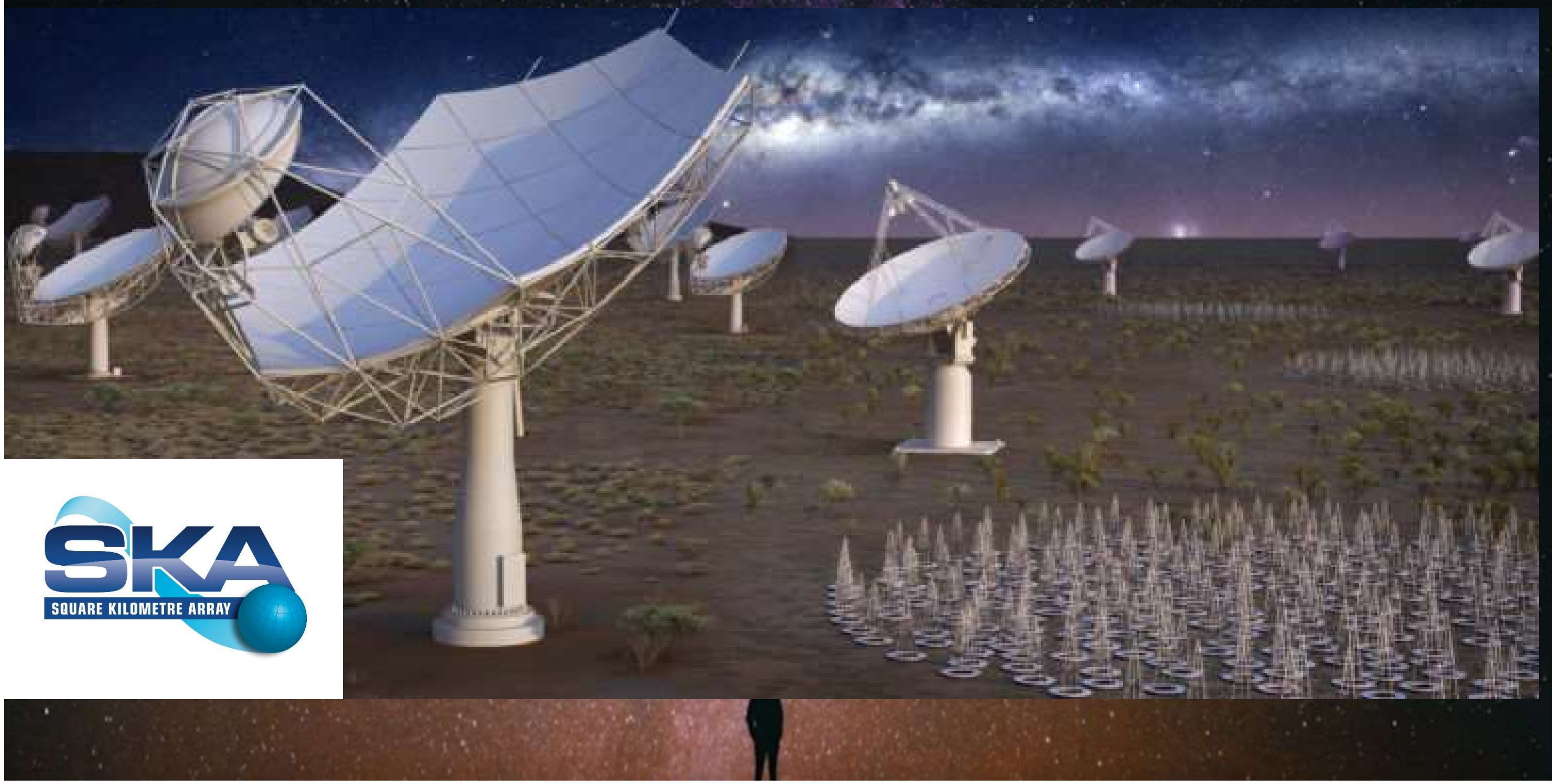
2) Online applications

- Online service providers manage and deliver big data to billions of users worldwide
  - YouTube serves more than 1 billion page views daily
    - Several petabytes
  - Netflix stores several petabytes of data on Amazon’s EC2
  - eBay multi-petabyte (users & event logs data)

3) Other businesses (telecommunication & banks)

- AT&T
  - Multi-petabytes of network daily data

The BIGGEST ever



Scientific data are the biggest ever

- Phase 1– representing approximately 10% of the whole Square Kilometer Array (SKA) Telescope – will generate around **300 PB** (petabytes) of data products every year
- This is ten times more than today's biggest science experiments

- **From tutorial titled: "Solving astrophysics mysteries with big data"**

By : A/Melanie Johnston-Hollitt, Board of Directors, New Zealand

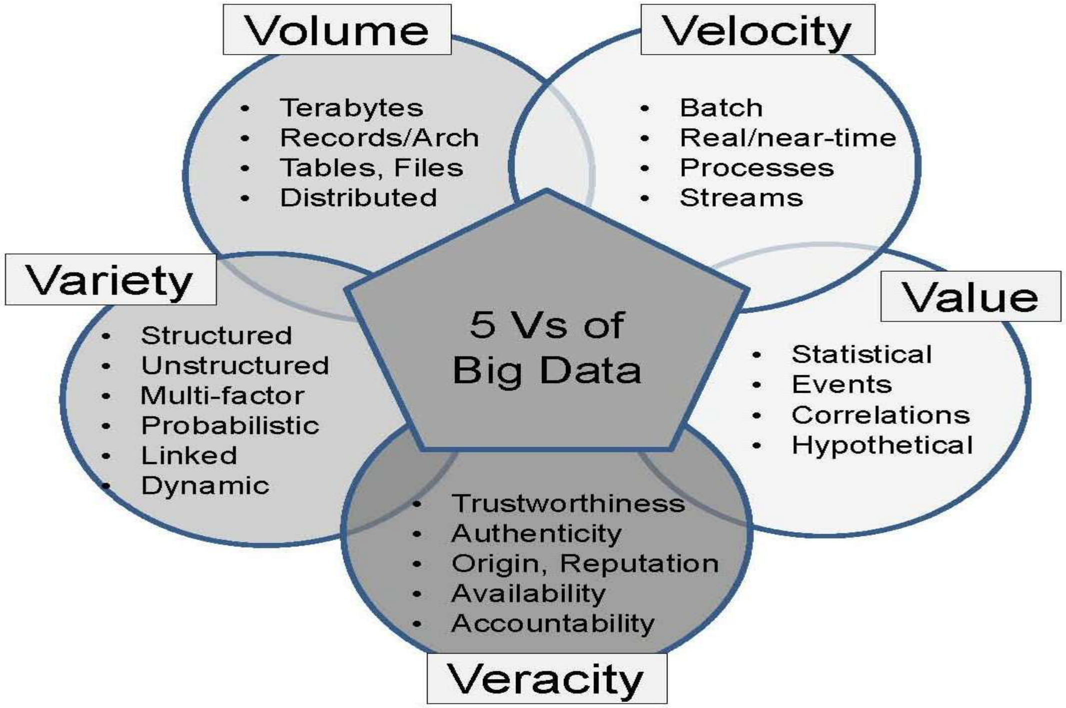
Big Data & more

Information systems require a **quality-aware vision that can organize the whole data lifecycle**

5 V's for new data processing
and
novel data treatment

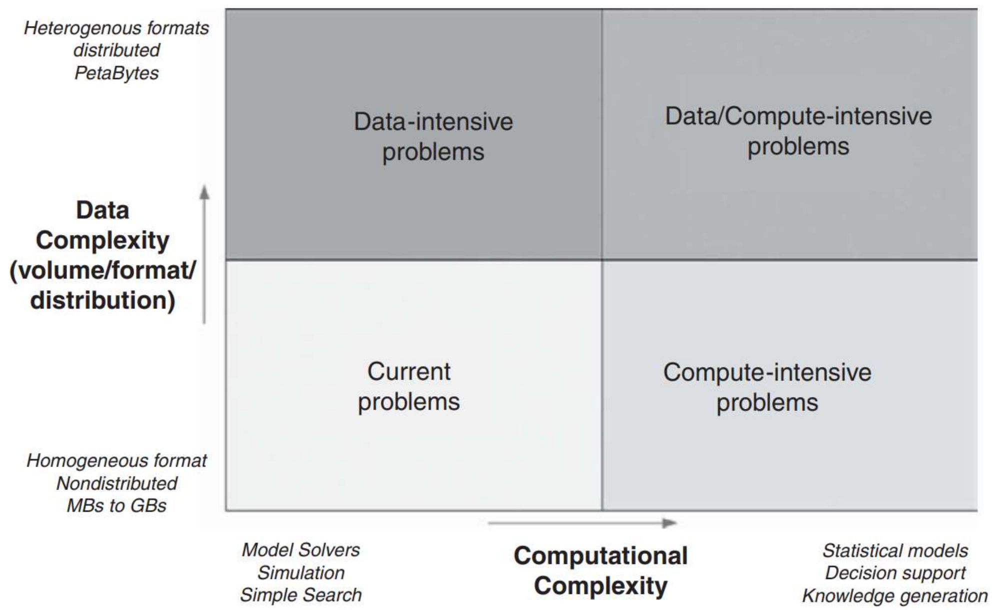
- **Volume of Data**
- **Variety of Data**
- **Velocity**
- **Value**
- **Veracity**

6 V's also Data Dynamicity
- **Variability**



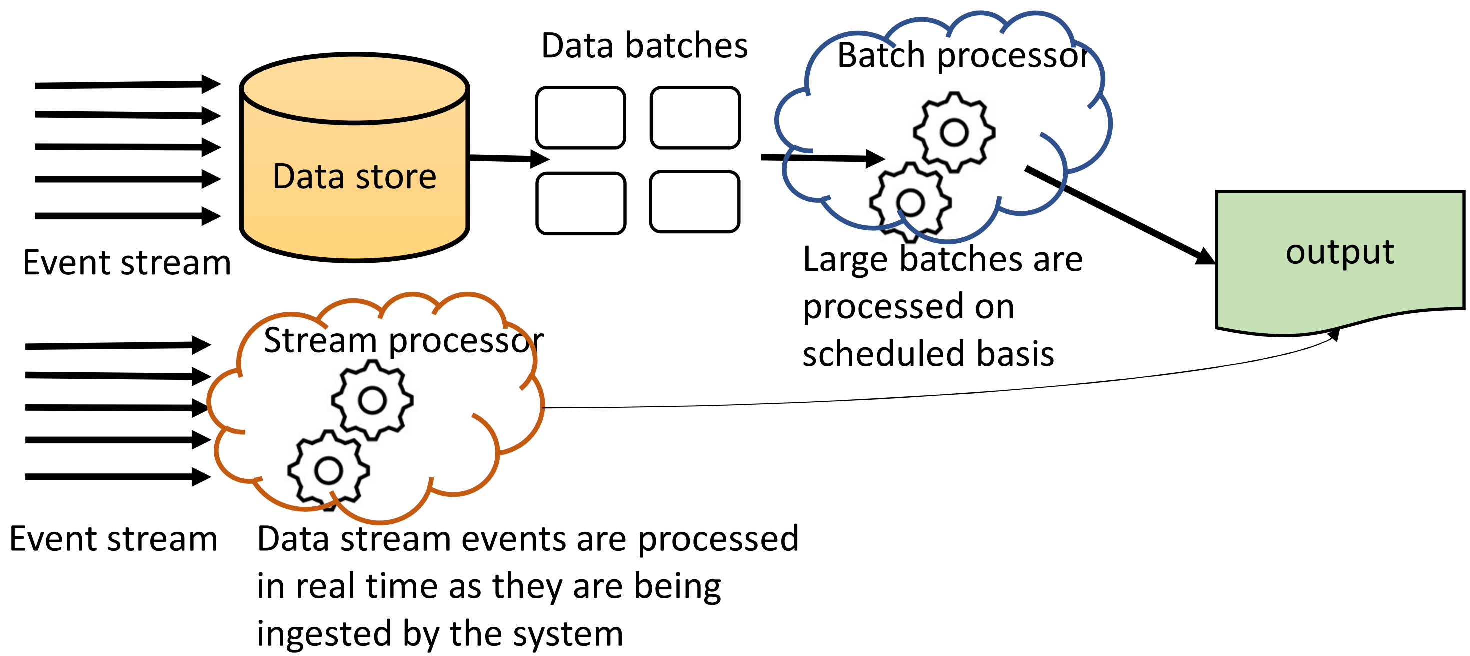
Data-intensive domain

- To make it clear the distinction of data-intensive from other domains
- Characteristics of data-intensive applications
  - Manage **multi-petabytes of data**
  - Distributed data coming from **heterogeneous sources** (requires fusion)
  - Amenable to straightforward **parallelization**
- Challenges in distributed systems include
  - **Data management**
  - **Fusion techniques**
  - **Data distribution & querying**



Building blocks of data-intensive applications

- Common building blocks include:
  - Data storage (***database***)
  - Keeping the output of expensive operations (***caching***)
  - Appropriately searching & filtering data (***indexing***)
  - Processing data on-the-fly (***stream processing***)
    - Unbounded stream of data instead of a batch of data points
  - Crunching huge amount of static data (***batch processing***)
    - Fixed pool of data that we will process to get a result



Challenges

- Several tools to choose from for various applications with varying requirements
  - **Indexing**, caching , **batch** **&** **stream** **processing** may differ significantly across different frameworks
  - Is single tool enough for satisfying the application requirements
  - Do we need to combine functionalities from various tools
- How can we build efficient data-intensive applications?
- What tools have in common, what distinguishes a tool from others for a specific data-intensive workload
- What design decisions should be considered when building a specific data-intensive application

Challenge: single tool does not fit all!

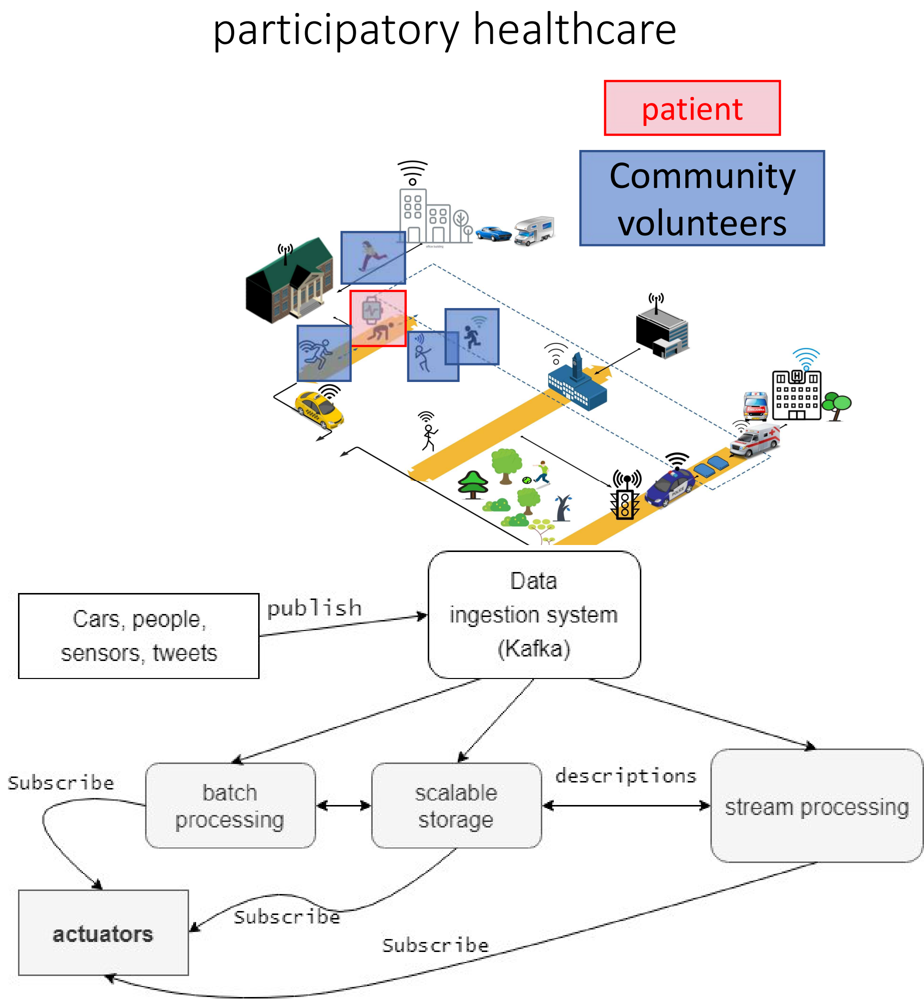
- Data-intensive applications are characterized by having wide-ranging demanding requirements that there is no such thing like " single tool fits all"
  - No single tool can meet the storage & processing requirements altogether
- One size does not fit all
  - Different application workloads may require purpose-built systems
  - Design tradeoffs decisions → performance tradeoffs
- **Divide & conquer**
  - **Divide** the workload into tasks
  - **Run** each task on a single tool
  - **Stitch** single tools together to accomplish the big task

Example data-intensive Application Scenario

- A mixed-workload scenario requiring at least
  - **Traffic Light Controller**. Actuator decides to change lights consistently for ambulance to pass
  - **Smart Real-time Pathfinder**. Interactive navigation map for ambulances and other vehicles
  - **Real-time Community Detector**. Identify volunteers' communities in the surroundings of the patient

➢ Combining tools to provide the service

- ➢ Creating a special-purpose data-intensive system by **stitching** together various general-purpose tools
  - ➢ Batch & stream processing, scalable storage, and stream data ingestion
  - ➢ What **guarantees** we can assure by this combination**?**



Requirement for services

In distributed systems, while services must be correctly provided

A critical goal is the **Quality of Service (QoS)**, in the sense of **provisioning with some parameters** and **respecting some requirements**

The **QoS** has many **different meanings**, because it is a very **general quality indicator**

It can stress **response time**, **security**, **correctness**, **availability**, **confidence**, **user satisfaction**, …

QoS goals (conflicting?) in the **Old** and the **New World**

- **Old world:** typically, main goals **reliability** and **enforced consistency**
- **New world: scalability and availability** matters **most of all**

Focus on **extremely rapid response times:** Amazon estimates that **each millisecond** of delay has a measurable impact on sales!

Common desired guarantees

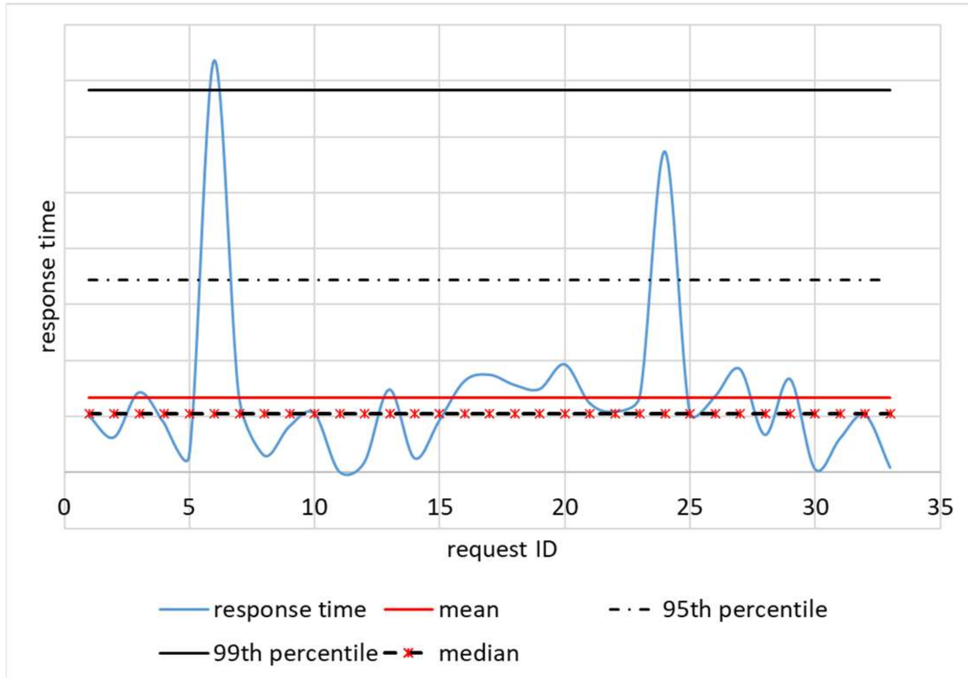
- Reliability
  - The performance of the system is predictable in face of data load and volume
  - Avoiding failures, such that the system continues providing the expected service
- **Scalability**
  - Coping up with data loads. As **data size grows**, **complexity and speed**, system should adapt appropriately
    - Hardware scalability. **Overprovisioning** resources, or
    - **Approximate Query Processing (AQP)**. Data reduction techniques.
- Maintainability
  - The system should be adaptable in face of emerging scenarios

Scalability

- **Load** can be described in several ways
  - Number of requests per second for a specific service
  - Ratio of reads to writes
  - Number of users active simultaneously
- Design choices are affected by the **average loads**
- Performance
  - How the system is behaving when **load** changes
  - If we need the to maintain the performance, what choice should we make
    - **Hardware scalability** or **AQP**
- **Measurements**
  - ***Throughput***
    - Number of records that can be processed per second
    - Total time to run on a given data of specific size

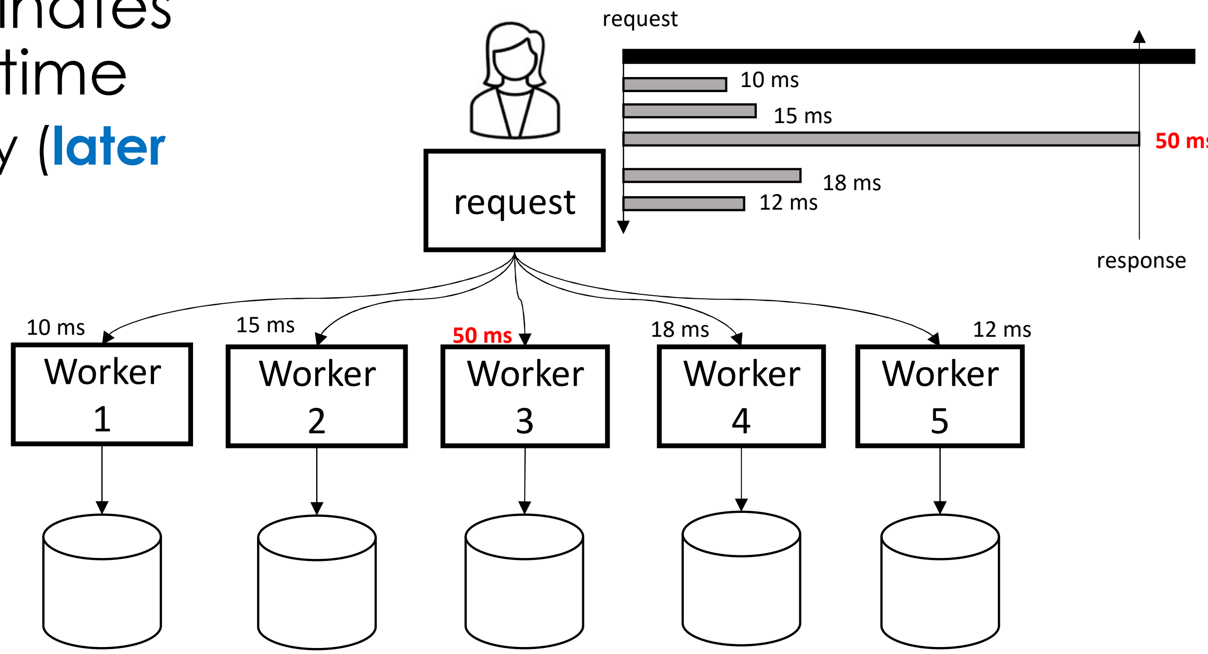
Response time

- ***Response time***
  - The time between sending request and receiving response
  - Actual request processing time (i.e., ***service time***) plus network & queueing **delays**
  - May differ for different requests, need to be measured for each workload
  - We normally report the ***average* response time**, **percentiles** , or **median** (**50th percentile**)
    - Mean does not show the **outliers**
    - **Percentiles** are preferred
    - Sorting response times in decreasing order, the ***median*** is the halfway point
- Specified in a service level objective (SLO) or service level agreement (SLA)
  - e.g., median response time less than 100 MS, 95th percentile under 1 second



How response time is affected in parallel computing systems

- The slowest call dominates the overall response time
  - Load balancing is key (**later discussion**)



Coping up with load fluctuations

1) Scaling
- Up (**vertical**). Deploying more powerful single beefed-up servers
- Out (**horizontal**, **shared-nothing architectures**). Distributing the load to multiple machines

- Design decision
  - What kinds of operations are common
    - **Stateless** (parallelization is straightforward), **stateful** (additional complexities are facing distributed architectures)
- No single architecture is the best
  - Reading & writing **loads** (access patterns),
  - Data **complexity**
  - **Response time** requirements

2) Approximate Query Processing (AQP)
- Reduce data size with techniques that guarantee QoS (accuracy, response time, etc.,) to some extent

Coping up with load fluctuations (cont.)

- Vertical Scaling
  - Increasing **single server** capacity
    - More powerful CPU, more RAM, more storage space
    - Could easily be hindered by limitations in technology
- Horizontal Scaling
  - Dividing data and load to **multiple servers**
  - Each machine handles **partial** set of the data workload, providing much better efficiency than a single high-capacity server
  - Increased infrastructure complexity and maintenance

Behind the Woods: support for…

To **provide QoS** distributed systems have to support some coverage of **properties** and **functions**

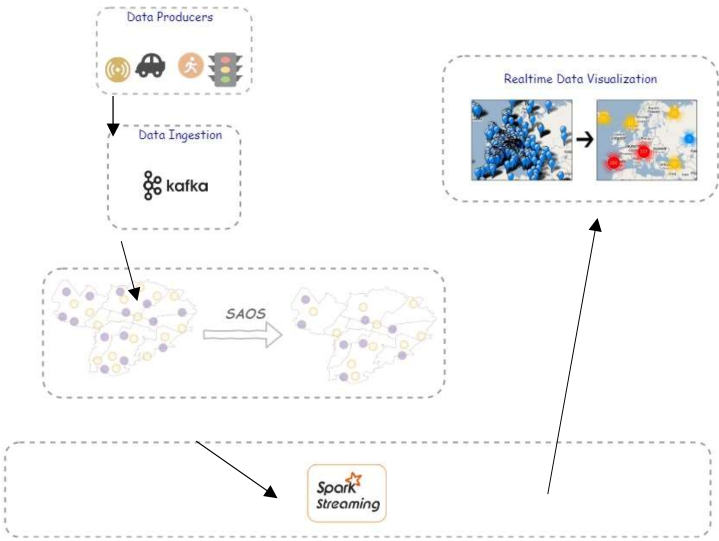
- **Replication**: usage of multiple copies of resources
- **Grouping**: keeping together different copies and behavior
- **Simplified delivery**: new tools and technologies to fasten development & deployment of complex applications
- **Automated management**: infrastructures taking care of management burden with minimal human intervention
- **Batch data processing:** storage/processing of massive amounts of data, such as for Google Web indexing
- **Streaming data**: dealing with information series coming from a set of grouped info, such as a video, sensors, etc.

Anatomy of distributed model solutions for data-intensive problems

Processing pipelines & stages

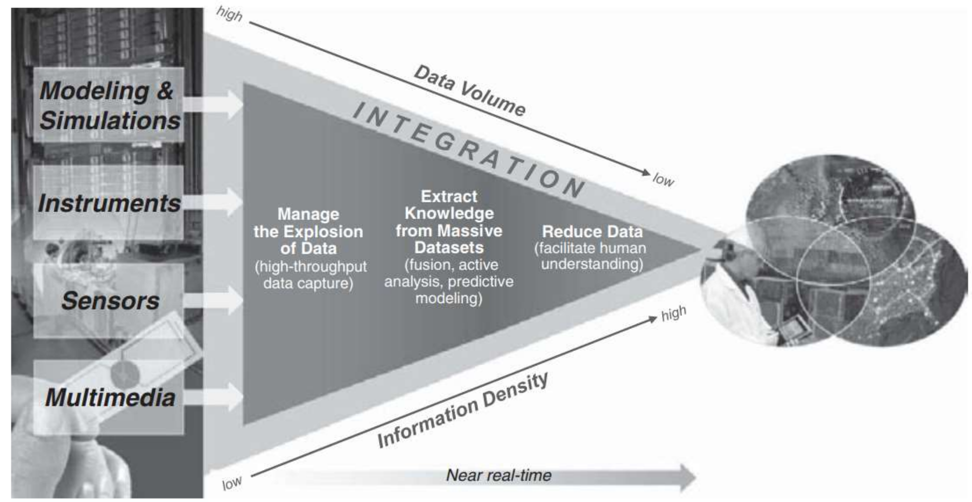
Typical architecture of data-intensive applications

- Common stages
  - Data **collection**
    - Bringing data from sources (probably heterogeneous) to data-intensive applications
  - Data **transformation**
    - **Reduction. transformation** of data into a simplified form,
      which is more amenable to downstream processing
    - Normally single-pass for scalability
    - Sampling, data pruning, etc.,
  - Data **storage**
  - **Analysis**
    - Discover patterns in the data
  - and **Visualization**
    - Visualizing the output of data intensive applications, helping the user make informative decisions



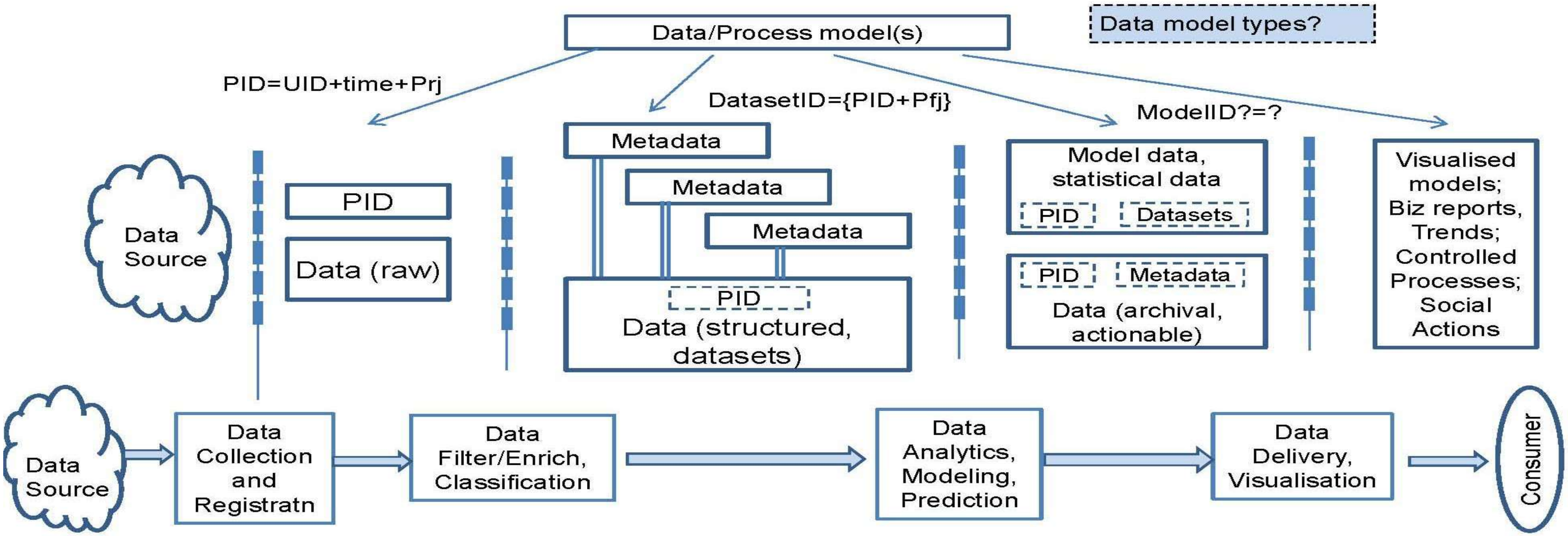
Data-intensive processing pipeline

- Scientific data-intensive problems need processing pipelines
  - **Collecting** the data
  - Reducing it size and performing other **transformations** (sampling, summarizations, aggregations, indexing, etc.,)
  - Applying advanced specialized algorithms to **analyze** & **process** the midway data, resulting in human-readable knowledge
- Normally requires **data parallelism** (**distributed computing** clusters or HPC)
- User **visualize** the data in informative ways, investigating and validating the outputs



Data Transformation Model

The main workflow is to move data from source to sink via a pipeline easy to map and describe

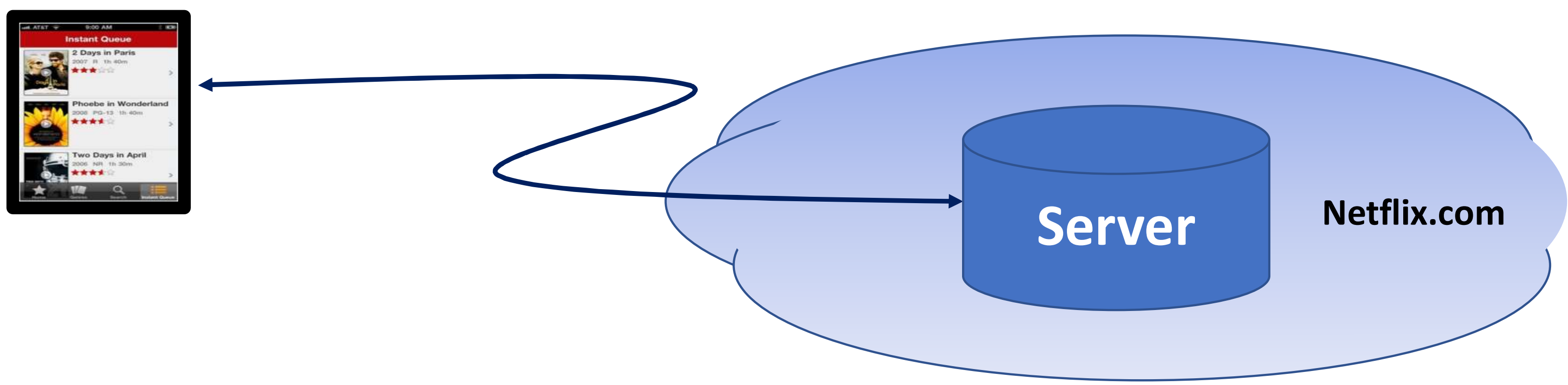


New support architectures with novel design principles based on quality-aware services

An example: Netflix

Personal service to **play movies on demand**

User Perspective



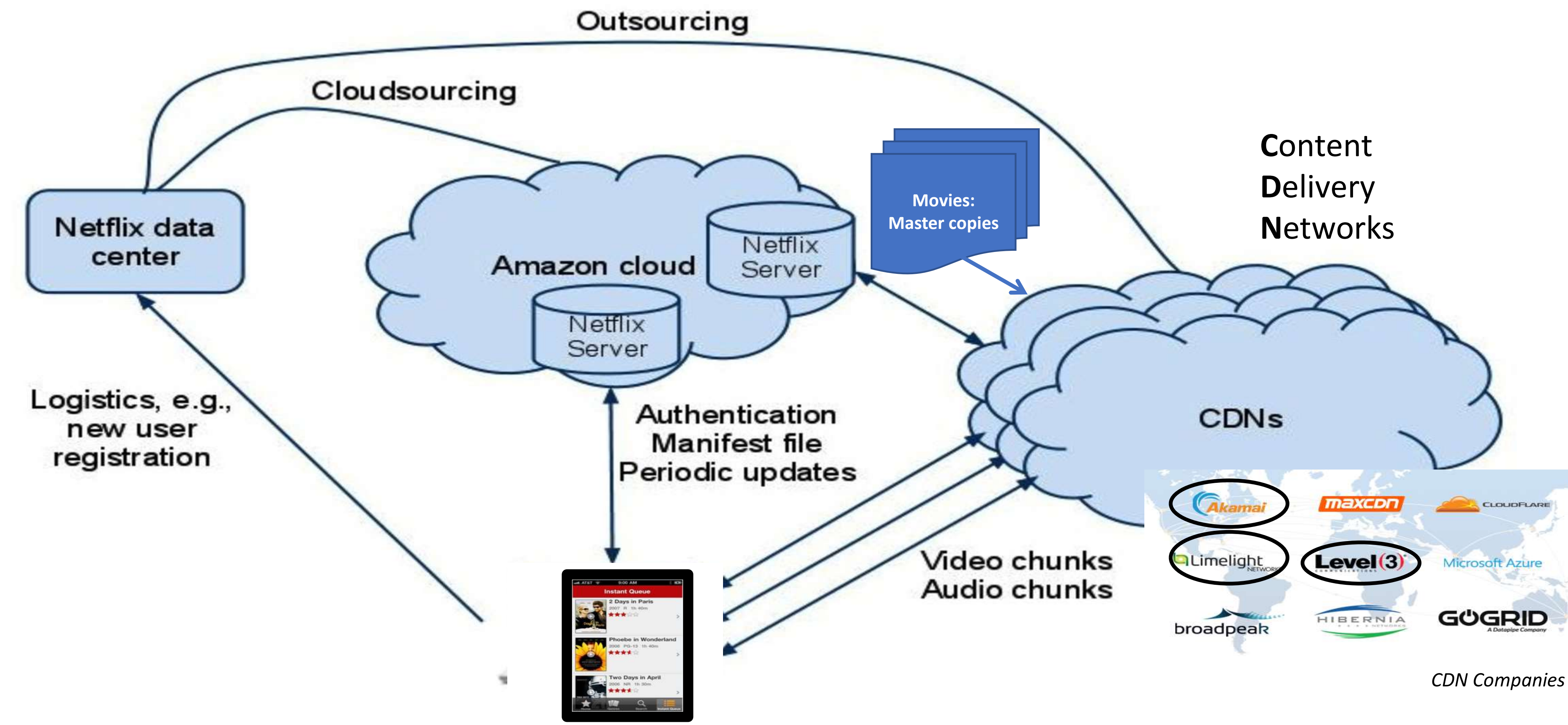
Simple design?

Netflix owns the data center and content distribution infrastructure

BUT, in reality….

Netflix owns **neither** a data center **nor** a distribution infrastructure

Netflix: the complex picture



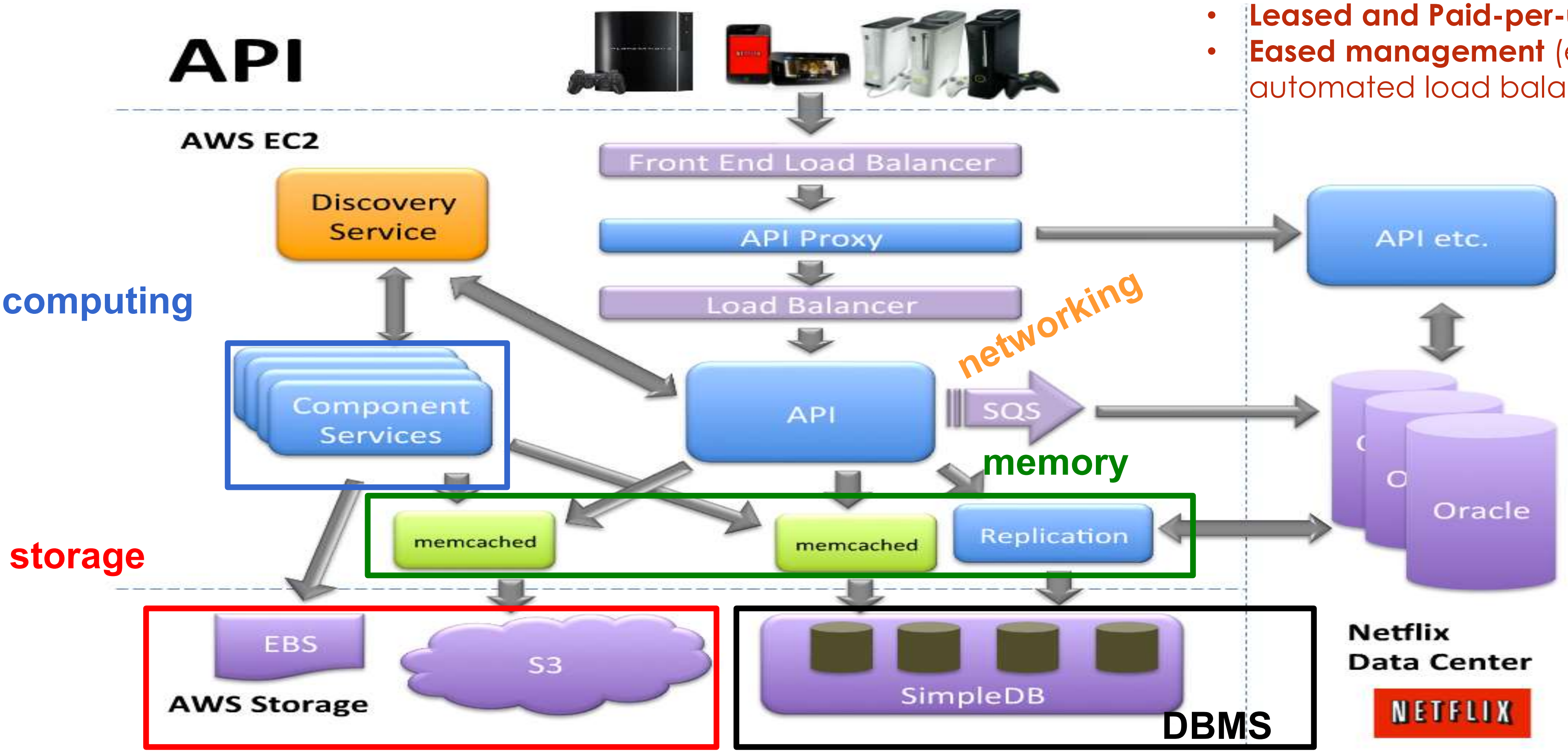
CDN Companies

V.K. Adhikari *et al.*, “Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery“, *IEEE INFOCOM*, 2012.

Netflix & AWS EC2 in a Nutshell

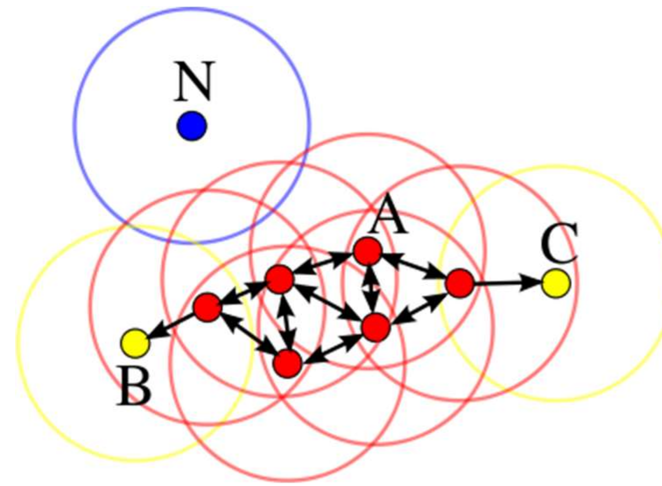
Amazon Web Services (WS) **Elastic Cloud Computing** (EC2) resources

- **Leased and Paid-per-use**
- **Eased management** (e.g., automated load balancing)



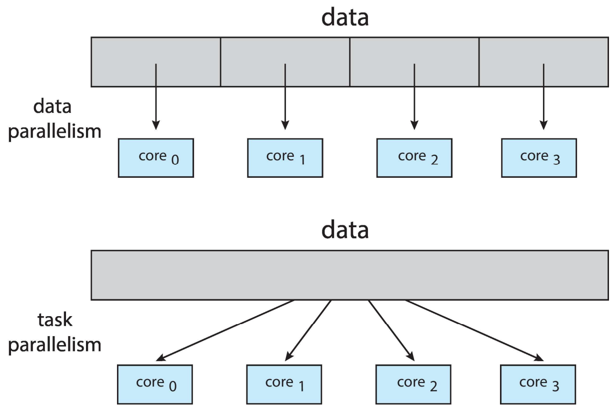
Example processing & analysis in data intensive applications

- **Clustering** (e.g., DBSCAN-MR, for DBSCAN MapReduce)
    - **Grouping** data into **clusters**, such that same-cluster items are more similar than items in other clusters
    - Similarity is a **domain-specific** measurement
        - e.g., spatial applications, nearby spatial objects in real geometries form clusters
- **Search** (proximity search)
    - Finding objects with specific attribute values



Parallelism is essential

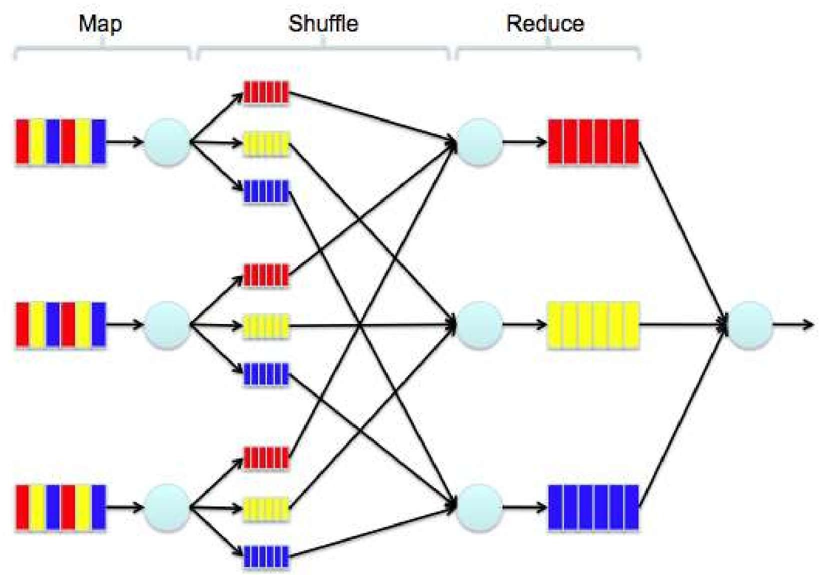
- Reduced data size does not guarantee the ability of efficient processing
  - Data **parallelism** is often involved, using **computing clusters of machines**
- Data parallelism simply implies partitioning data to multiple portions (**MapReduce** is the baseline)
  - **Process** each portion independently & concurrently across multiple computing machines in a cluster
  - **Combine** the sub results to produce the output
- Google & Microsoft multi-petabyte data centers each might contain 100K low-cost commodity hardware nodes



Example programming model: MapReduce

Programming **paradigm** for computing and aggregating **large amounts of data**

- Mainly abstractions **for data-intensive** applications to exploit data distributed in computing clusters
- Distributes **data & processing** to computing nodes of a cluster
  - Then **process the data locally** at each computing node independently & **in parallel**
  - Then, it **combines** the local results to form the output



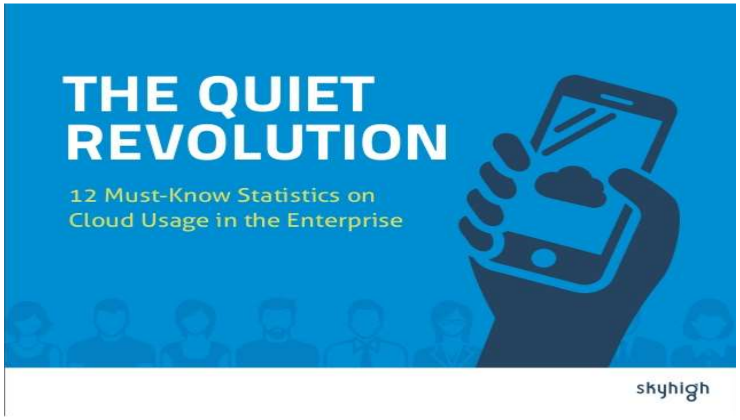
Supporting infrastructures & enabling technologies for data intensive applications

Clusters in public Cloud, private Cloud, virtual machines, and virtualization of clusters

Cloud Revolution…

Cloud is a buzzword to be used in advertising and it is sometimes depicted as a revolution

The are many books about Cloud as a revolutionary technology



In general terms, there is **no *solution of continuity*** both under an **organization** and a **technical perspective**

Clouds are Cheaper…
and Winning…

Range in size from “edge” facilities to **megascale**

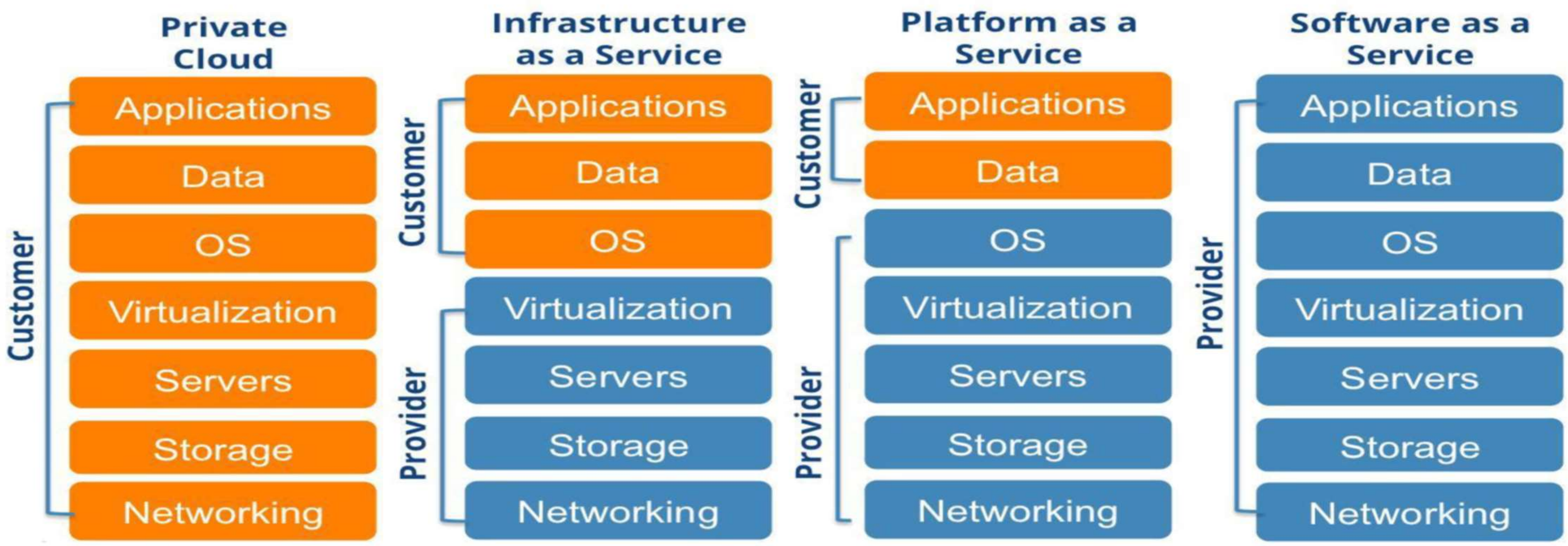
Scale economies

Approximate **costs for a small size center** (1K servers) and a **larger**, 50K server center

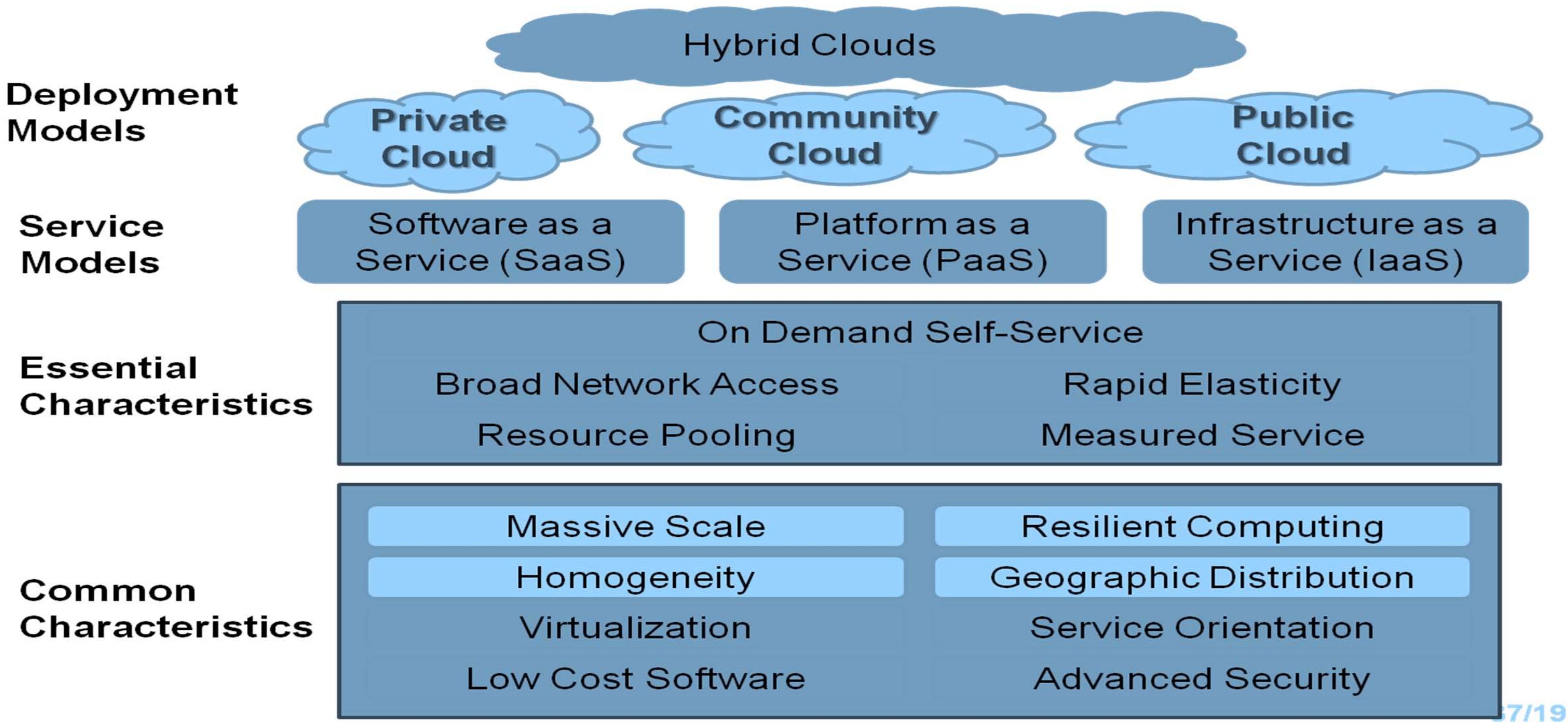
Technology	Cost in small-sized Data Center	Cost in Large Data Center	Cloud Advantage
Network	$95 per Mbps/ month	$13 per Mbps/ month	7.1
Storage	$2.20 per GB/ month	$0.40 per GB/ month	5.7
Administration	~140 servers/ Administrator	>1000 Servers/ Administrator	7.1

Each data center is
11.5 times
the size of a football field

Cloud architectural comparison



The NIST Cloud Definition Framework



***N**ational **I**nstitute of **S**tandard and **T**echnology* **NIST**

What is a Cloud

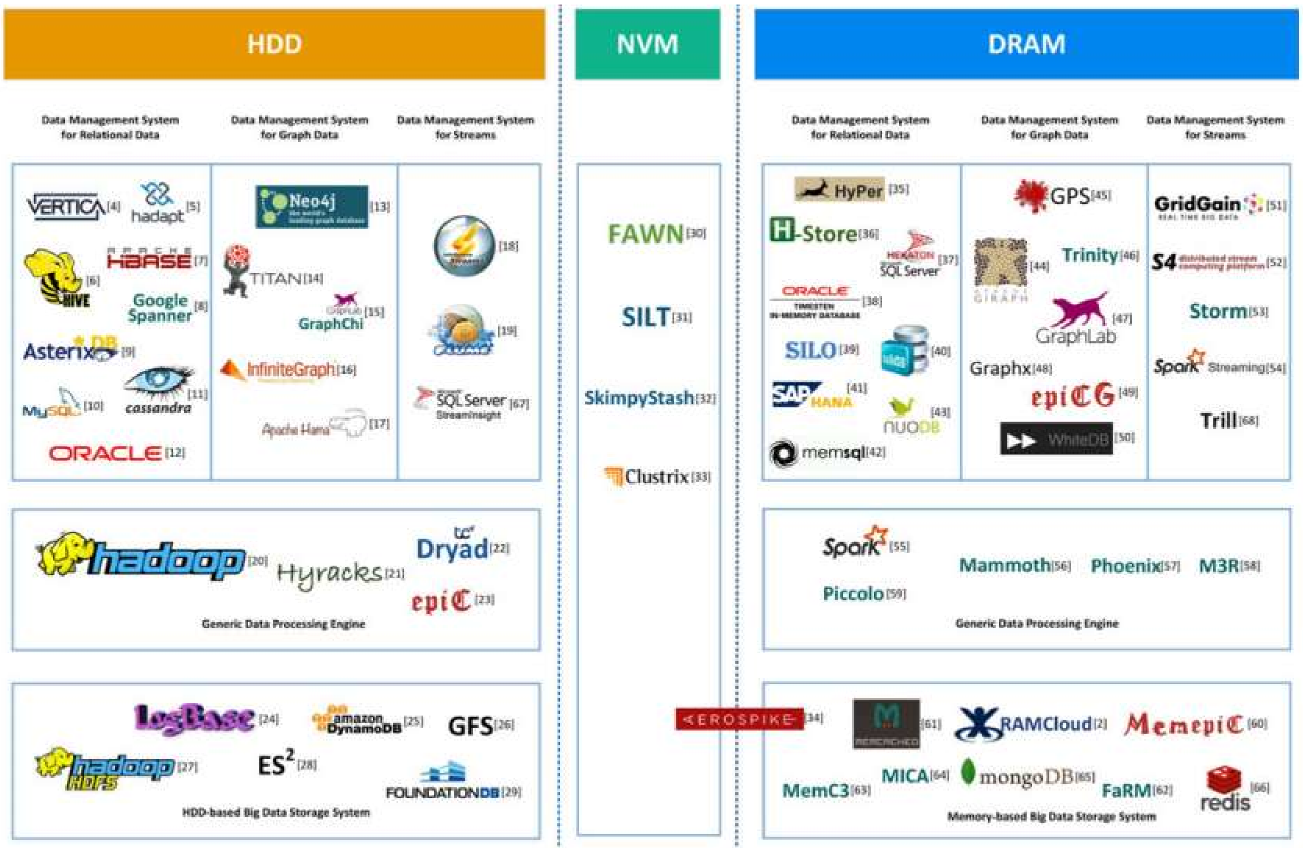
One Cloud is capable of providing IT resources 'as a Service'

One Cloud is an **IT service** delivered to users that have:

- A **user interface** that makes the infrastructure underlying the service transparent to the user
- **Massive scalability**
- **Service-oriented management** architecture
- Reduced **incremental management costs** when additional IT resources are added

- Services are available via **Web or REST interfaces**
- Other **user requirements** possible based on **geographical preferences, localization constraints, …**

Partial landscape of Cloud-based systems



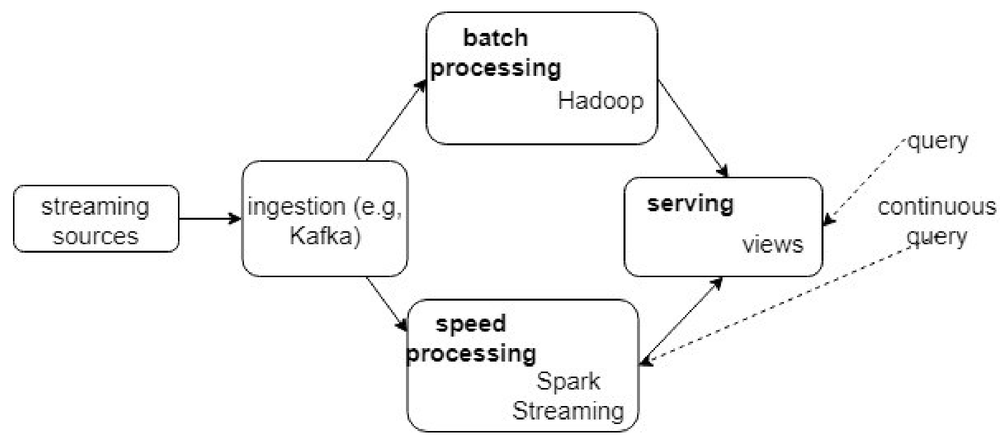
H. Zhang, G. Chen, B. C. Ooi, K. L. Tan and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," in IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 7, pp. 1920-1948, July 1 2015.

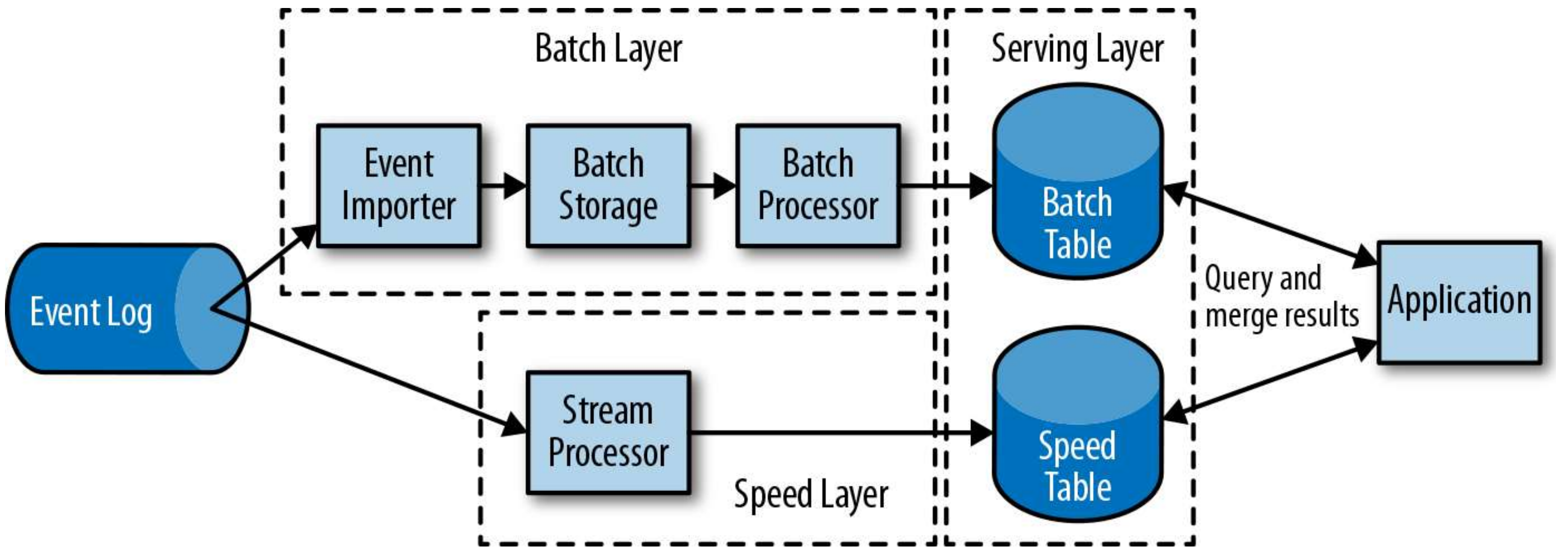
Distributed architectures for big data management

Reference architectures for storage and processing of big data, such as Lambda architecture

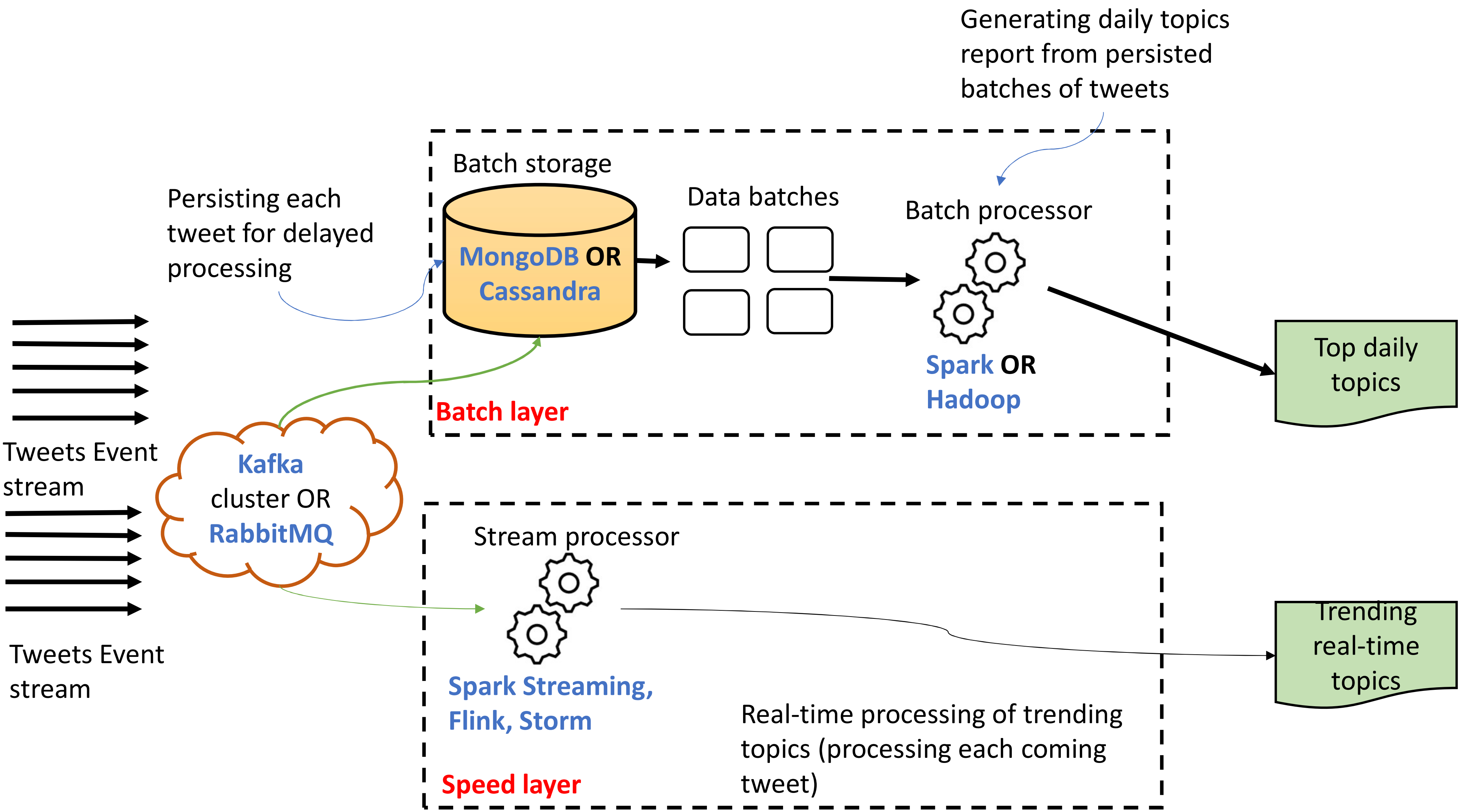
Lambda Architecture

- Challenges associated with managing mixed streaming big data workloads have motivated the emergence of novel dynamic architectural patterns such as the **Lambda architecture**
- The Lambda architecture employs real-time **stream processing** for timely approximate results and **batch processing** for delayed accurate results





Batch Layer
Serving Layer
Event Importer
Batch Storage
Batch Processor
Batch Table
Event Log
Query and merge results
Application
Stream Processor
Speed Layer
Speed Table



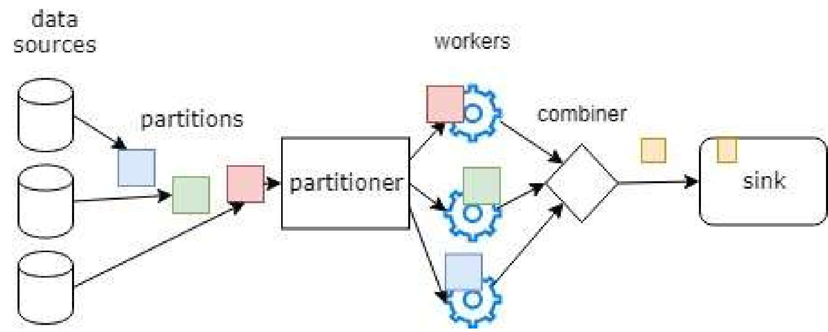
Generating daily topics report from persisted batches of tweets
Batch storage
Persisting each tweet for delayed processing
MongoDB OR Cassandra
Data batches
Batch processor
Spark OR Hadoop
Top daily topics
Batch layer
Tweets Event stream
Kafka cluster OR RabbitMQ
Stream processor
Trending real-time topics
Spark Streaming, Flink, Storm
Real-time processing of trending topics (processing each coming tweet)
Tweets Event stream
Speed layer

Key tasks in distributed management of big data

Partitioning, rebalancing & serialization

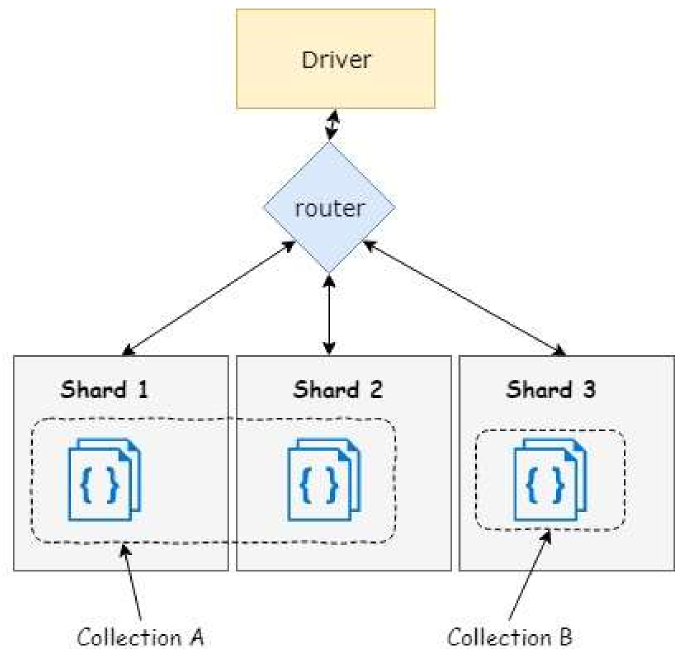
Data partitioning

- Distributing **partitions** of data over several processing (i.e., **worker nodes**) or **storage** elements in a **parallel** computing environment (i.e., Cloud)
  - Processing is accomplished simultaneously by each processor instance on the corresponding partition
- One of the reasons to distribute data loads to multiple machines is the desire for **scalability**
  - **Read & write loads** grow significantly
  - Large datasets & **query loads** are distributed



Data partitioning (cont.)

- Known as ***sharding*** *in* MongoDB, Elasticsearch, and SolrCloud, ***region*** in HBase, a ***tablet*** in Bigtable, a ***vnode*** in Cassandra, and a ***vBucket*** in Couchbase
- Shared-nothing architectures (**scaling out** or **horizontal** scaling) are preferred over shared-memory counterparts for **data-intensive applications**
  - A single machine (or **virtual** machine) running the database software is known as a ***node***
  - Each **node** uses its CPUs, RAM, and disks independently



Sharding in MongoDB

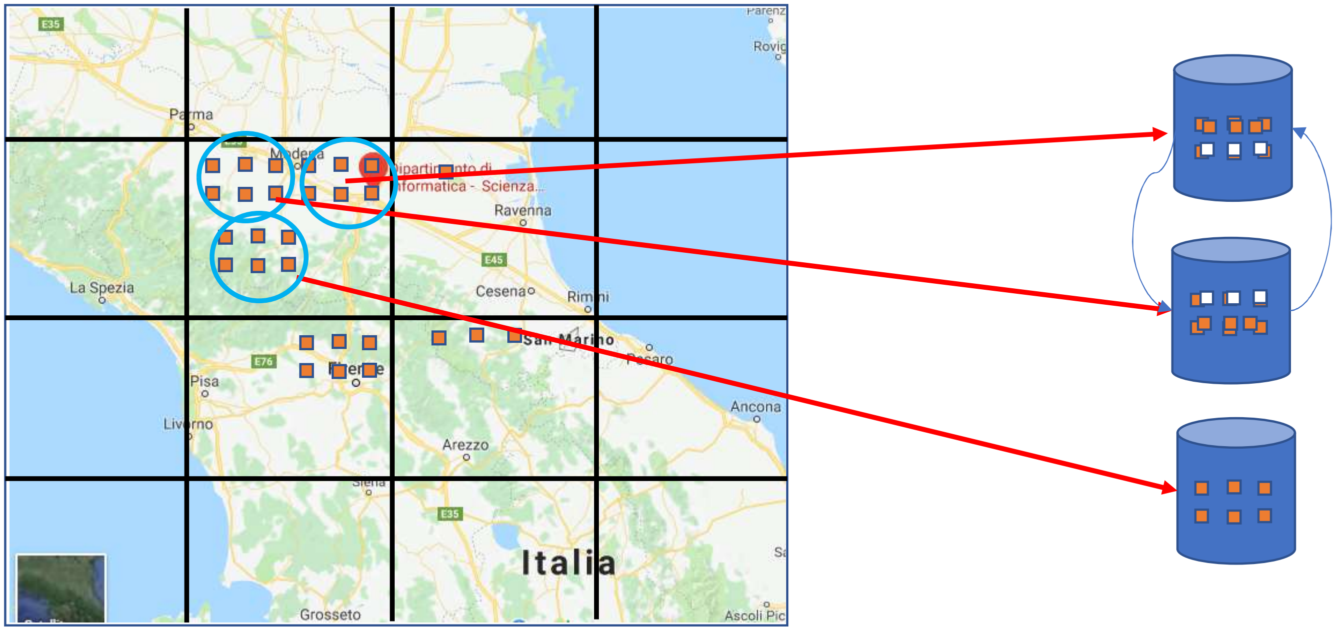
Load balancing is essential

- The main goal of partitioning is to evenly **distribute** the **data & query loads** across **parallelly** connected nodes
  - This is known as **load balancing**
- If data is distributed **evenly**, then in a perfect setting, it means sending the same amount of data to each node
  - In theory, 100 **nodes** can handle 100 times as much data as a single node can handle, also having a collective **read/write throughput** that is 100 times of that of a single node

Load balancing is essential (cont.)

- On the other hand,
  - If data is **unevenly distributed**, then some nodes are **overlooked**, having less data
  - While others having much more data, to the point that they become the **bottleneck** of **storage** & **processing**. Those nodes are typically known as **hotspots**
  - In this case, the benefits of partitioning easily diminish
  - Imagine a worst case where all data load ends up in one partition, while other partitions are will be **idle**

Load balancing (smart city scenario)



In Spark join requires data to reside on the same partition

Is load balancing alone sufficient?!
Only load balancing = shuffling (huge toll) for co-location queries

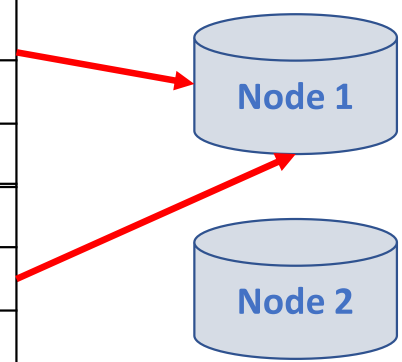
Partitioning approaches

- The simplest is **randomly** & **evenly** assigning records to nodes
  - Achieves **load balancing**, however,
  - Read queries need **brute force full scan** to find specific records
    - We have no knowledge where specific records reside
- **Partitioning by keys**
  - **Key range partitioning**
    - Assign values within a specific key range to same partitions
    - If data is **skewed** (few keys have more data than others), choose the range wisely in such a way that you also preserve (to some extent) the **load balancing** property
    - Sorting keys in each partition speeds up the **range queries**
    - Bigtable, Hbase, and MongoDB

Key range partitioning challenges

Time unit range readings

Sensor#	date	timestamp	PM10
1	**2022-01-01**	**20220101-000000**	**5.3**
1	2022-01-01	20220101-000010	4.2
1	2022-01-01	20220101-000020	2.2
2	**2022-01-01**	**20220101-000000**	**4.7**
2	2022-01-01	20220101-000010	7.2
2	2022-01-01	20220101-000020	9.1



Partition key

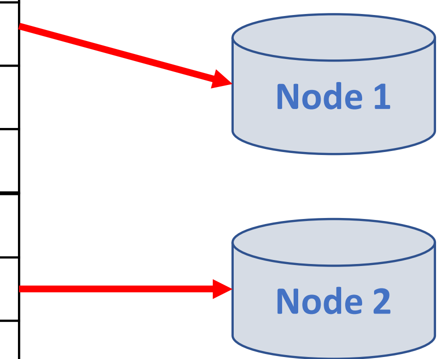
Since the key is a **timestamp**, **partitions** correspond to time ranges, which leads to **overloading** specific partitions by writes (on-the-fly writes as data coming from sensors) → leads to **hotspots**

Better design – key range partitioning

Sensor #	timestamp	date	PM10
1	**20220101-000000**	**2022-01-01**	**5.3**
1	20220101-000010	2022-01-01	4.2
1	20220101-000020	2022-01-01	2.2
2	**20220101-000000**	**2022-01-01**	**4.7**
2	20220101-000010	2022-01-01	7.2
2	20220101-000020	2022-01-01	9.1

Time unit range readings

Partition key

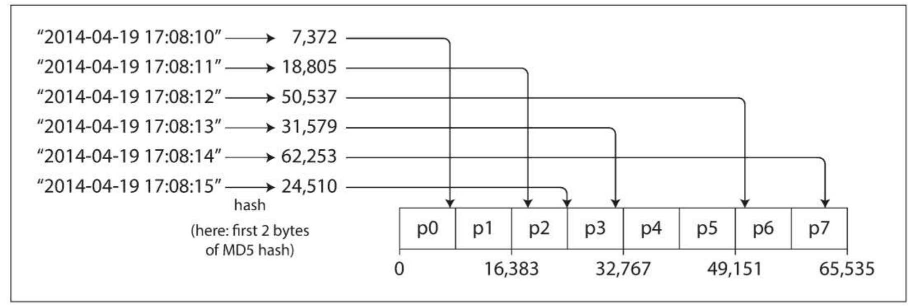


Prefix each timestamp with the sensor ID such that the partitioning is first by sensor ID and then by timestamp – **load balancing** is then achieved (to some extent), assuming that all sensors sending data at regular basis.

Is something else preserved here?

data **co-locality**, a desired property for **proximity scans** → readings from same sensors ends up in same partitions

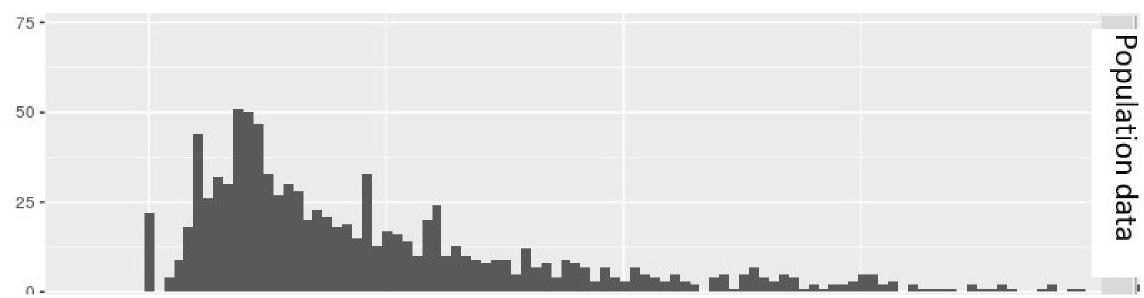
Hash key partitioning



- Avoiding **skewness** & **hotspots** requires other schemes for partitioning data
  - Here where **hash key partitioning** comes in!
  - Using a **hash function** to specify the partition for a specific key
  - Good functions transform **skewed** data to **uniformly** distributed counterpart
    - Cassandra and MongoDB use **MD5**
  - Assign range of hashes to each partition
    - Transform key using the hash function, look up the corresponding partition having a hash range where the hashed key can be assigned and assign it to that partition.
  - Good for **load balancing**,
  - and (depending on the application domain) for data **co-locality**
    - True only for some domains such as **spatial data**, where co-locality can be preserved by encoding schemes such as **geohash** (discussed in **part 3**)
    - However, in general purpose domains, co-locality is typically not preserved by hashing, so it negatively affects range scans (example, MongoDB **range scans all partitions if hash-based sharding is enabled**!)

Data skewness & partitioning challenge

- Some data in specific domains is highly **skewed**
  - **Skewness** is the asymmetry of a distribution of a variable's value around its mean
- Some keys in the data may have more **frequency** than others
  - Hashing in this case does not help **load balancing** as few keys may dominate the distribution, and will be routed to same partitions, turning them into **hotspots**
  - As this is domain-specific problem
    - In most cases, it can not be automatically mitigated at the **system level**
    - It, otherwise, need to be managed at the **application level**
      - More **logistics** handling

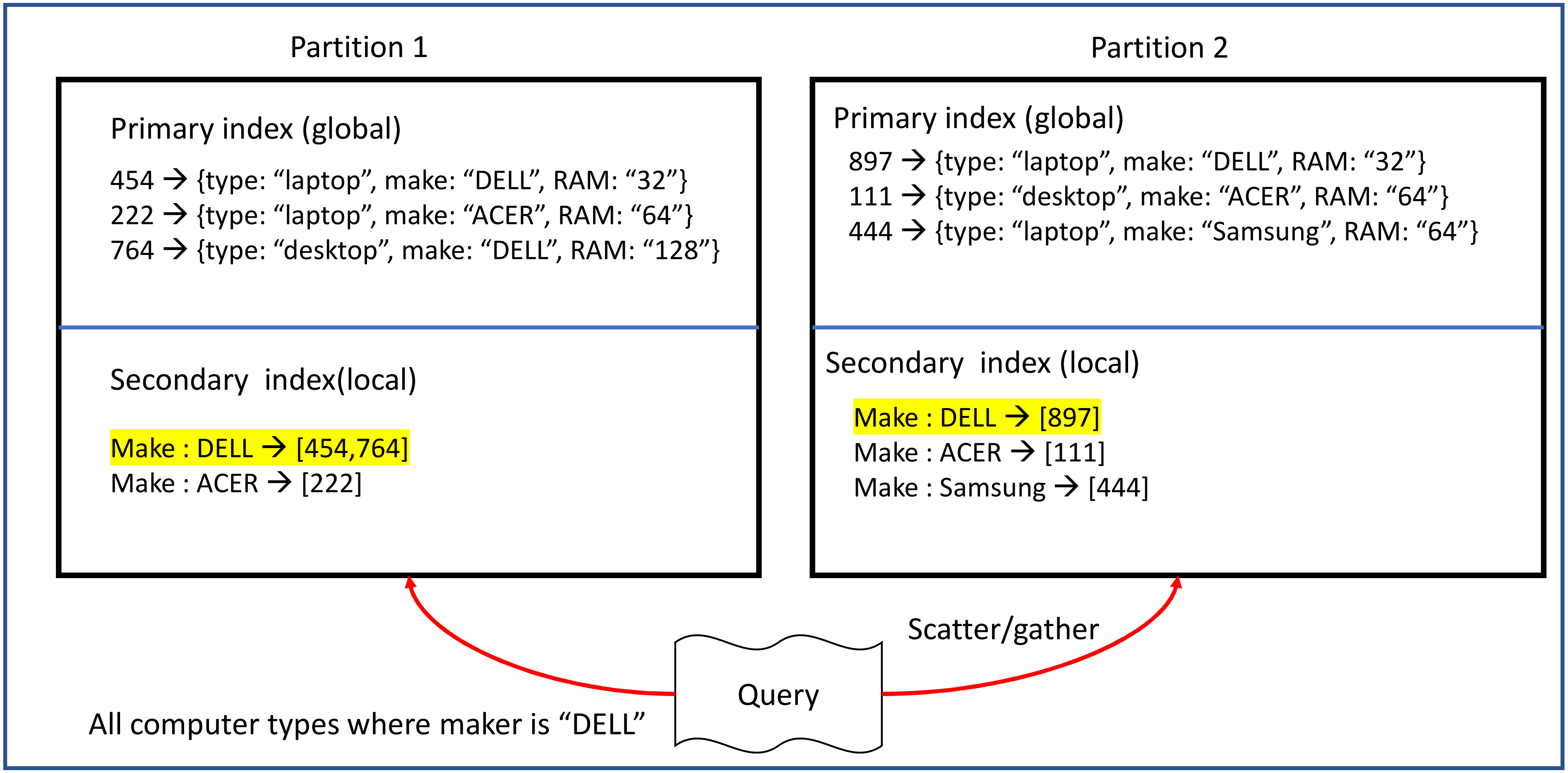


Mobility data. NYC taxicab dataset is highly skewed

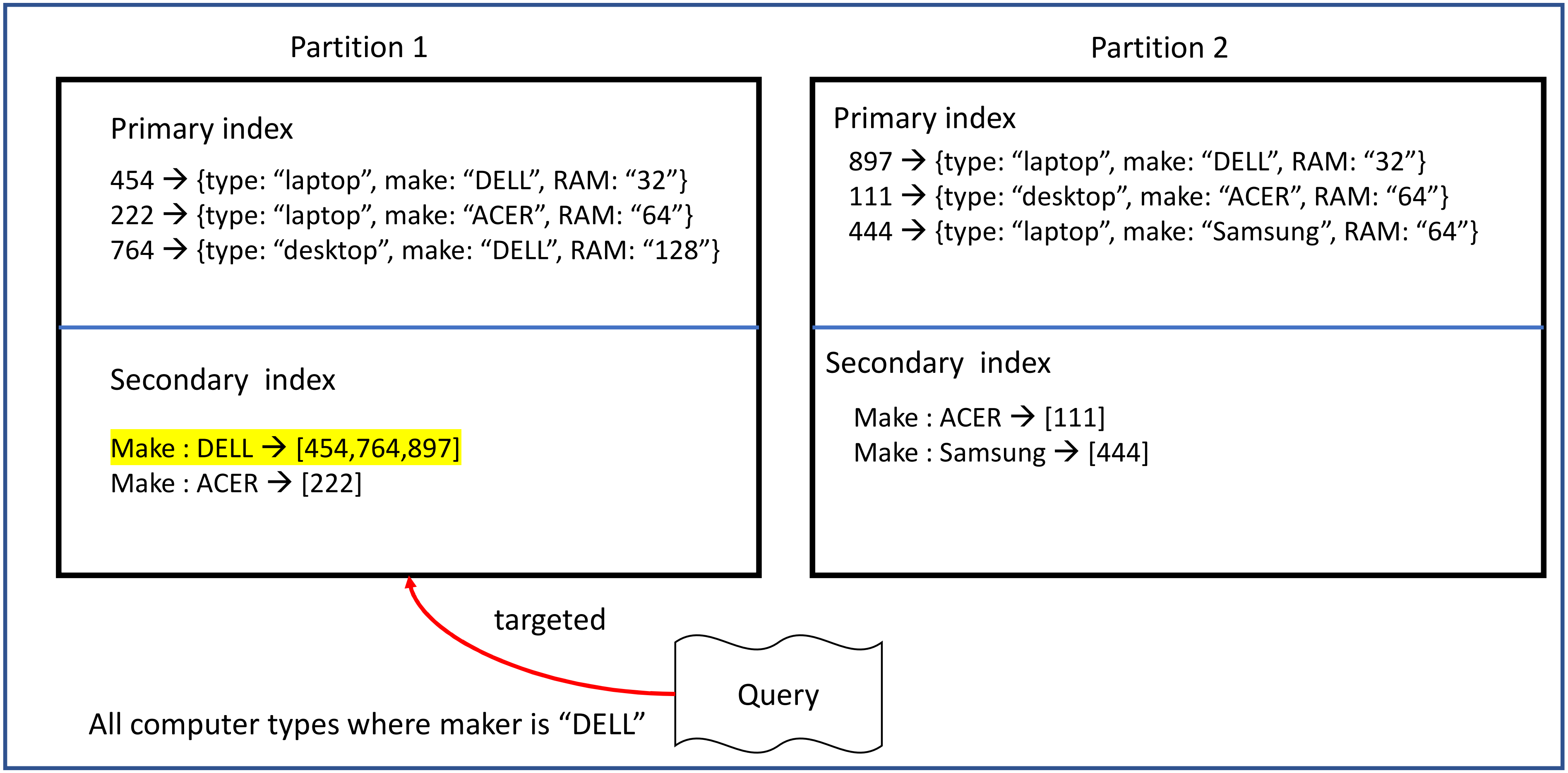
Secondary indexes & partitioning

- Schemes discussed so far work very well for **key/value** data, where data is indexed with a single key
  - For example, the location in mobility data is a sufficient primary index as most spatial queries ask location-driven questions (**proximity**, **range**, **kNN**, spatial **join**, etc.,. To be discussed in **Part 2** of the course)
  - But what if we have a **secondary index**?!
    - Frequent scans search for values of specific attributes, beyond the value of a primary key!
    - We need to take the secondary key into consideration for proper partitioning

Challenge of secondary indexes in partitioning



Possible solution



Rebalancing

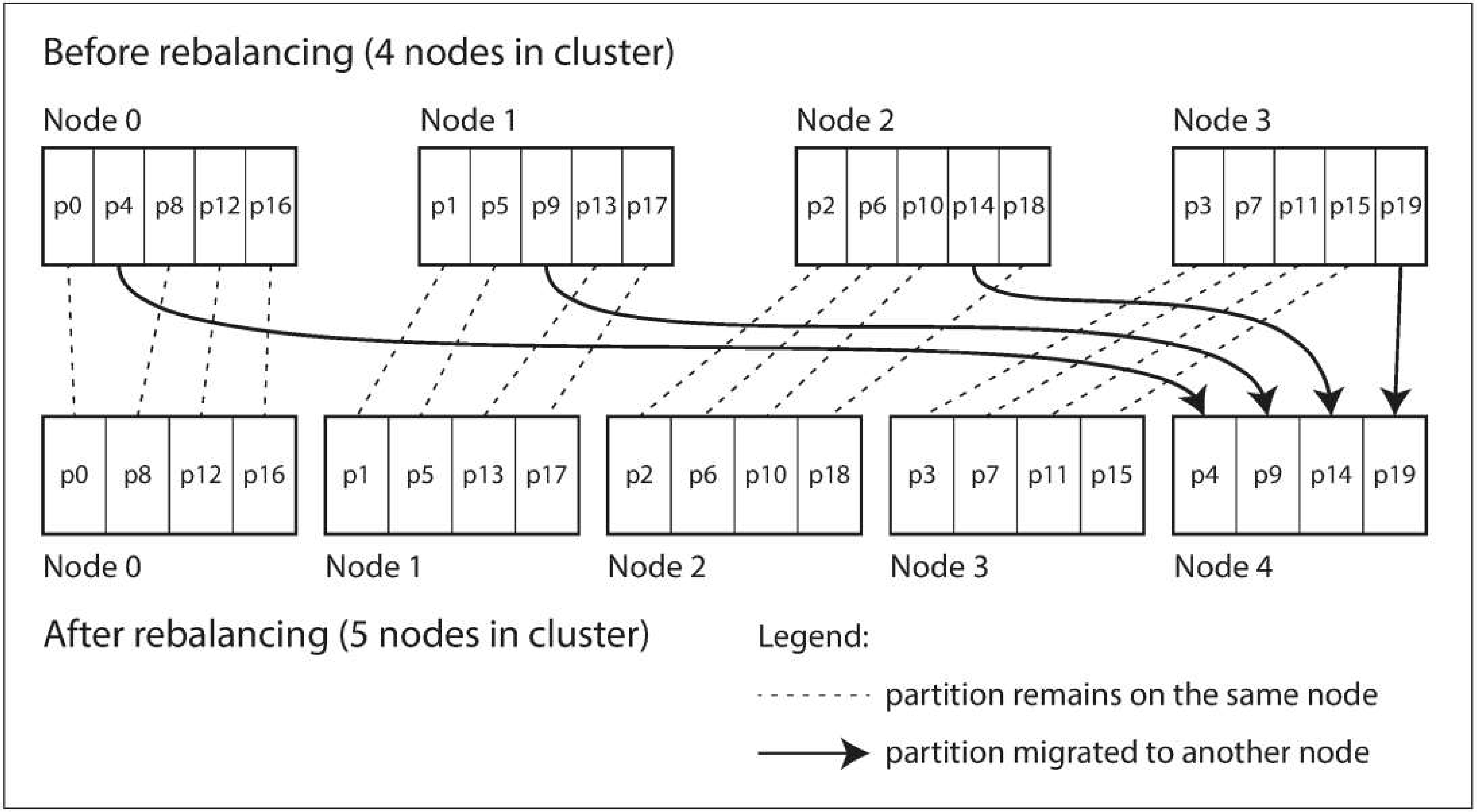
- Things change as time ticks forward
  - More CPU is needed as query **throughput** changes (**read/write throughputs**)
  - Data size increases, adding more RAM and disk storage is paramount
  - Machines may fail or need to reconfigured (**downtime** is unavoidable)
- **Rebalancing** means moving data or query requests between cluster **nodes**
- Requirements
  - Load should be **evenly** distributed after rebalancing
  - Reads/writes should **continue operating** while in the rebalancing phase
  - Moving what is necessary only, to **minimize the IO and network overheads**

Rebalancing approaches

- Two approaches
  - Approaches that partition in a way proportional to dataset size
    - **Fixed number** of partitions
      - With **hash key** partitioning
    - **Dynamic** partitioning
      - With **key range** partitioning
  - Approaches that partition in a way proportional to cluster size (**number of nodes**)
    - Fixed number of partitions per node

Rebalancing approaches

- For **hash key partitioning**
  - Using fixed number of partitions is preferred over other assignments (such as using the mod operation over the hash key)
    - If we use "mod" over hash key, then every time we add partitions or nodes, all records need to be redistributed because the operation (hash code % value) would result in a new value (partition number, thus another node), **expensive**
    - Alternatively, having a fixed number of partitions (say 100) means that adding nodes does not affect the intra-partition data
      - What then needs to be redistributed is full partitions, not **record-by-record**
      - Used in **Elasticsearch** & **Couchbase**



Before rebalancing (4 nodes in cluster)
Node 0
p0 p4 p8 p12 p16
Node 1
p1 p5 p9 p13 p17
Node 2
p2 p6 p10 p14 p18
Node 3
p3 p7 p11 p15 p19
p0 p8 p12 p16
p1 p5 p13 p17
p2 p6 p10 p18
p3 p7 p11 p15
p4 p9 p14 p19
Node 0
Node 1
Node 2
Node 3
Node 4
After rebalancing (5 nodes in cluster)
Legend:
partition remains on the same node
partition migrated to another node

Rebalancing approaches (cont.)

- For **key range partitioning**
  - Fixed number of partitions is prone to unbalanced loads
  - Some partitions would have more data (**hotspots**) than others (**idle**)
- Partition **dynamically**
  - Build partitions as data arrive
    - **Adaptable** partitioning that senses the data volume
  - When the size exceeds the **threshold**, **split** the partition and send the new partition to another node if necessary
  - When the size **shrinks**, **combine adjacent** partitions
  - However, the start is an issue
    - With single partition, all writes, and reads are handled by a single node
    - Until the partition size reaches the limit, only then **parallelization** benefits come on board
- Common in **MongoDB, RethinkDB & HBase**

Cluster size-driven partitioning

- **Fixed** number of partitions per node of the cluster
- Adding nodes
  - **Split** partitions **randomly** so that the number of partitions per node for the new configuration matches the preset configuration
  - Move some of the **split** partitions to the new nodes to achieve the required number of partitions per node (**approximately**)
  - Adopted in **Cassandra**

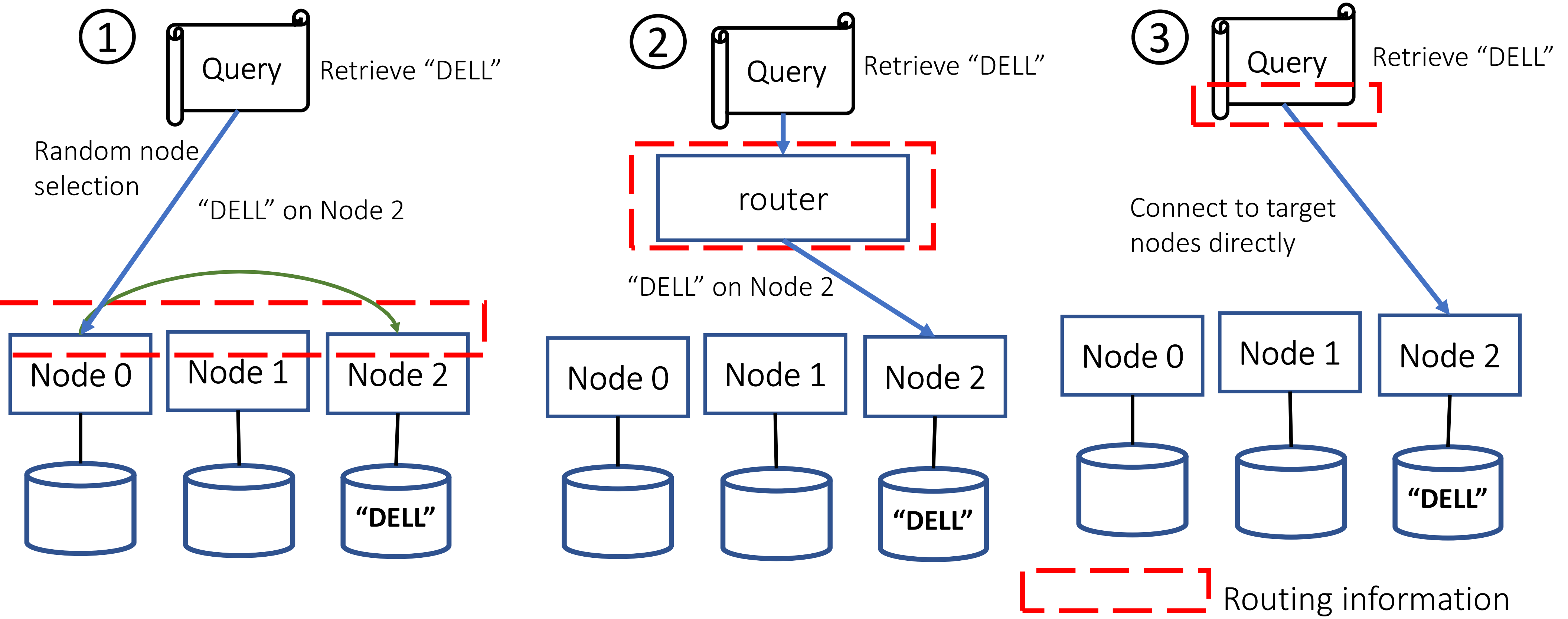
Human-in-the-loop (HITL) for rebalancing

- Rebalancing could be very expensive
  - **IO** and **network** transfer **overheads**
  - A mistakenly rebalancing decision with a fake automatic failure detection can bring the system into halt!
  - So, **HITL** is preferred

Query forwarding

- Also known as **query request routing**
  - Which **nodes to visit** for answering a specific query
- Various approaches
  - **Random**
  - **Routers**
  - **Client-side**
- How the **router** knows about the partition assignment?
  - **coordination service** such as ***Zookeeper*** to keep track of this kind cluster metadata
  - HBase, SolrCloud, and Kafka also use ZooKeeper
  - MongoDB relies on its own ***config server*** implementation and ***mongos*** daemons as the **routing tier.** Also, **Couchbase** utilize a similar approach with routing tier known as *moxi*
  - Cassandra uses **Gossip *protocol*** → random approach

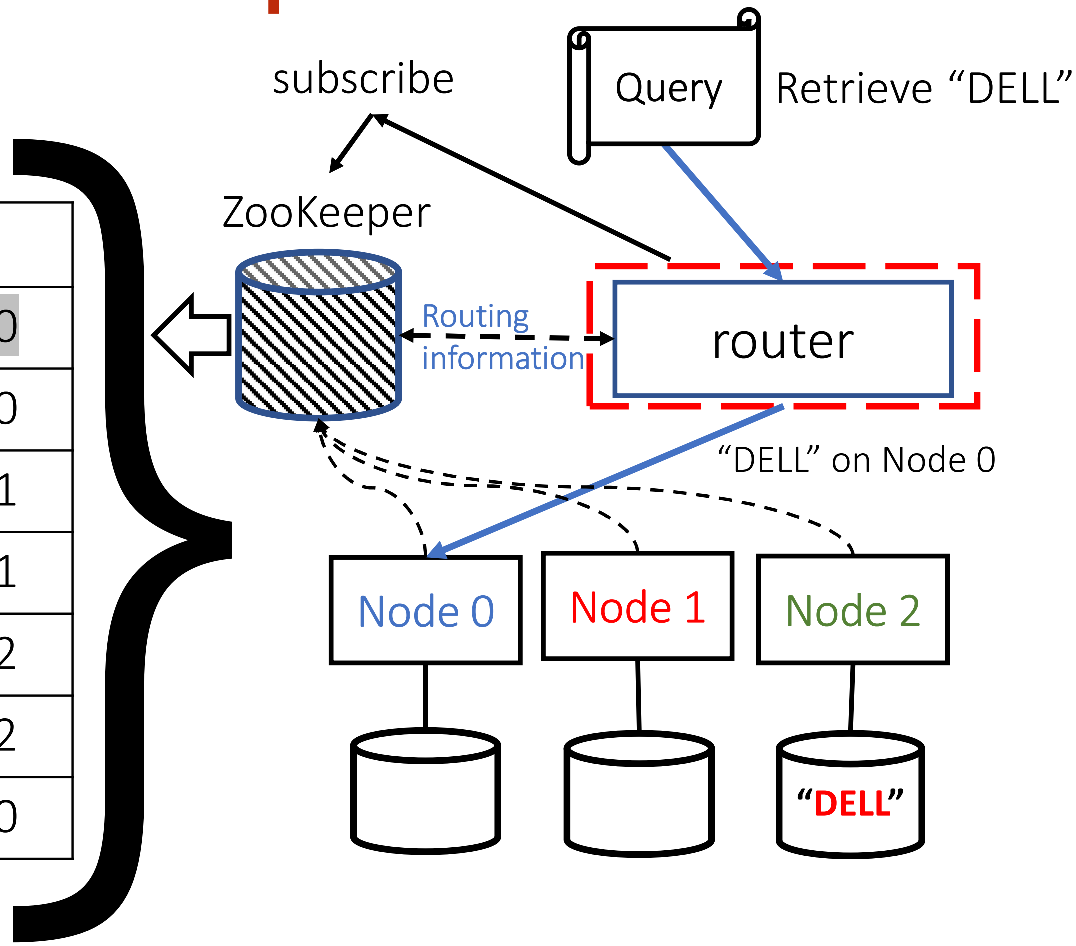
Query forwarding approaches



Coordination service - Zookeeper

mapping of partitions to nodes

Key range	partition	Node	IP address
A – D	Partition 0	Node 0	10.10.10.100
E – H	Partition 1	Node 0	10.10.10.100
I – L	Partition 2	Node 1	10.10.10.101
M – O	Partition 3	Node 1	10.10.10.101
Q – S	Partition 4	Node 2	10.10.10.102
T – W	Partition 5	Node 2	10.10.10.102
X - Z	Partition 6	Node 0	10.10.10.100



Cloud data management solutions

Data models & query languages

Data models layers

- Layering one data model on top of another
  - For each layer, the key question is how it is *represented* in terms of the **next-lower layer**
  - each layer **hides** the **complexity** of the layers below it by providing a clean **data model**

Objects (sensors, cars)	Application developer view
general-purpose data model (JSON or XML)	Logical storage view
bytes in memory, on disk, or on a network.	Physical storage view
Electric currents, light pulses, magnetic fields	Hardware level

Choosing a data model

- Many kinds of **data models**
- Data model in a layer affects the performance of the software on a **top layer**
  - Select a data model that helps the performance of the data application
- How to choose
  - **Easy** to use against **hard** usage
  - Supported **operations** and how fast
  - Supported **data** transformation

Challenges in choosing data models

- The key challenge in selecting data model is the ability to strike the **plausible balance** of the **needs** of the application,
  - the **performance** characteristics of the database engine, and the data **retrieval patterns**
- When designing data models, we always consider
  - the **usage** of the data by the underlying application (i.e., queries, updates, and processing of the data)
  - In addition to the inherent **structure** of the data

Relational Databases Example

Example SQL queries

```
1. SELECT zipcode FROM users WHERE name = "Bob";
2. SELECT url FROM blog WHERE id = 3;
3. SELECT users.zipcode, blog.num_posts FROM users JOIN blog ON users.blog_url = blog.url;
```

user_id	name	zipcode	blog_url	blog_id
101	Alice	12345	alice.net	1
422	Charlie	45783	charlie.com	2
555	Bob	99910	bob.bloogspot .com	3

Users Tables

Primary keys

Foreign keys

id	url	last_updat ed	num_posts
1	alice.net	5/2/14	332
2	bob.bloogspot .com	4/2/13	10003
555	charlie.com	6/15/14	7

Blog Tables

Mismatch with today workloads

Data are extremely **large and unstructured**

Lots of **random reads and writes**

Sometimes **write-heavy**

Foreign keys rarely needed

Joins rare

Typically, **not regular queries** and sometimes very **forecastable** (so you can **prepare for them**)

In other terms, you can prepare data for the usage you want to optimize

Requirement of today workloads

- **Speed in answering**
- **No Single point of Failure** (**SPoF**)
- **Low TCO** (**T**otal **C**ost of **O**peration) or **efficiency**
- **Fewer system administrators**
- **Incremental Scalability**

- **Scale out, not up**
  - What?

Scale out, not scale out

Scale up => grow **your cluster capacity** by replacing **more powerful machines**

the so-called **vertical scalability**

- Traditional approach
- Not cost-effective, as you are buying above the sweet spot on the price curve
- and you need to replace machines often

Scale out => incrementally **grow your cluster capacity by adding more COTS machines** (Components Off The Shelf)

the so-called **horizontal scalability**

- Cheaper and more effective
- Over a long duration, phase in a few newer (faster) machines as you phase out a few older machines
- Used by **most companies who run datacenters and clouds** today

Key-value/NoSQL Data Model

NoSQL = "**N**ot **o**nly **SQL**"

Necessary API operations: `get(key)` and `put(key, value);`

- And some extended operations, e.g., use of MapReduce in MongoDB

Tables

- Similar to RDBMS tables, but they …
- **Are unstructured: do not have schemas**
  - Some columns may be missing from some rows
- **Do not always support joins nor have foreign keys**
- **Can have index tables**, just like RDBMSs
  - "Table" in HBase
  - "Collection" in MongoDB

Key-value/NoSQL Data Model

Unstructured

Columns **Missing** of some Rows

No schema imposed

No foreign keys

Joins may not be supported

Key Value

user_id	name	zipcode	blog_url
101	Alice	12345	alice.net
422	Charl ie		charlie.com
555	Bob	99910	bob.bloogspot .com

Key Value

id	url	last_updated	num_posts
1	alice.net	5/2/14	332
2	bob.bloogspot .com		10003
55 5	chalie.com	6/15/14	7

Column-Oriented Storage

NoSQL systems can use **column-oriented storage**

RDBMSs store an **entire row together (on a disk)**

NoSQL systems typically **store a column together (also a group of columns)**

- Entries within a column are indexed and easy to locate, given a key (and vice-versa)

Why?

- Range searches **within a column are fast** since you do not need to fetch the entire database

  e.g., Get me all the blog_ids from the blog table that were updated within the past month;

  Search in the the **last_updated** column, fetch corresponding blog_id column, without fetching the other columns

MongoDB

MongoDB is **Document-oriented NoSQL** tool

Open source NoSQL DB

- In memory access to data
- Native replications toward reliability and high availability (CAP)
- Collection partitioning by using sharding key so to keep the information fast available and also replicated
- Designed in C++

Relational Model Concepts (cont'd.)

- Tables (relations), rows, columns
- Example: list of employees, containing their ID, name and phone
- Solution:

Relation

EMPLOYEE

Attributes (columns)

ID	Name	phone
1	Tony	999
2	Mark	888
3	Lisa	777

Rows (tuples)

Keys (cont'd.)

Less storage space is required!

Looks better!

WORKS_FOR

employeeID	deptID
2	11
2	22
3	22
4	11

EMPLOYEE

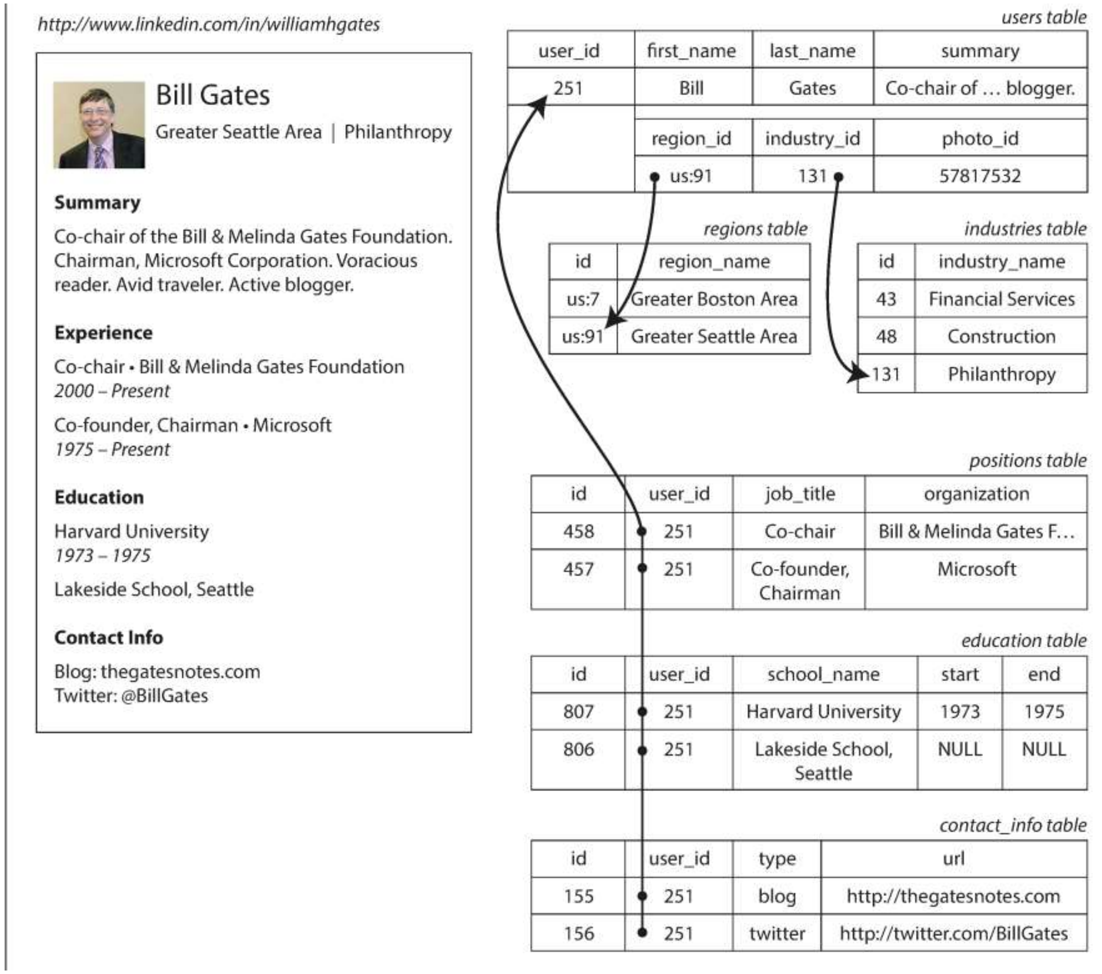
ID	name	phone
2	Mark	888
1	Tony	999
3	Lisa	777
4	Tom	NULL

DEPARTMENT

ID	name
11	marketing
22	IT
33	PR
44	communication

Why not relational model

- Requires costly **join**



NoSQL models

- **JSON** (e.g., MongoDB)
  - better ***locality*** than the multi-table schema
- No **join** is required (single query), read performance
  - support for **joins** is often **weak**
  - **Joins** can be performed in the **application layer**
- **Schema-less** (schema flexibility)
  - ***schema-on-read*** Vs. ***schema-on-write***
- closer to the data structures used by the application
- Limitations
  - Reading **nested** items
  - **Many-many** and **many-one** relationships

```
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University",       "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog":    "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

```
{
  "user_id":     251,
  "first_name":  "Bill",
  "last_name":   "Gates",
  "summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id":   "us:91",
  "industry_id": 131,
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
```

Data encoding

Serialization & marshalling

Data representation

- In-memory
  - Objects, structs, lists, arrays, hash tables, trees
  - Using pointers to speed up access
- Disk-resident & cross-network
  - Sequence of bytes (e.g., **JSON**)
  - Pointers diminish at this stage, different data representation
- Translation between i**n-memory** and **disk-resident** representations is required
  - **Encoding** (also goes by other names (**serialization** or **marshalling**)
  - The opposite process is **decoding** (**parsing**, **deserialization**, **unmarshalling**)

Encoding models


```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```


- Language specific
  - Examples
    - **Java** Serializable
    - Python **pickle**
    - **Kryo** for Java (3rd party)
  - **Tied** to specific language, reading in other languages requires taking care of additional **logistics**
- **JSON** & **XML**
  - Standardized encodings textual format that can be written and read by many programming languages
  - **JSON** is simpler
  - **CSV** is another popular option
  - **Schema-less** (schema-on-read)
  - **BSON** is a binary encoding variant of JSON, requires **less space**
  - **Avro** is another binary encoding
    - Uses a **schema** to specify the structure of the data being encoded
    - The most compact of all the encodings we have seen
      - Omit field names from the encoded data
- **JSON** is a very viable choice for cloud data management

Cloud programming models

Batch processing models

Data processing in today large clusters

- Engineers can focus only on the application logic and parallel tasks, without the hassle of dealing with scheduling, fault-tolerance, and synchronization
- **MapReduce** is a **programming framework** that provides
- **High-level API** to specify **parallel tasks**
- **Runtime system** that takes care of
    - Automatic parallelization & scheduling
    - Load balancing
    - Fault tolerance
    - I/O scheduling
    - Monitoring & status updates
- Everything runs on top of **GFS** (the distributed file system)

User Benefits

- **Automatize everything** – for useful **special-purpose** behavior in **two steps of complementary operations**

- Based on **abstract black box** approach
- Huge **speedups** in programming/prototyping
  «it makes it possible to write a simple program and run it efficiently on a thousand machines in a half hour»
- Programmers can exploit **quite easily very large amounts of resources**
  - Including **users with no experience** in distributed / parallel systems

Traditional MapReduce definitions

- Statements that go back to **functional languages** (such as LISP, Scheme) as a **sequence of two steps for parallel exploration and results (Map and Reduce)**.

- Also in other programming languages: **Map/Reduce** in Python, Map in Perl

- **Map** (**distribution** phase)
  1. **Input**: a *list of data* and one *function*
  2. **Execution**: the function **is applied to** each list item
  3. **Result**: a *new list* with all the results of the function

- **Reduce** (**result harvesting** phase)
  1. **Input**: a *list* and one *function*
  2. **Execution**: the function **combines/aggregates** the list items
  3. **Result**: one **new final item**

What is MapReduce in a nutshell

- The terms are borrowed from Functional Languages (e.g., Lisp)

- **Sum of squares**:
- `(map square '(1 2 3 4)) => Output: (1 4 9 16)`

  [processes each record **sequentially and independently**]
- **(reduce + '(1 4 9 16)) =>** (+ 16 (+ 9 (+ 4 1))) => Output: 30

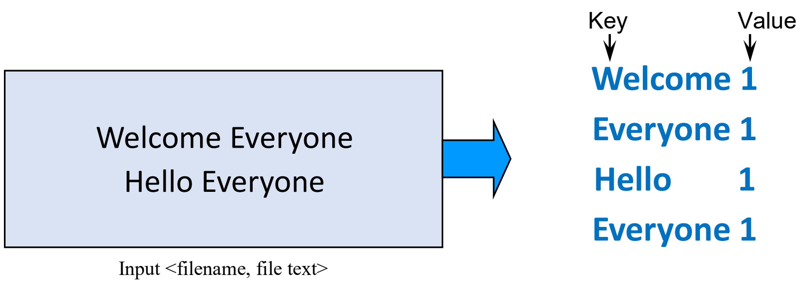
  [processes set of **all records in batches**]

- Let us consider a sample application: Wordcount

You are given a **huge dataset** (e.g., Wikipedia dump – or all of Shakespeare's works) and asked to list **the count for each of the words in any of the searched documents**

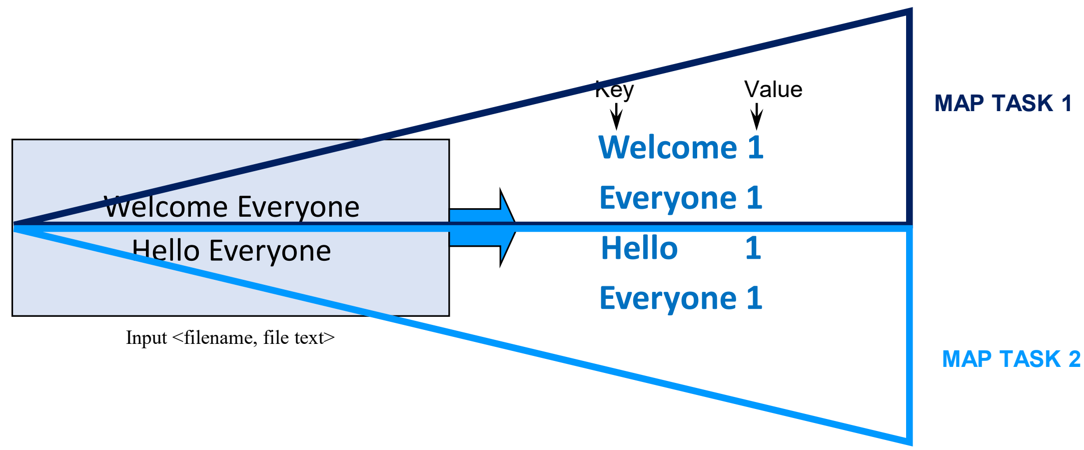
Map

- Extensively apply the function
- **Process all single records to generate intermediate key/value pairs.**



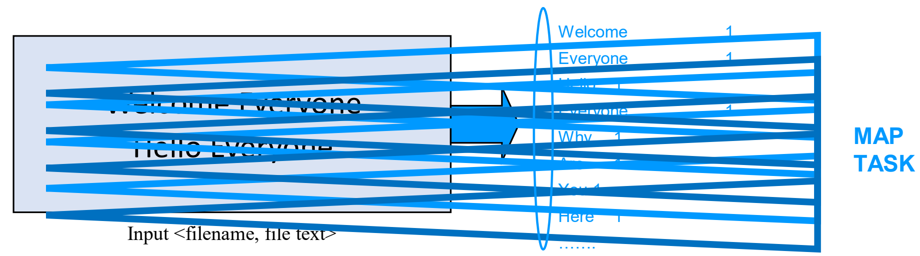
Map

- **In parallel** process individual records to generate intermediate key/value pairs



Map

- **In parallel** process **a large number** of individual records to generate intermediate key/value pairs



Reduce

- Collect the whole information
- Reduce processes and **merges all intermediate values** associated **per key**

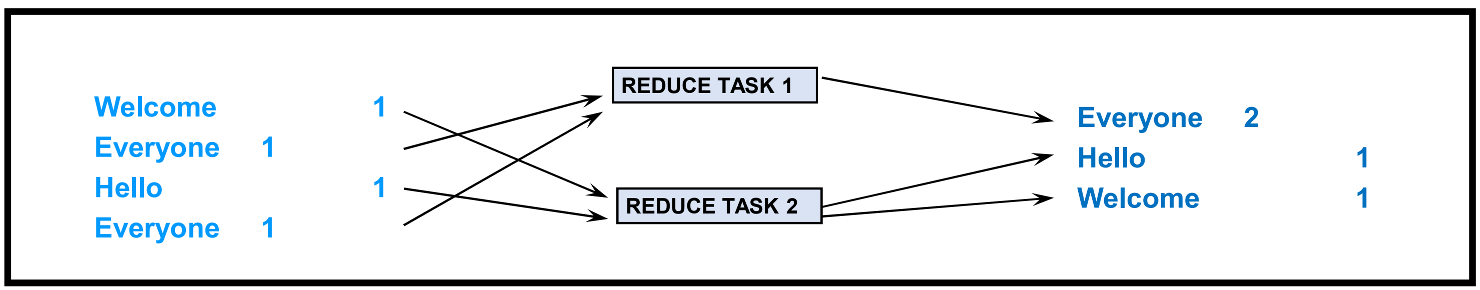
Welcome	1
Everyone	1
Hello	1
Everyone	1

Key	Value
Everyone	2
Hello	1
Welcome	1

Reduce

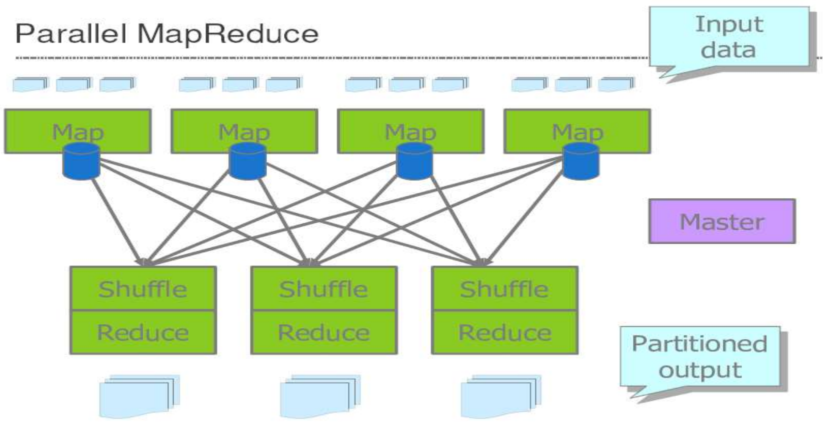
- Each key assigned to one Reduce
- **In parallel** processes and merges all intermediate values by partitioning keys

- **Popular splitting**: *Hash partitioning, such as* key is assigned to
  - reduce # = hash(key)%number of reduce tasks



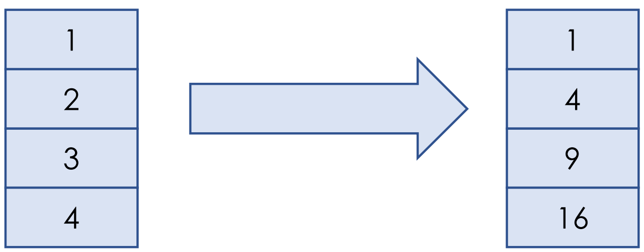
MapReduce: a deployment view

- Read many chunks of distributed data (no data dependencies)
- **Map**: extract something from each chunk of data
- **Shuffle and sort**
- **Reduce**: aggregate, summarize, filter or transform sorted data
- Programmers can specify the **Map** and **Reduce functions**

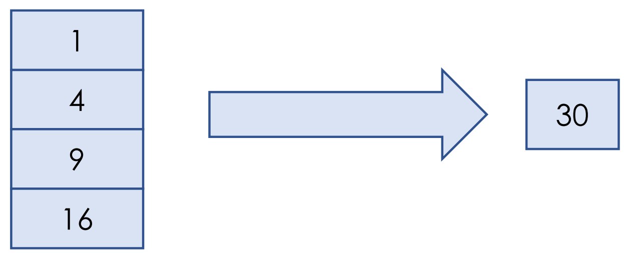


Traditional MapReduce examples (again)

Map (square, [1, 2, 3, 4])



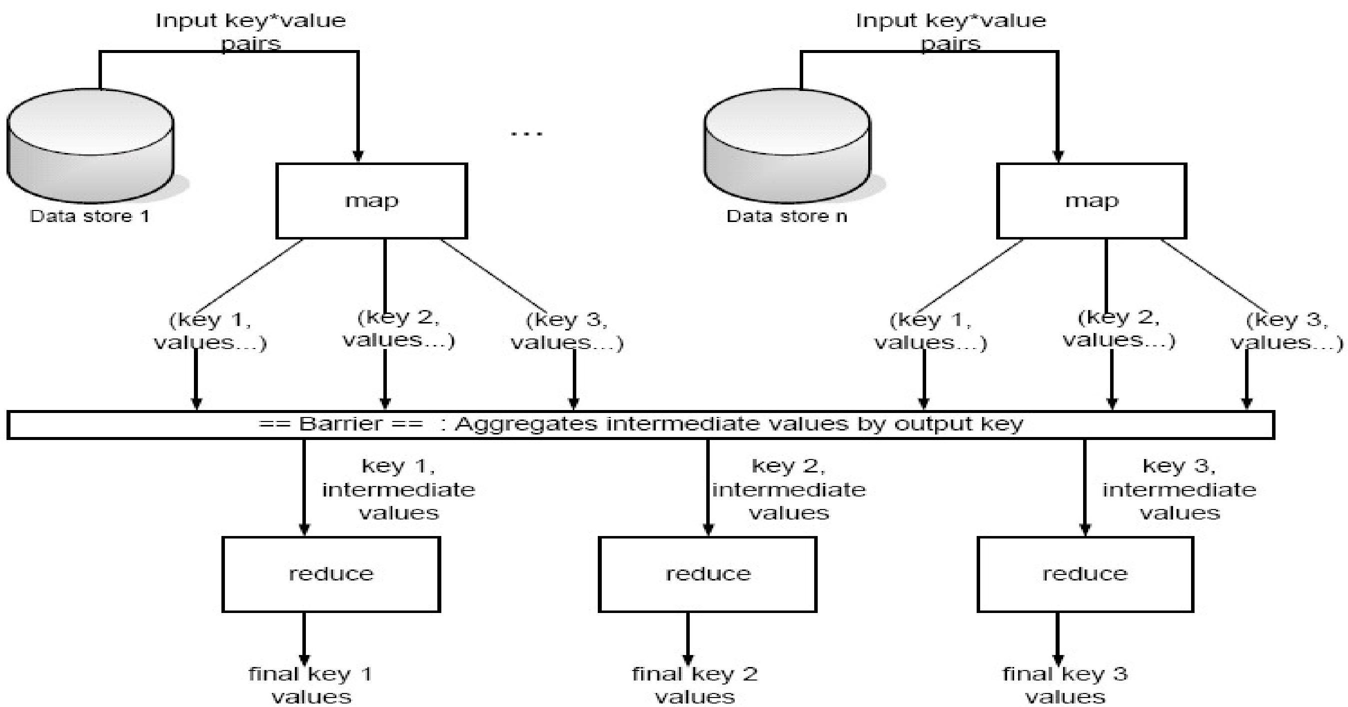
Reduce (add, [1, 4, 9, 16])



Google MapReduce definition

- **map (String key, String val)** runs on each item in the set
- **Input example**: a set of files, with keys being **file names** and values being **file contents**
- Keys & values can have different types: the programmer has to convert between Strings and appropriate types inside map()
- **Emits**, i.e., outputs, (new-key, new-val) pairs
- Size of output set can be different from size of input set
- The runtime system **aggregates the output of map by key**
- **reduce (String key, Iterator vals)** runs for each *unique* key emitted by map()
- It is possible to have more values for one key
- **Emits final output pairs** (possibly smaller set than the intermediate sorted set)

Map & aggregation must finish before reduce can start



Running a MapReduce program

- The final user fills in ***specification object:***
- ***Input/output file names***
- ***Optional tuning parameters***
  (e.g., size to split input/output into)

- The final user defines ***MapReduce function*** and passes it ***the specification object***

- The **runtime system calls map() and reduce()**
  - While the final user just has to specify the operations

Word Count Example

- **map(String input_key, String input_value):**
- // input_key: document name
- // input_value: document contents

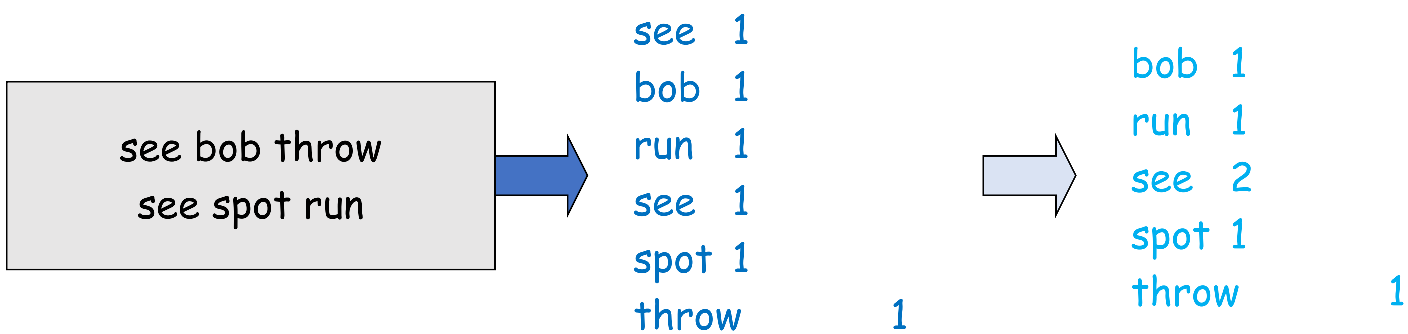
```
for each word w in input_value:
  EmitIntermediate(w, "1");
```

- **reduce(String output_key,
                  Iterator intermediate_values):**
- // output_key: a word
- // output_values: a list of counts

```
int result = 0;
for each v in intermediate_values:
  result += ParseInt(v);
Emit(AsString(result));
```

Word Count Illustrated

- `map(key=url, val=contents):`
  - `For each word w in contents, emit (w, "1")`
- `reduce(key=word, values=uniq_counts):`
  - `Sum all "1"s in values list`
  - `Emit result "(word, sum)"`



Many other applications

- **Distributed *grep***
  - map() emits a line if it matches a supplied pattern
  - reduce() is an identity function; just emit same line

- **Distributed *sort***
  - map() extracts *sorting key* from record (file) and outputs (key, record) pairs
  - reduce() is an identity function; just emit same pairs
  - The actual sort is done automatically by runtime system
- **Reverse web-link graph**
  - map() emits (*target*, *source*) pairs for each link to a *target* URL found in a file *source*
  - reduce() emits pairs (*target*, list(*source*))

other applications

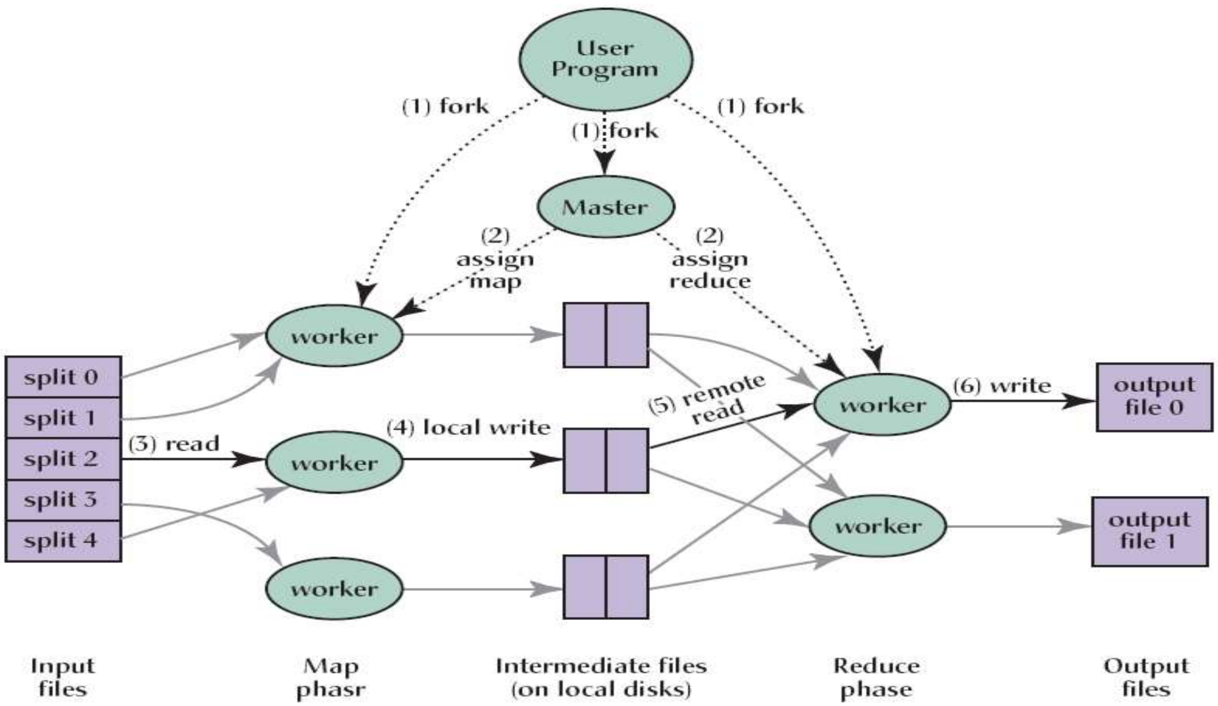
- **Machine learning** issues
- Google **news clustering** problems
- **Extracting data** + reporting **popular queries** (Zeitgeist)
- Extract **properties** of web pages for tests/products
- Processing satellite imagery data
- Graph computations
- Language model for machine translation

- **Rewrite of Google Indexing Code** in MapReduce
  Size of one phase 3800 => 700 lines, over 5x drop

Implementation overview (at google)

- **Environment:**
- **Large clusters of PCs connected with Gigabit links**
  - 4-8 GB RAM per machine, dual x86 processors
  - Network bandwidth often significantly less than 1 GB/s
  - Machine failures are common due to **#** machines
- **GFS**: distributed file system manages data
  - Storage is provided by cheap IDE disks attached to machine
- **Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines**
- **Implementation is a C++ library linked into user programs**

Architecture example



Scheduling and execution

- **One master, many workers**
- Input data split into M map tasks (typically 64 MB in size)
- Reduce phase partitioned into R reduce tasks
- Tasks are assigned to workers dynamically
- Often: M=200,000; R=4000; workers=2000
- **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)
- Intermediate key/value pairs written to **local disk**, divided **into R regions**, and the locations of the regions are passed to the master
- **Master assigns each reduce task to a free worker**
- Worker reads intermediate k/v pairs from map workers
- Worker applies user reduce operation to produce the output (stored in GFS)

Fault-Tolerance

- **On master failure**:
- State is checkpointed to GFS: new master recovers & continues
- **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)
- **Robustness**:
- Example: Lost 1600 of 1800 machines once, but success

Favouring Data Locality

- The goal is **to preserve and to conserve network bandwidth**
- In GFS, we know that data files are divided into 64 MB 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)
  - In this way, the machines can read input at local disk speed. Otherwise, rack switches would limit read rate

backup Tasks

Problem: stragglers (i.e., **slow workers in ending**) significantly lengthen the completion time

- Other jobs may be consuming resources on machine
- Bad disks with soft errors (i.e., correctable) transfer data very slowly
- Other weird things: processor caches disabled at machine init

- **Solution:** Close to completion, **spawn backup copies of the remaining in-progress tasks**
- Whichever one finishes first, wins
- Additional cost: a few percent more resource usage
- Example: A sort program without backup was 44% longer

Example systems

Apache Hadoop, Flink, Storm, Spark, Kafka, Cassandra and MongoDB

Batch Processing

Hadoop: a Java-based MapReduce

- **Hadoop** is an **open source platform for MapReduce by Apache**
- Started as open source MapReduce written in Java, but evolved to support other languages such as Pig and Hive
- ***Hadoop common***
  set of utilities that support the other subprojects:
- FileSystem, RPC, and serialization libraries
- **Several essential subprojects:**
- **Distributed file system (*HDFS*)**
- ***MapReduce***
- **Yet Another Resource Negotiator** (**YARN**) for cluster resource management

Hadoop MapReduce

- Its batch-processing component is called **Hadoop MapReduce**