BigData & NoSQL DBMSs

Tecnologie delle Basi di Dati M

lucidi a cura del prof. Torlone (univ. Roma3)

Big data? Why? Why not just data?

- Well, because they are:
  1. Big
     - “The greater the struggle, the more glorious the triumph.” (Butterfly Circus)
  2. Necessary
     - “It is a capital mistake to theorize before one has data.” (S. Holmes)
  3. Fashionable
     - “I always wanted to be fashionable.” (J. Malkovich)
  4. Profitable
     - “Data is a precious thing and will last longer than the systems themselves.” (T. Berners-Lee)
  5. Exciting
     - “The most exciting phrase to hear in science, is not ‘Eureka!’, but ‘That's funny’…” (I. Asimov)

Goals

- Show state-of-the-art techniques for dealing with collections of unstructured data whose size exceeds the capacity of storage, management, and analysis typical for traditional (relational) database systems
- In particular:
  - Requirements for modern applications
  - Problems with big data
  - Available hardware/software solutions

Roadmap

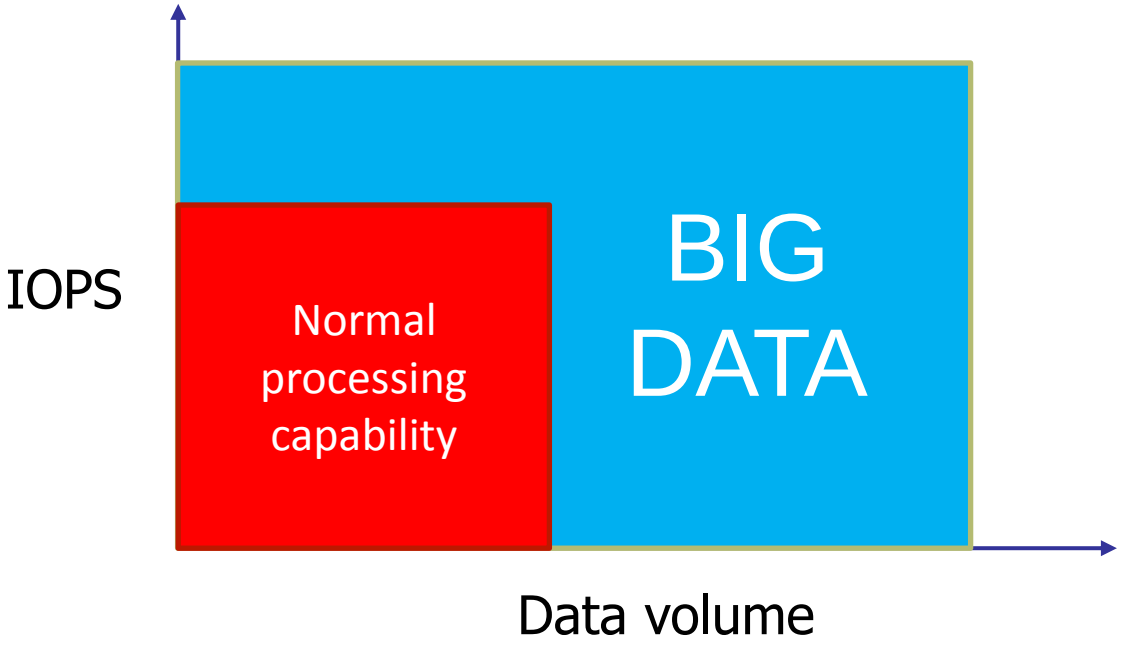
- Introduction
  - Terminology, principal characteristics, and application samples
- Storing Big Data
  - Hadoop & Map-reduce
  - Cloud computing
  - NoSQL DBMS

- …but there's more!
  - Big data computing (high-level tools like Pig/Hive)
  - Big data analysis (technologies like Mahout/Open R)
  - Applications (Semantic web/open data/social networks/genomic data)
- simply not enough time…

Big Data? Different definitions!

- “Big data exceeds the reach of commonly used hardware environments and software tools to capture, manage, and process it with in a tolerable elapsed time for its user population.” (Teradata Magazine article, 2011)
- “Big data refers to data sets whose size is beyond the ability of typical database software tools to capture, store, manage and analyze.” (The McKinsey Global Institute, 2012)
- “Big data is a term for data sets that are so large or complex that traditional data processing applications are inadequate.” (Wikipedia, 2016)

When data become "Big"?



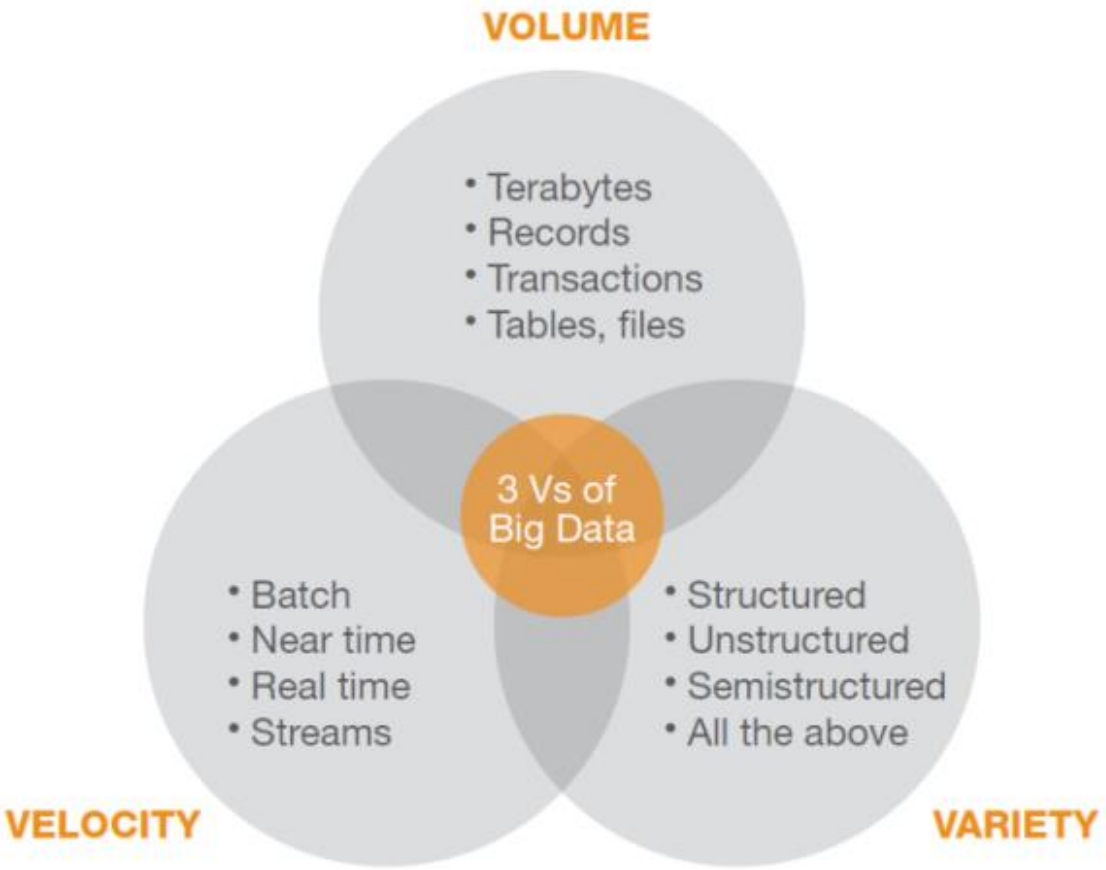
IOPS: Input/Output Operations Per Second

Some numbers

- How many data in the world?
  - 800 Terabytes, 2000
  - 160 Exabytes, 2006 (1EB = 10^{18}B)
  - 500 Exabytes, 2009
  - 2.7 Zettabytes, 2012 (1ZB = 10^{21}B)
  - 35 Zettabytes by 2020
- How much is a zettabyte?
  - 1,000,000,000,000,000,000,000 bytes
  - A stack of 1TB hard disks that is 25,400 km high
- How many data in a day?
  - 7 TB, Twitter
  - 10 TB, Facebook
- 90% of world's data:
  - generated over last two years!

The three "V's" of Big Data

- Not just a matter of volume…



What is more important?

- The “Big”
- The “Data”
- Both
- Neither

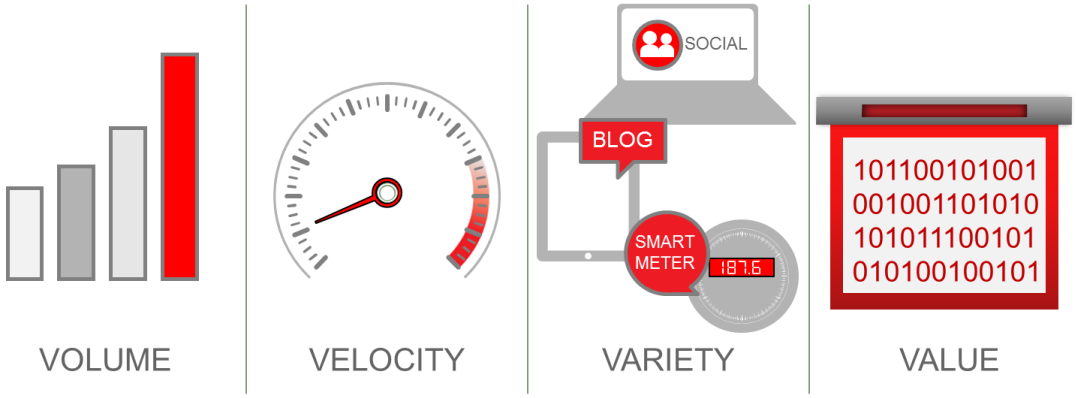
What organizations do with big data

"Data is not information,
information is not knowledge,
knowledge is not understanding,
understanding is not wisdom"

(Cliff Stoll)

Big Data: V^3+VALUE

- Volume: Gigabyte(109), Terabyte(1012), Petabyte(1015), Exabyte(1018), Zettabyte (1021)
- Variety: Structured, semi-structured, unstructured; Text, image, audio, video, record
- Velocity: Periodic, Near Real Time, Real Time
- Value: they can generate huge competitive advantages!



What's new?

- The wide availability of data allows us to apply more sophisticated models and you get much more accurate results than in the past!

"It is a capital mistake to theorize before one has data."

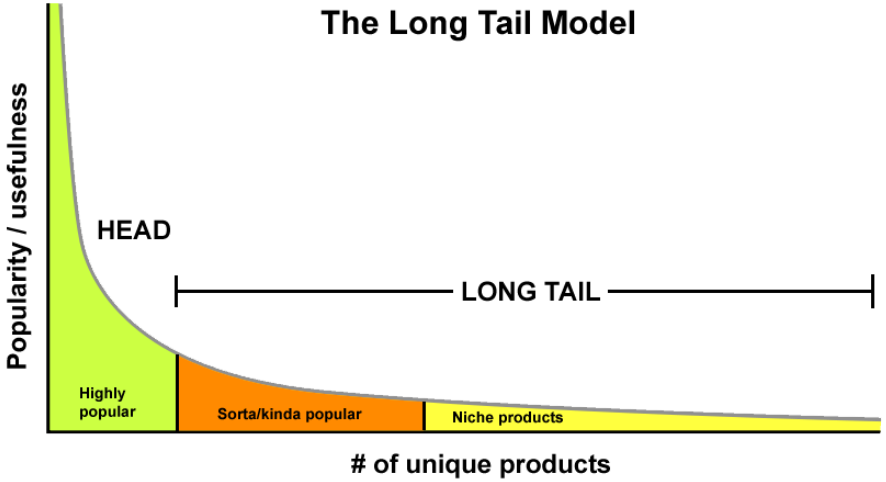
Sherlock Holmes

"Big data is mostly about taking numbers
and using those numbers to make predictions about the future.
The bigger the data set you have, the more accurate
the predictions about the future will be."

Anthony Goldbloom

Bigger = Smarter?

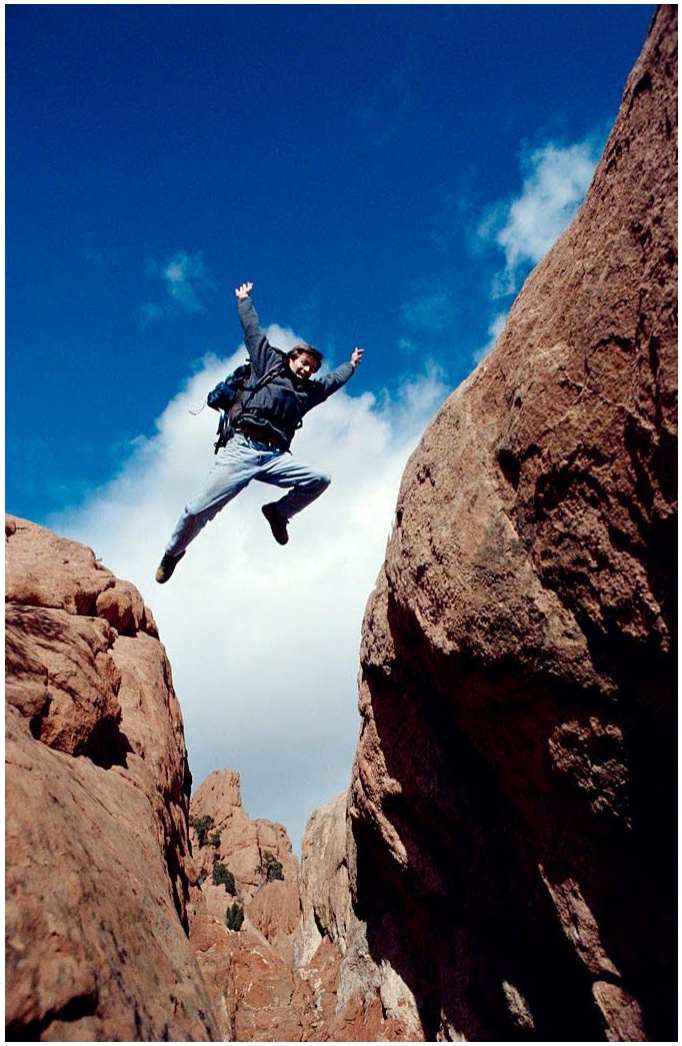
- Yes!
  - tolerate errors
  - discover the "long tail" and "corner cases"
  - algorithms work much better
- BUT:
  - more heterogeneity
  - data grows faster than energy on chip
  - still need humans to ask right questions



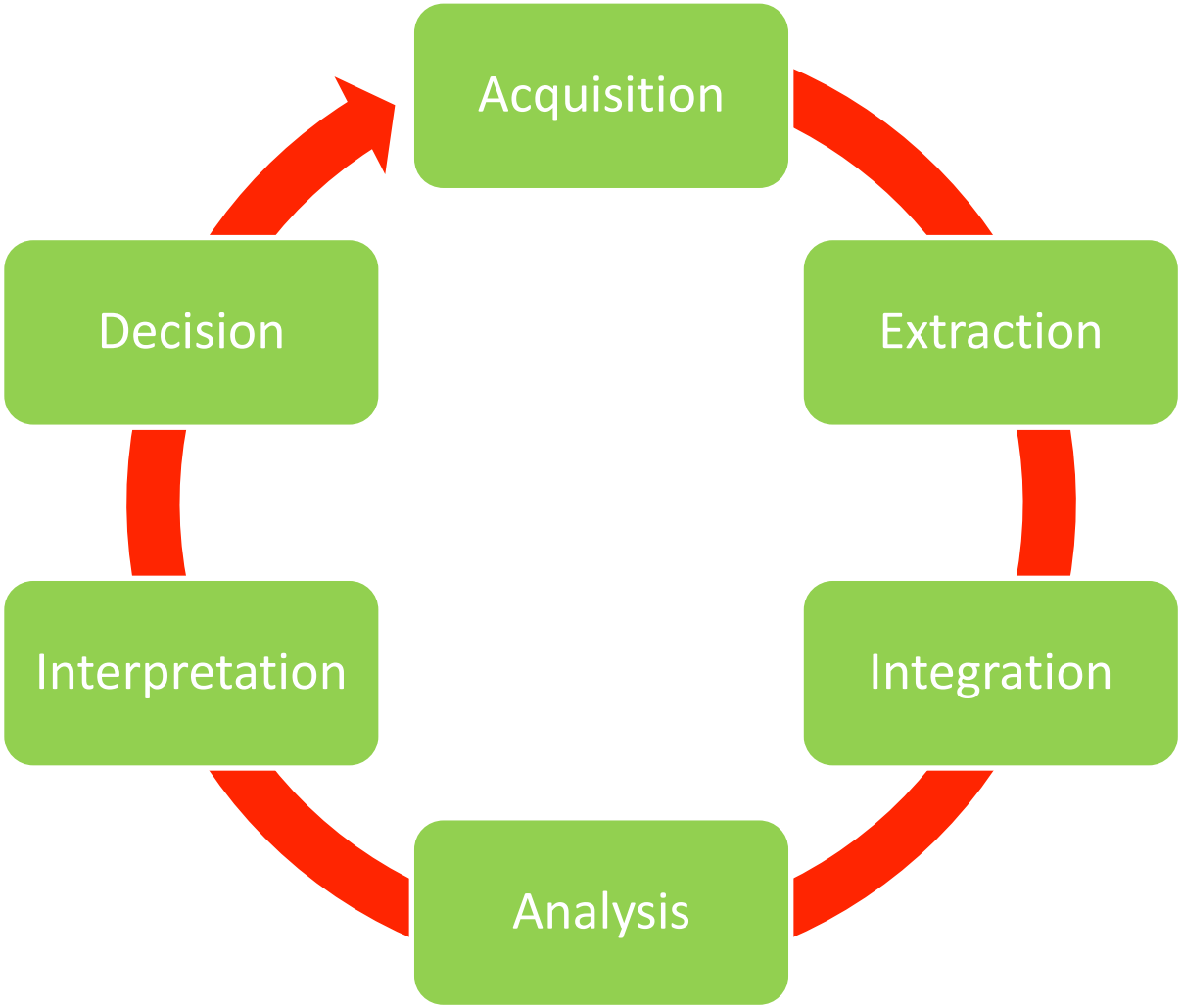
"We sold more books today that didn't sell at all yesterday than we sold today of all the books that did sell yesterday" (Amazon employee)

The risks of Big Data

- Data grows faster than energy on chip
  - Efficiency
  - Effectiveness
  - Scalability
- Costs
- Privacy



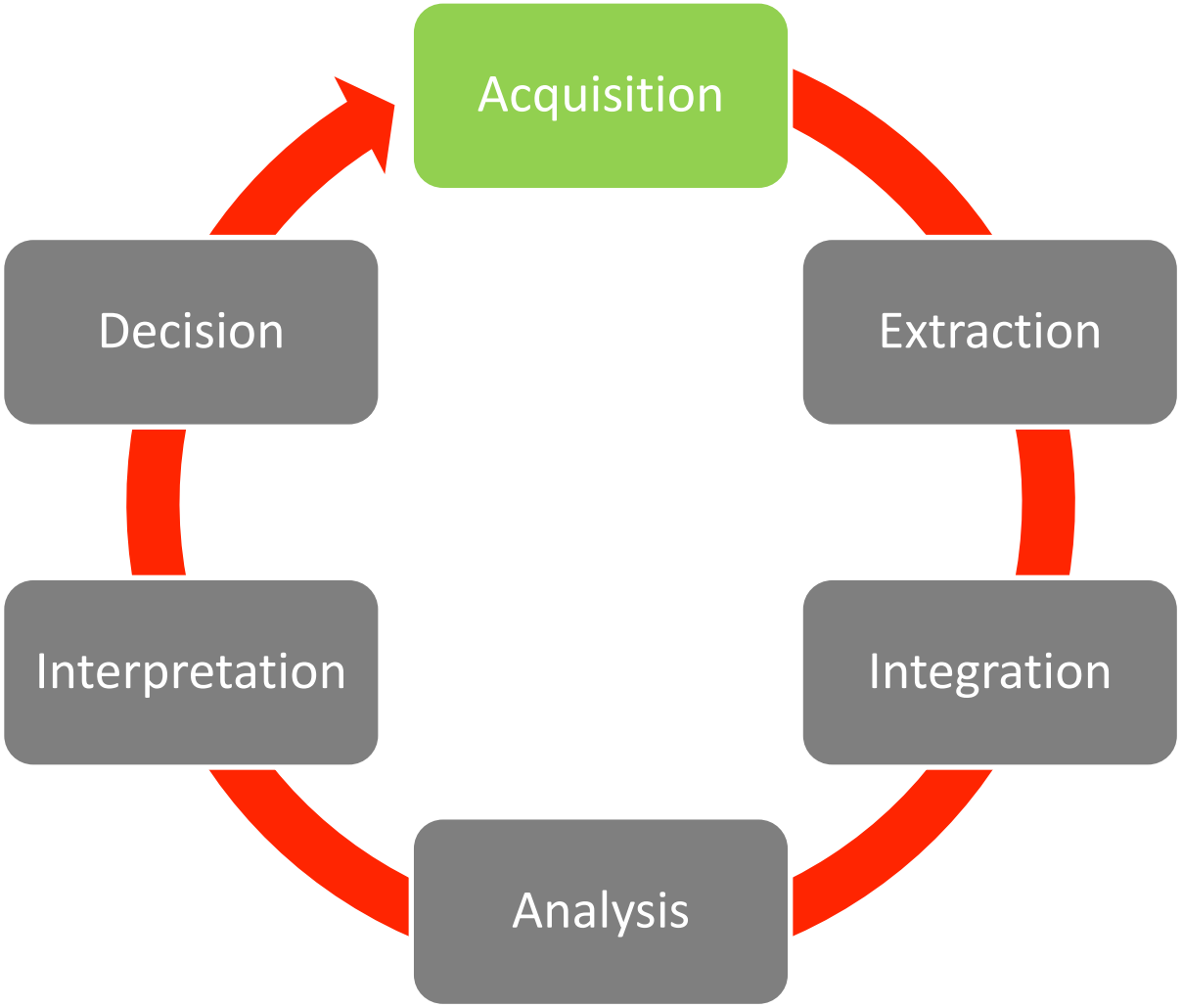
Big Data in action



Goal:

to make effective strategic decisions exploiting the availability of big data

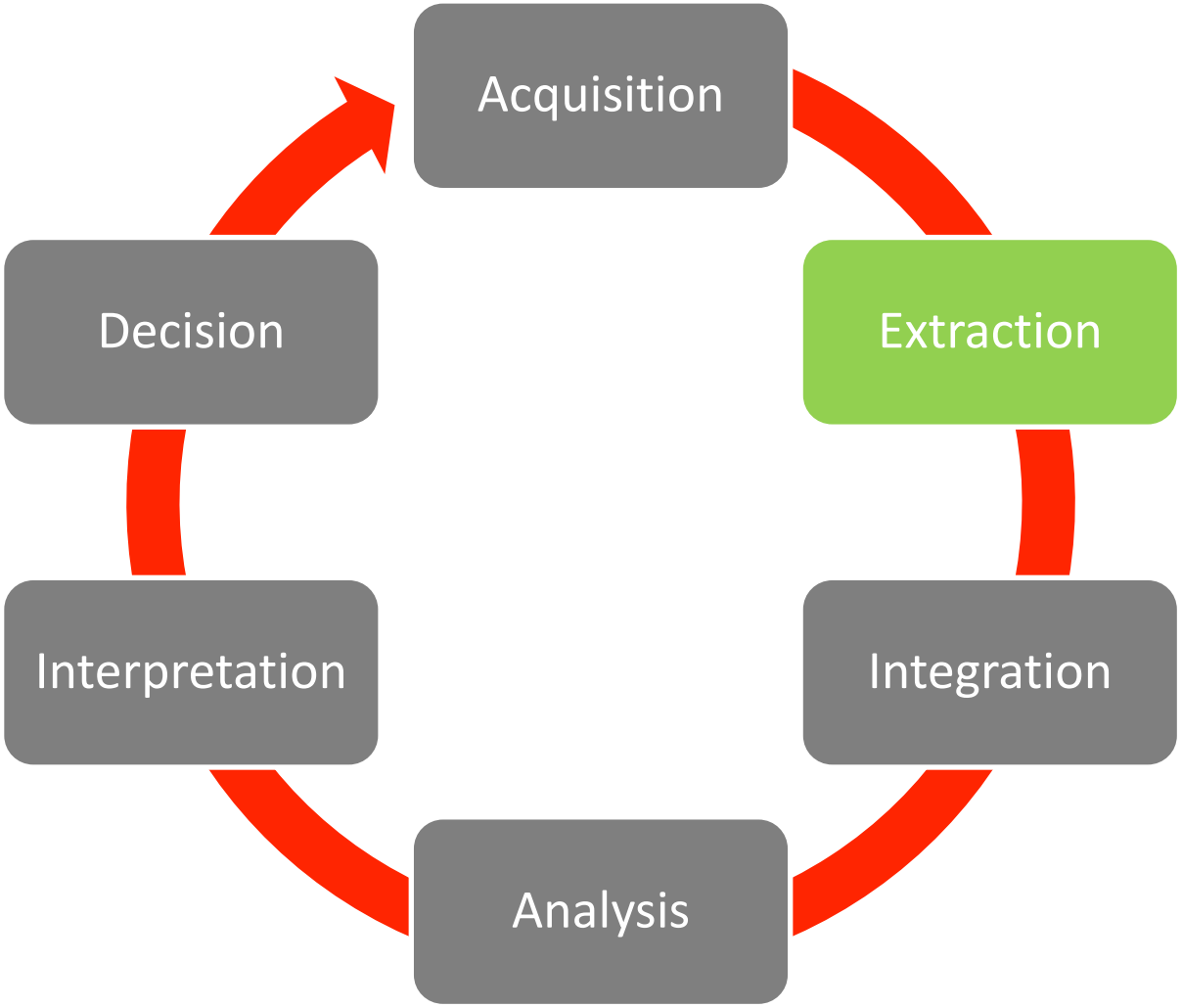
Big Data in action



Requires:

- selection
- filtering
- metadata generation
- managing provenance

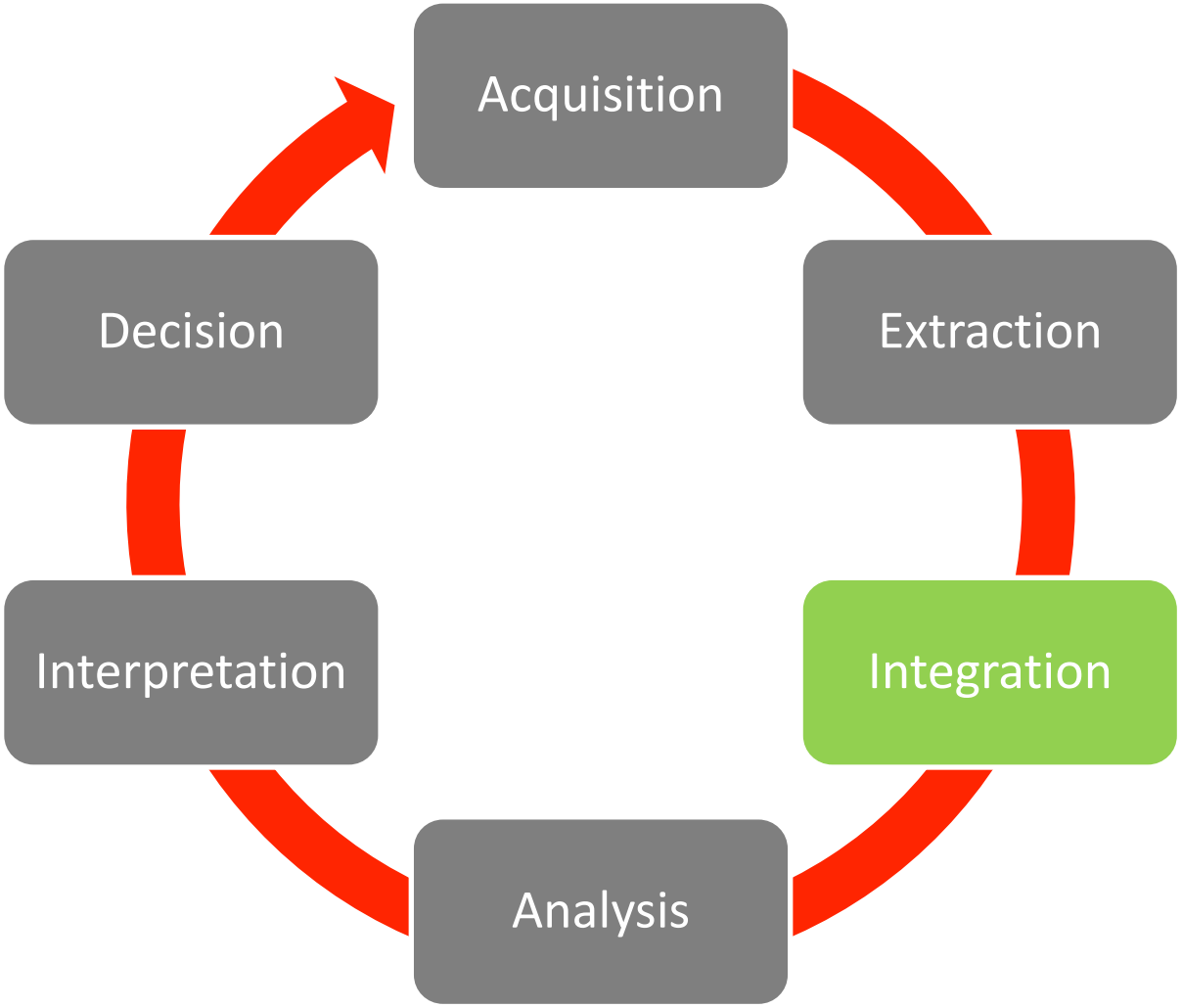
Big Data in action



Requires:

- transformation
- normalization
- cleaning
- aggregation
- error handling

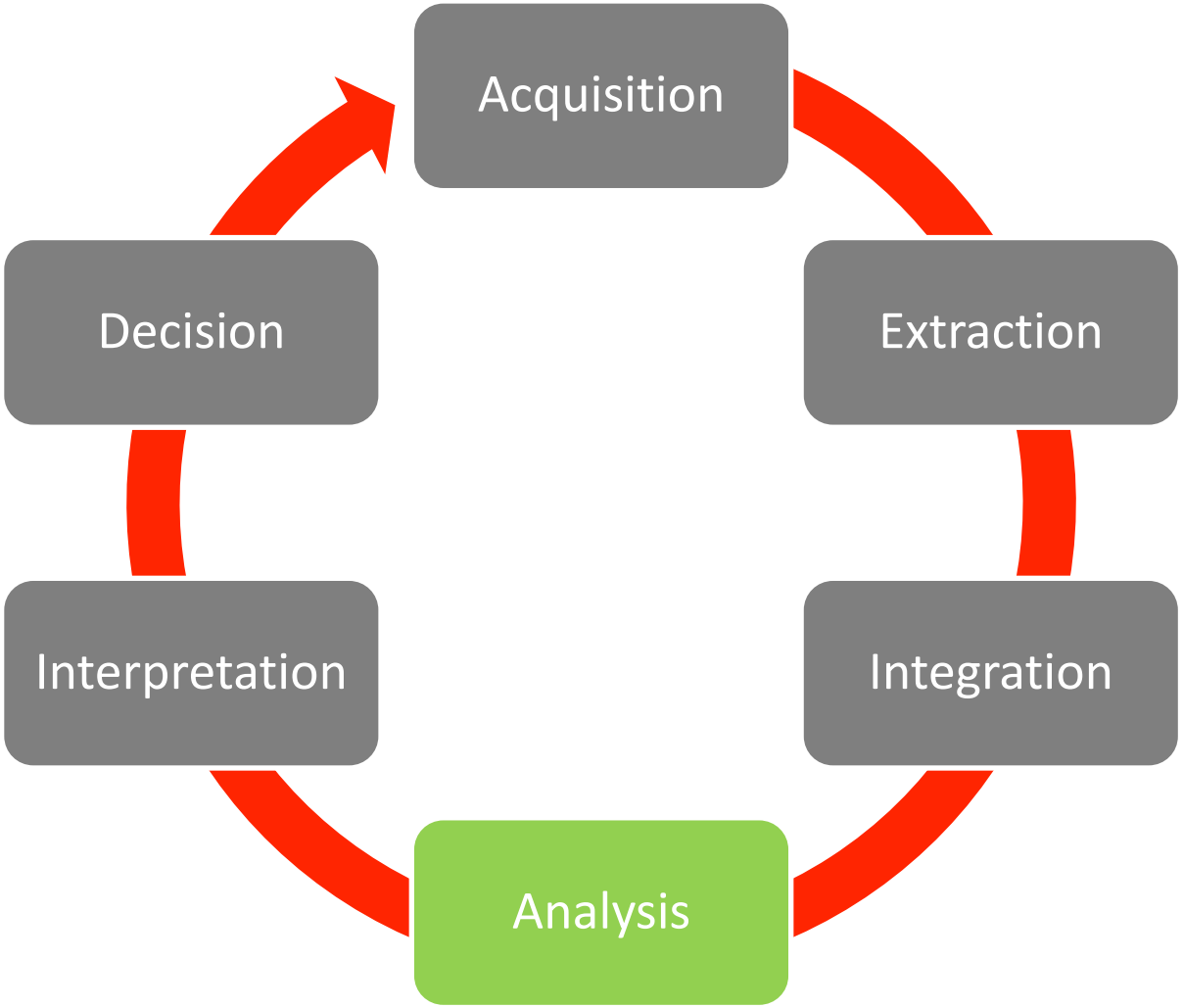
Big Data in action



Requires:

- standardization
- conflict management
- reconciliation
- mapping definition

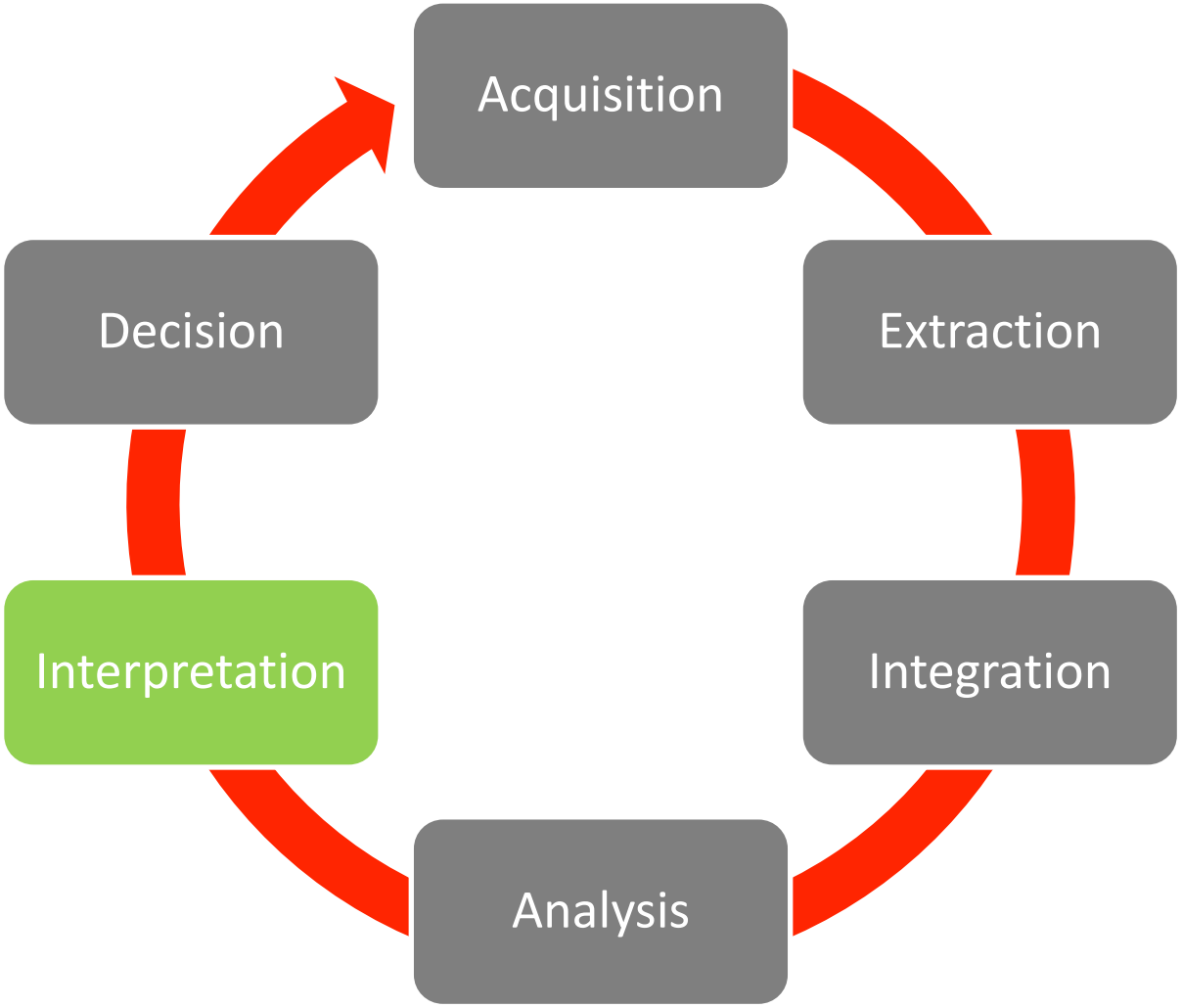
Big Data in action



Requires:

- exploration
- data mining
- machine learning
- visualization

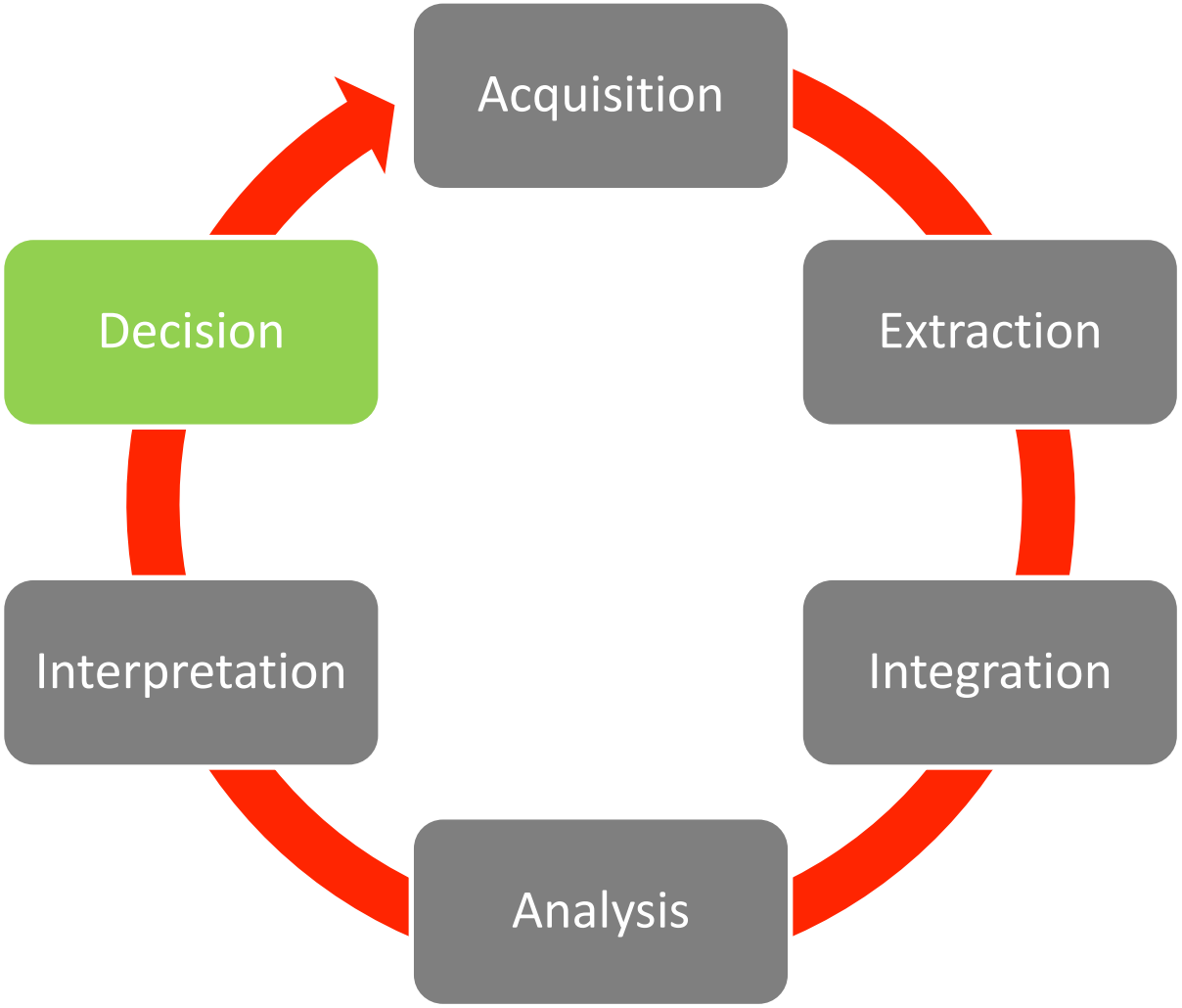
Big Data in action



Requires:

- knowledge of the domain
- knowledge of the provenance
- identification of patterns of interest
- flexibility of the process

Big Data in action



Requires:

- managerial skills
- continuous improvement of the process

Challenges

- Performance, performance, performance!
- Scalability
- Heterogeneity
- Effectiveness
- Flexibility
- Privacy
- Property
- Human collaboration

The Big Data flow

Real Time Streams

Real-Time Processing

Near Real-Time Processing

ETL

OLAP

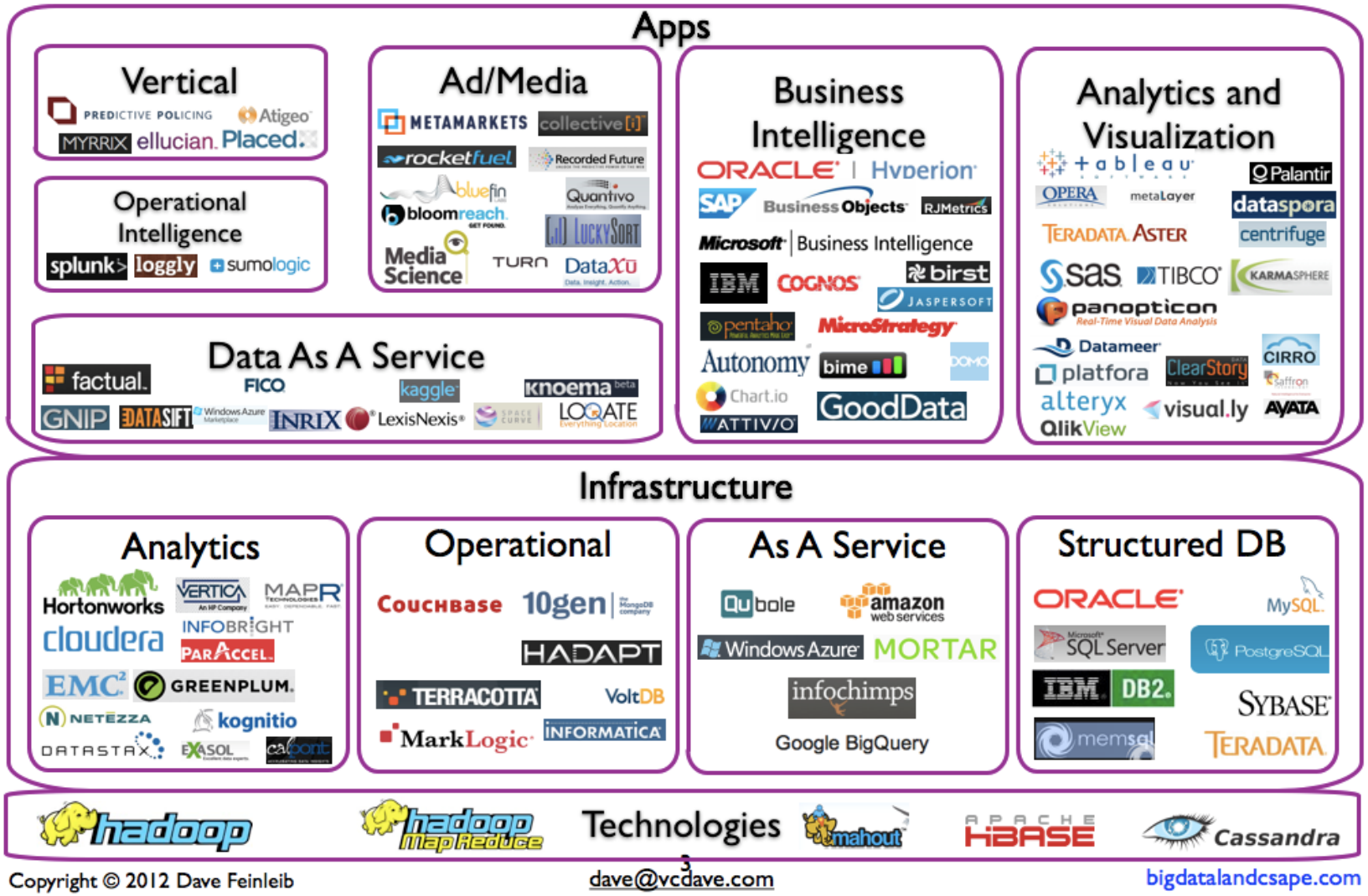
NoSQL (HBase, Cassandra, MongoDB)

Big SQL (Oracle, InfoSphere, Teradata)

Analytics (Vertica, Penthao, Greenplum)

Distributed file system (HDFS)

The Big Data Landscape

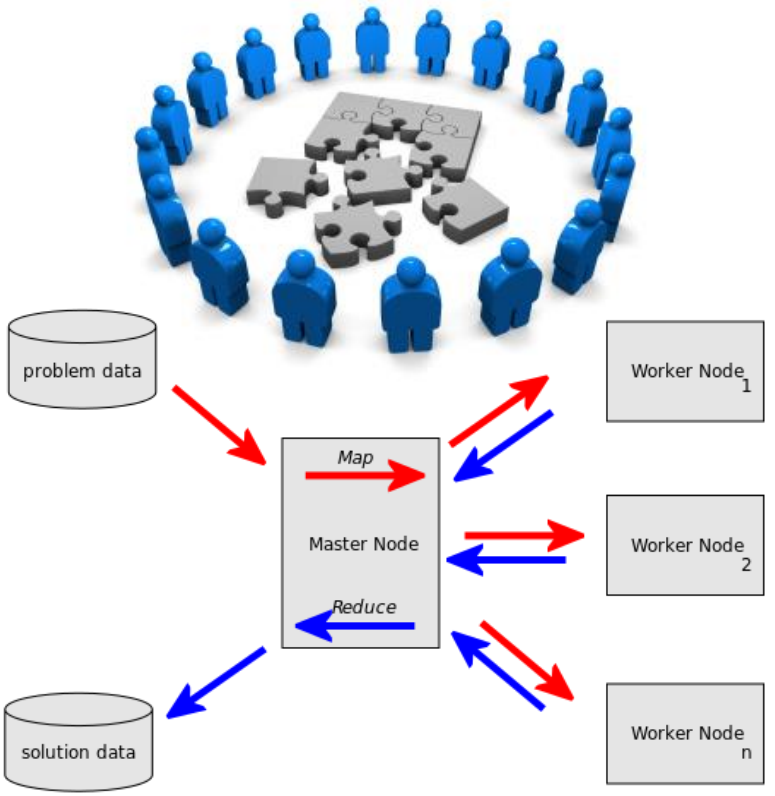


dave@vcdave.com

bigdatalandcsape.com

Distribution of resources and services

- Distributed Architecture
  - Clusters of computers that work together to a common goal
  - Scale out not up!
- Fault-tolerance
  - Resource replication
  - Eventual consistency
- Distributed processing
  - Shared-nothing model
  - Map-Reduce paradigm



Technology: Hadoop & MapReduce

- What is Hadoop?
  - An open-source software framework (Apache project)
  - Originally developed by Yahoo!
- Goal: storage and processing of data-sets at massive scale
- Infrastructure: clusters of commodity hardware
- Core:
  - HDFS, a distributed file system
  - MapReduce, a programming model for large scale data processing
- Includes a number of related projects
  - Apache Pig, Apache Hive, Apache HBase, etc..
- Used in production by Google, Facebook, Yahoo! and many others

The core of Hadoop

- HDFS
  - A distributed file systems
  - Servers can fail and not abort the computation process
  - Data is replicated with redundancy across the cluster
- MapReduce
  - Programming paradigm for expressing distributed computations over multiple servers
  - The powerhouse behind most of today's big data processing
  - Also used in other MPP environments and NoSQL databases (e.g., Vertica and MongoDB)
- Improving programmability: Pig and Hive
- Improving data access: HBase, Sqoop, and Flume

Hadoop: some History

- 2003: Google publishes about its cluster architecture & distributed file system (GFS)
- 2004: Google publishes about its MapReduce programming model used on top of GFS
  - written in C++
  - closed-source, Python and Java APIs available to Google programmers only
- 2006: Apache & Yahoo! → Hadoop & HDFS (Doug Cutting and Mike Cafarella)
  - open-source, Java implementations of Google MapReduce and GFS
  - a diverse set of APIs available to public
- 2008: becomes an independent Apache project
  - Yahoo! uses Hadoop in production
- Today: used as a general-purpose storage and analysis platform for big data
  - other Hadoop distributions from several vendors including EMC, IBM, Microsoft, Oracle, Cloudera, etc.
  - many users (http://wiki.apache.org/hadoop/PoweredBy)
  - research and development actively continues…

What is MapReduce?

- Programming model for expressing distributed computations at a massive scale
- Execution framework for organizing and performing such computations
  - running the various tasks in parallel
  - providing for redundancy and fault tolerance
- Inspired by the map and reduce functions commonly used in functional programming
- Various implementations:
  - Google
  - Hadoop
  - …
- Google has been granted a patent on MapReduce

The good news

- "More data usually beats better algorithms!"
  Anand Rajaman (about the Netflix Challenge)
  But also: Alon Halevy, Peter Norvig,
  Fernando Pereira, ...

Google YAHOO! facebook

WIKIPEDIA NASA CERN INTERNET ARCHIVE

The bad news

- The storage capacities of hard drives have increased massively over the years
- But access speeds have not kept up:

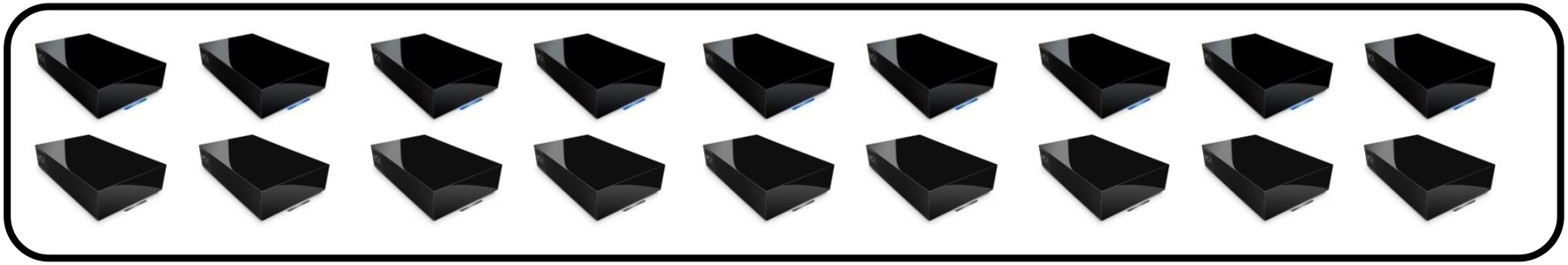
year: 1990
size: ~1.3GB
speed: 4.4 MB/s

5 mins

year: 2011
size: ~1TB
speed: 100 MB/s

2.5 hours!

Today: cluster computing

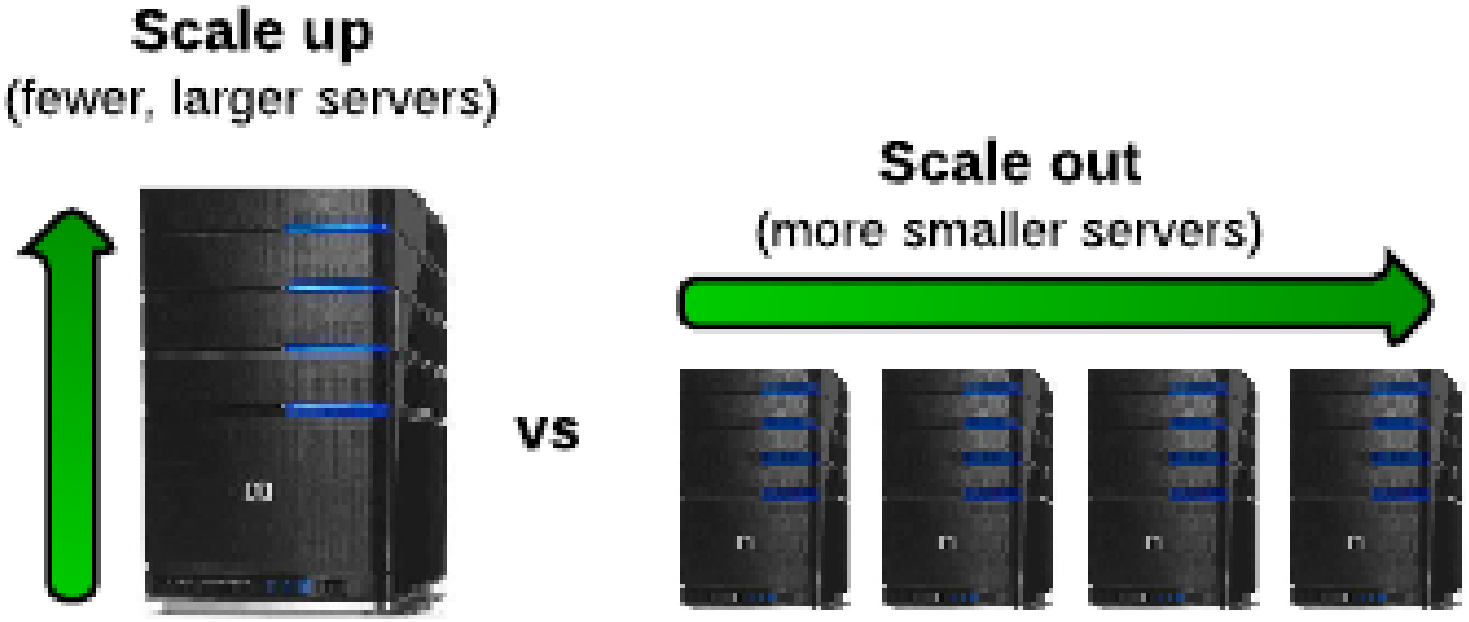


- 100 hard disks? 2 mins to read 1TB
- What about disk failures?
- Replication (RAID) ... or

Scale up

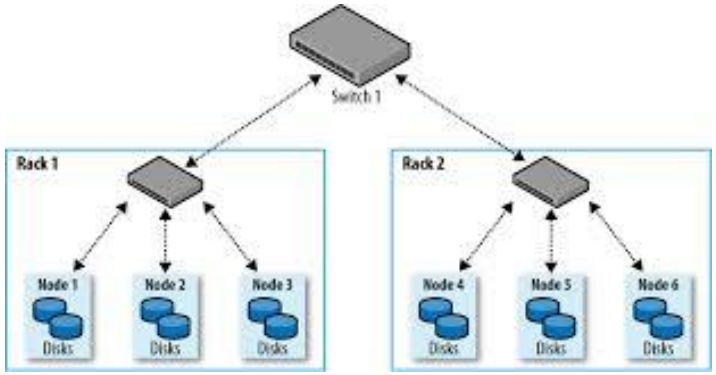
Scale out

Scale up vs scale out



Cluster computing

- Compute nodes are stored on racks
  - 8–64 compute nodes on a rack
- There can be many racks of compute nodes
- The nodes on a single rack are connected by a network
  - typically gigabit Ethernet
- Racks are connected by another level of network or a switch
- The bandwidth of intra-rack communication is usually much greater than that of inter-rack communication
- Compute nodes can fail! Solution:
  - Files are stored redundantly
  - Computations are divided into tasks



Commodity hardware

- You are not tied to expensive, proprietary offerings from a single vendor
- You can choose standardized, commonly available hardware from any of a large range of vendors to build your cluster
- Commodity ≠ Low-end!
  - cheap components with high failure rate can be a false economy
  - but expensive database class machines do not score well on the price/performance curve
- Example typical specifications of commodity hardware:
  - Processor 2 quad-core 2-2.5GHz CPUs
  - Memory 16-24 GB ECC RAM
  - Storage 4 × 1TB SATA disks
  - Network Gigabit Ethernet
- Yahoo! has a huge installation of Hadoop:
  - > 100,000 CPUs in > 36,000 computers
  - Used to support research for Ad Systems and Web Search
  - Also used to do scaling tests to support development of Hadoop

The New Software Stack

- New programming environments designed to get their parallelism not from a supercomputer but from computing clusters
- Bottom of the stack: distributed file system (DFS)
- On the top of a DFS:
  - many different high-level programming systems
  - We have a winner!
  - MapReduce

DFS: Assumptions

- Commodity hardware
  - Scale "out", not "up"
- Significant failure rates
  - Nodes can fail over time
- "Modest" number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

DFS: organization

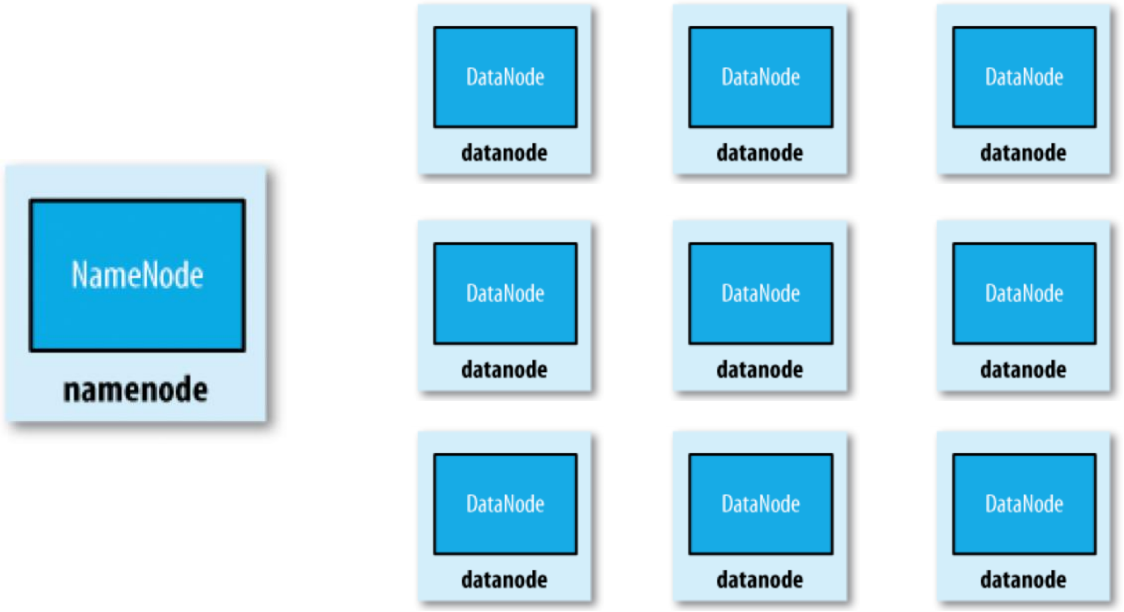
- Files are divided into chunks
  - typically 64 megabytes in size
- Chunks are replicated at different compute nodes (usually 3+)
- Nodes holding copies of one chunk are located on different racks
- Chunk size and the degree of replication can be decided by the user
- A special file (the master node) stores, for each file, the positions of its chunks
- The master node is itself replicated
- A directory for the file system knows where to find the master node
- The directory itself can be replicated
- All participants using the DFS know where the directory copies are

DFS implementations

- Several distributed file systems used in practice
- Among these:
  - The Google File System (GFS), the original of the class
  - CloudStore, an open-source DFS originally developed by Kosmix
  - Hadoop Distributed File System (HDFS), an open-source DFS used with Hadoop
    - Master node = Namenode
    - Compute node = Datanode
    - Node: both physical and logical entity

HDFS concepts

- An HDFS cluster has two types of nodes:
  - Multiple DataNodes
  - The NameNode



HDFS concepts

- The datanodes just store and retrieve the blocks when they are told to (by clients or the namenode)
- The namenode:
  - Manages the filesystem tree and the metadata for all the files and directories
  - Knows the datanodes on which all the blocks for a given file are located
- Without the namenode HDFS cannot be used
- It is important to make the namenode resilient to failure

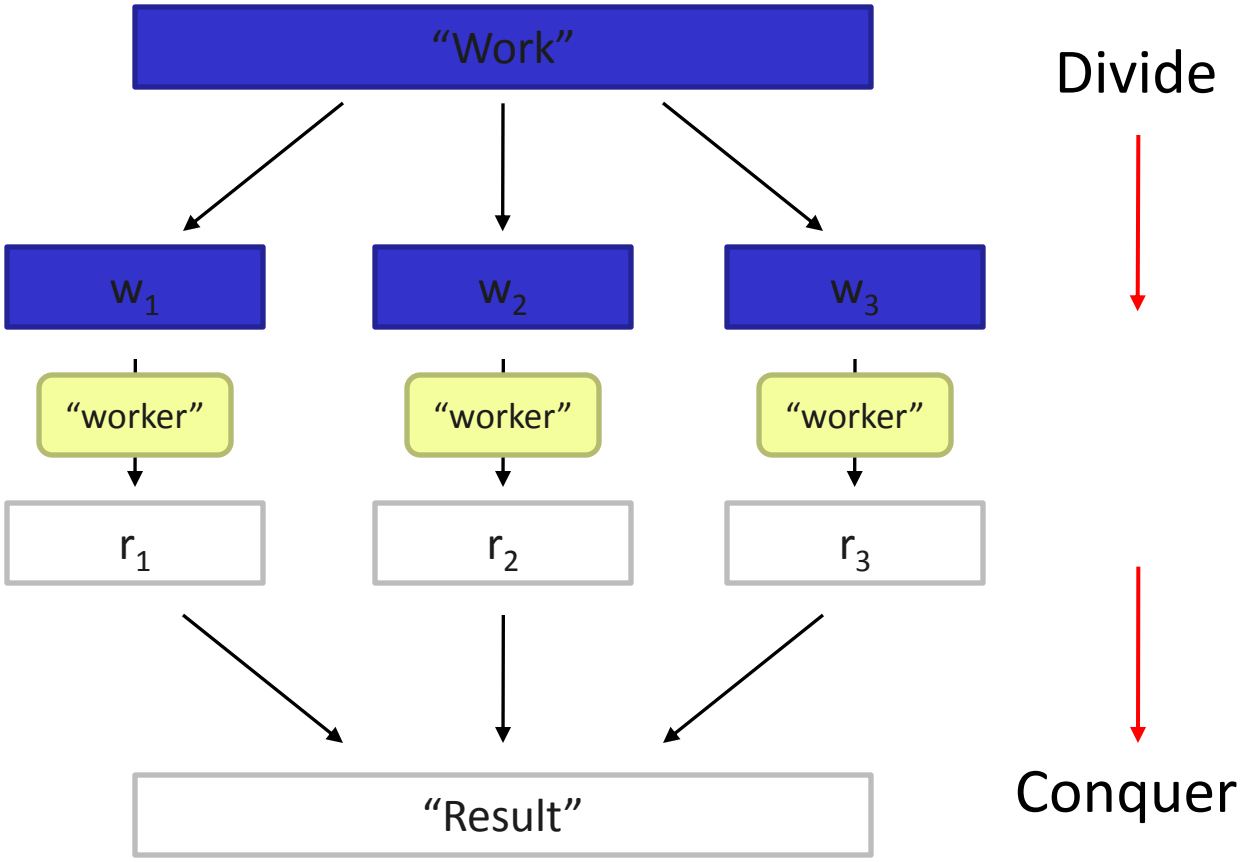
HDFS I/O

- An application client wishing to read a file (or a portion thereof) must first contact the namenode to determine where the actual data is stored
- In response to the client request the namenode returns:
  - the relevant block id
  - the location where the block is held (i.e., which datanode)
- The client then contacts the datanode to retrieve the data.
- Blocks are themselves stored on standard single-machine file systems
  - HDFS lies on top of the standard OS stack
- Important feature of the design:
  - data is never moved through the namenode
  - all data transfer occurs directly between clients and datanodes
  - communications with the namenode only involves transfer of metadata

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Distributed computing



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the risk of all of these problems?

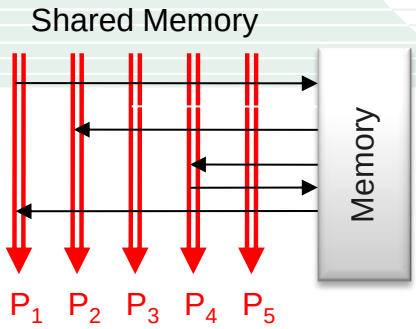
Source: Ricardo Guimarães Herrmann

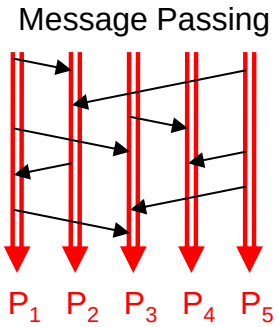
Risk?

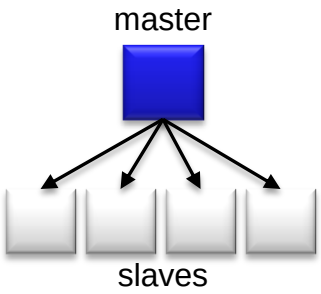
- A lot: for instance, deadlock and starvation
- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

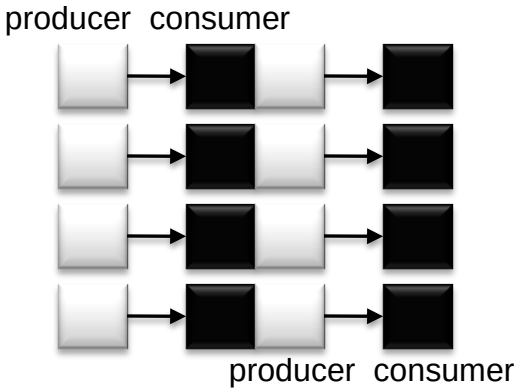
Current approaches

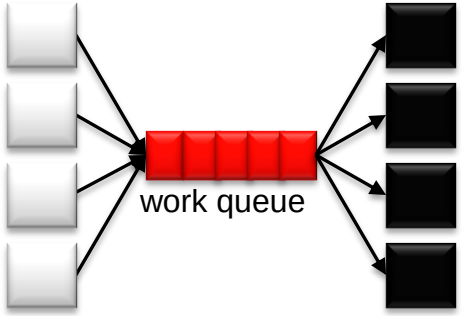
- Programming models
  - Shared memory (pthreads)
  - Shared nothing (Message passing)
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues











So, what?

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters (even across datacenters)
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

What's the point?

- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the what from how
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

The datacenter *is* the computer!

“Big Ideas” of large scale computing

- Scale “out”, not “up”
  - Limits of Symmetric Multi-Processing and large shared-memory machines
- Hide system-level details from the application developer
  - Concurrent programs are difficult to reason about and harder to debug
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

Typical Large-Data Problem

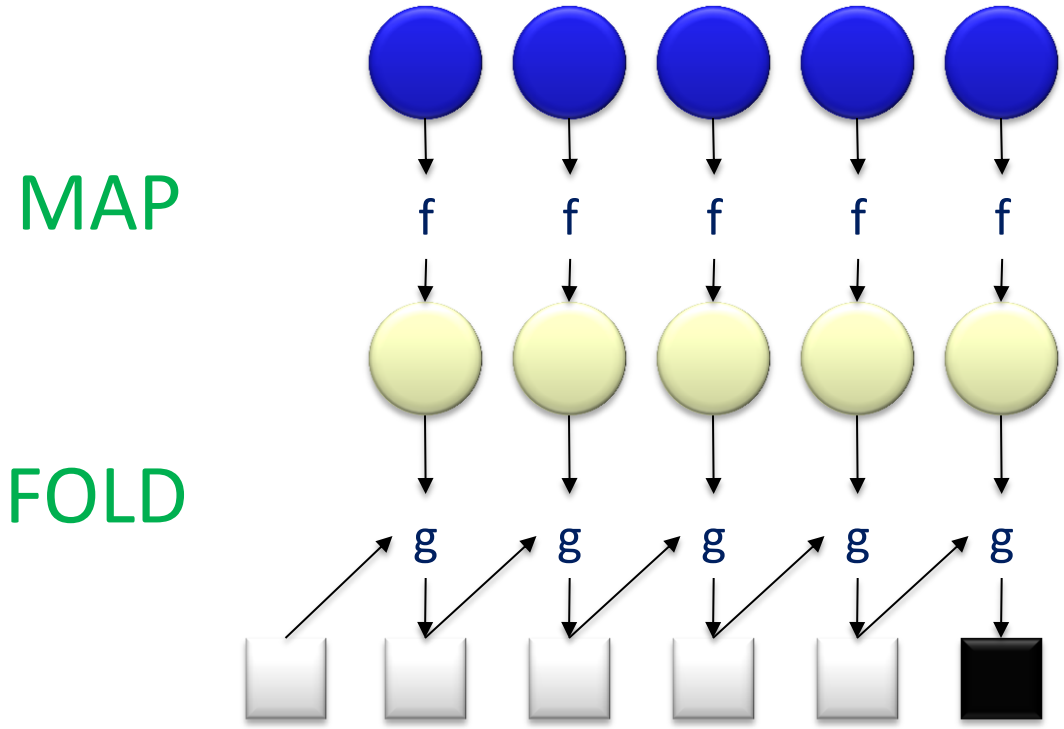
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

Key idea: provide a functional abstraction for these two operations

Roots in Functional Programming



- MAP takes a function f and applies it to every element in a list,
- FOLD iteratively applies a function g to aggregate results

Roots in Functional Programming

- The application of f to each item in a list can be parallelized in a straightforward manner, since each functional application happens in isolation
  - in a cluster, these operations can be distributed across many different machines
- The fold operation has more restrictions on data locality
  - elements in the list must be "brought together" before the function g can be applied
- However, many real-world applications do not require g to be applied to all elements of the list
- If elements in the list can be divided into groups, the fold aggregations can proceed in parallel

MapReduce

- Basic data structure: key-value pairs
- Programmers specify two functions:
  - **map** (k1, v1) → [(k2, v2)]
  - **reduce** (k1, [v1]) → [(k2, v2)]
    - (k, v) denotes a (key, value) pair
    - […] denotes a list
    - keys do not have to be unique: different pairs can have the same key
    - normally the keys of input elements are not relevant
- The execution framework handles everything else!

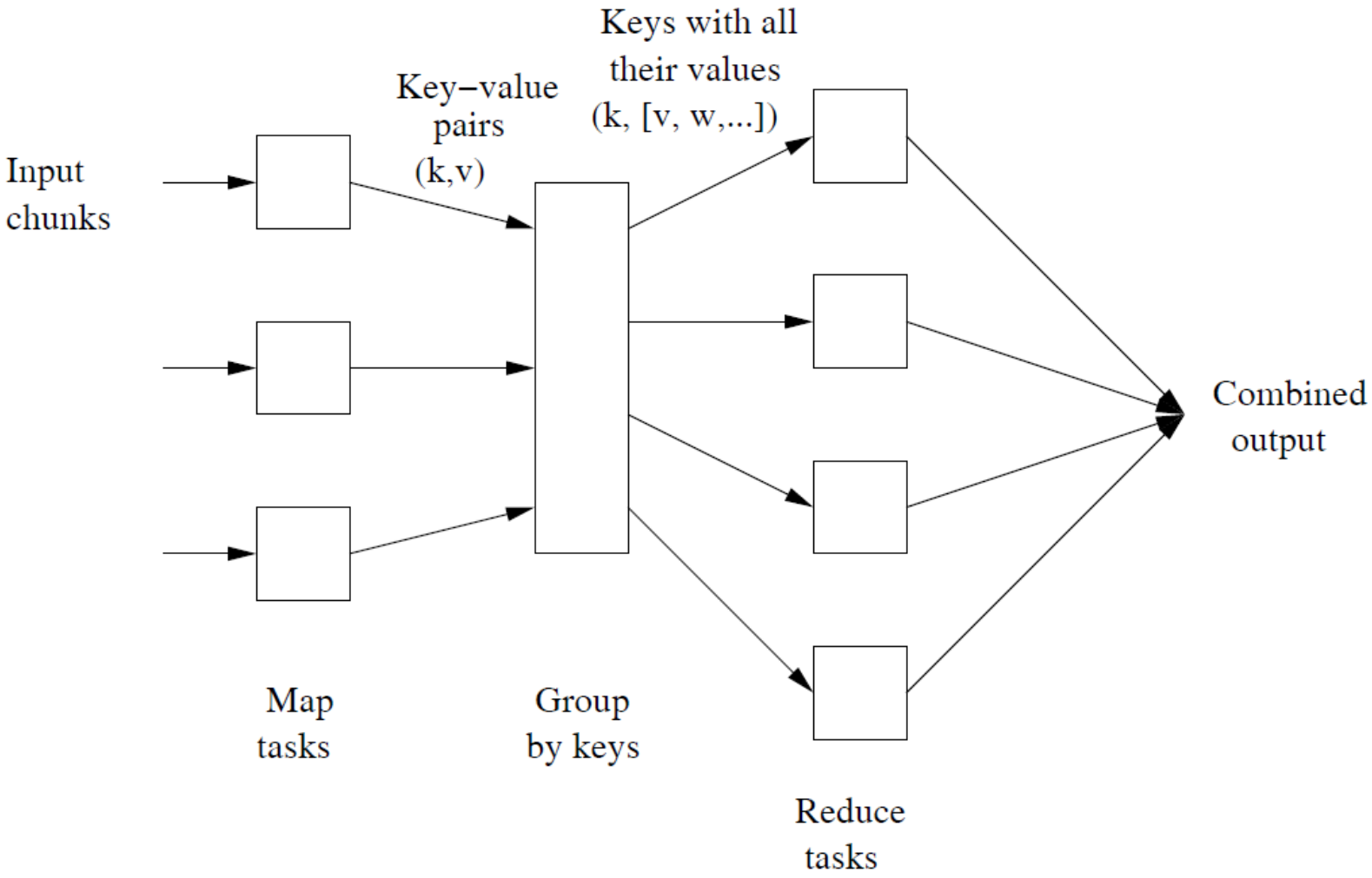
MapReduce program

- A MapReduce program, referred to as a job, consists of:
  - code for Map and Reduce packaged together
  - configuration parameters (where the input lies, where the output should be stored)
  - the input, stored on the underlying distributed file system
- Each MapReduce job is divided by the system into smaller units called tasks
  - Map tasks
  - Reduce tasks
- The output of MapReduce job is also stored on the underlying distributed file system

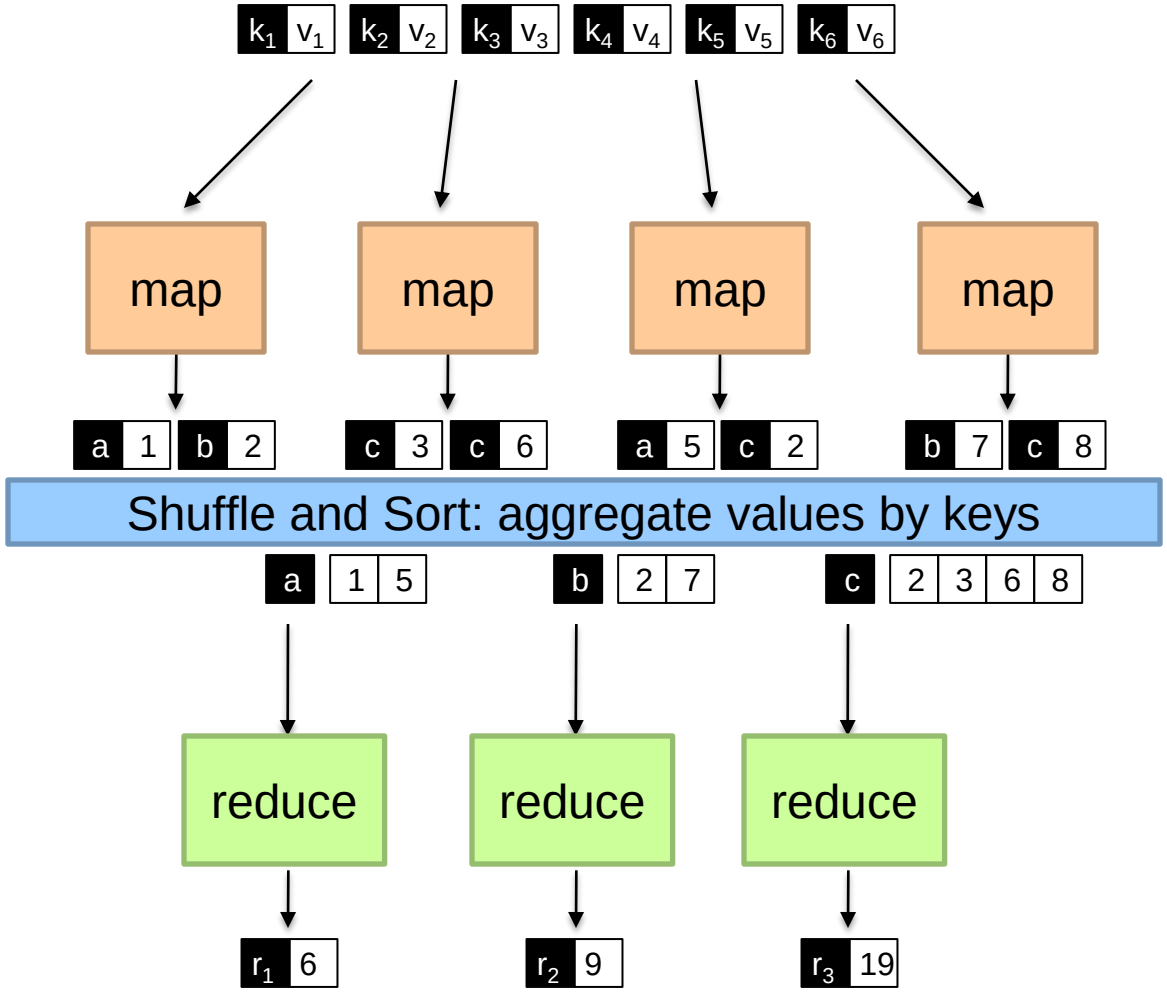
MapReduce process

- Some number of Map tasks each are given one or more chunks of data
- These Map tasks turn the chunk into a sequence of key-value pairs
  - The way key-value pairs are produced is determined by the code written by the user for the Map function
- The key-value pairs from each Map task are collected by a master controller and sorted and grouped by key (Shuffle and sort)
- The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task
- The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way
  - The way values are combined is determined by the code written by the user for the Reduce function
- Output key-value pairs from each reducer are written persistently back onto the distributed file system
- The output ends up in r files, where r is the number of reducers
  - the r files often serve as input to yet another MapReduce job

MapReduce process



An example


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 6
r2 9
r3 19

Example: Word Count

- **Problem**: counting the number of occurrences for each word in a collection of documents
- **Input**: a repository of documents, each document is an element
- **Map**: reads a document and emits a sequence of key-value pairs where keys are words of the documents and values are equal to 1:

  (w1, 1), (w2, 1), . . . , (wn, 1)

- **Grouping**: groups by key and generates pairs of the form

  (w1, [1, 1, . . . , 1]) , . . . , (wn, [1, 1, . . . , 1])

- **Reduce**: adds up all the values and emits:

  (w1, k) , . . . , (wn, l)

- **Output**: (w,m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents

Implementation

```
Map(String docid, String text):
     for each word w in text:
          Emit(w, 1);

Reduce(String term, counts[]):
     int sum = 0;
     for each c in counts:
          sum += c;
     Emit(term, sum);
```

A Map in Java

```
public static class Map extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable uno = new IntWritable(1);
    private Text parola = new Text();
    public void map(LongWritable chiave, Text testo,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String linea = testo.toString();
      StringTokenizer tokenizer = new StringTokenizer(linea);
      while (tokenizer.hasMoreTokens()) {
        parola.set(tokenizer.nextToken());
        output.collect(parola, uno);
      }
    }
}
```

A Reduce in Java

```
public static class Reduce extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(
         Text chiave,
         Iterator<IntWritable> valori,
         OutputCollector<Text, IntWritable> output,
         Reporter reporter)
    throws IOException {
      int somma = 0;
      while (valori.hasNext()) {
        somma += valori.next().get();
      }
      output.collect(chiave, new IntWritable(somma));
  }
}
```

Another example: Word Length Count

- **Problem**: counting how many words of certain lengths exist in a collection of documents
- **Input**: a repository of documents, each document is an element
- **Map**: reads a document and emits a sequence of key-value pairs where the key is the length of a word and the value is the word itself:

  (i,w1), . . . , (j,wn)

- **Grouping**: groups by key and generates pairs of the form

  (1, [w1, . . . , wk]) , . . . , (n, [wr, . . . , ws])

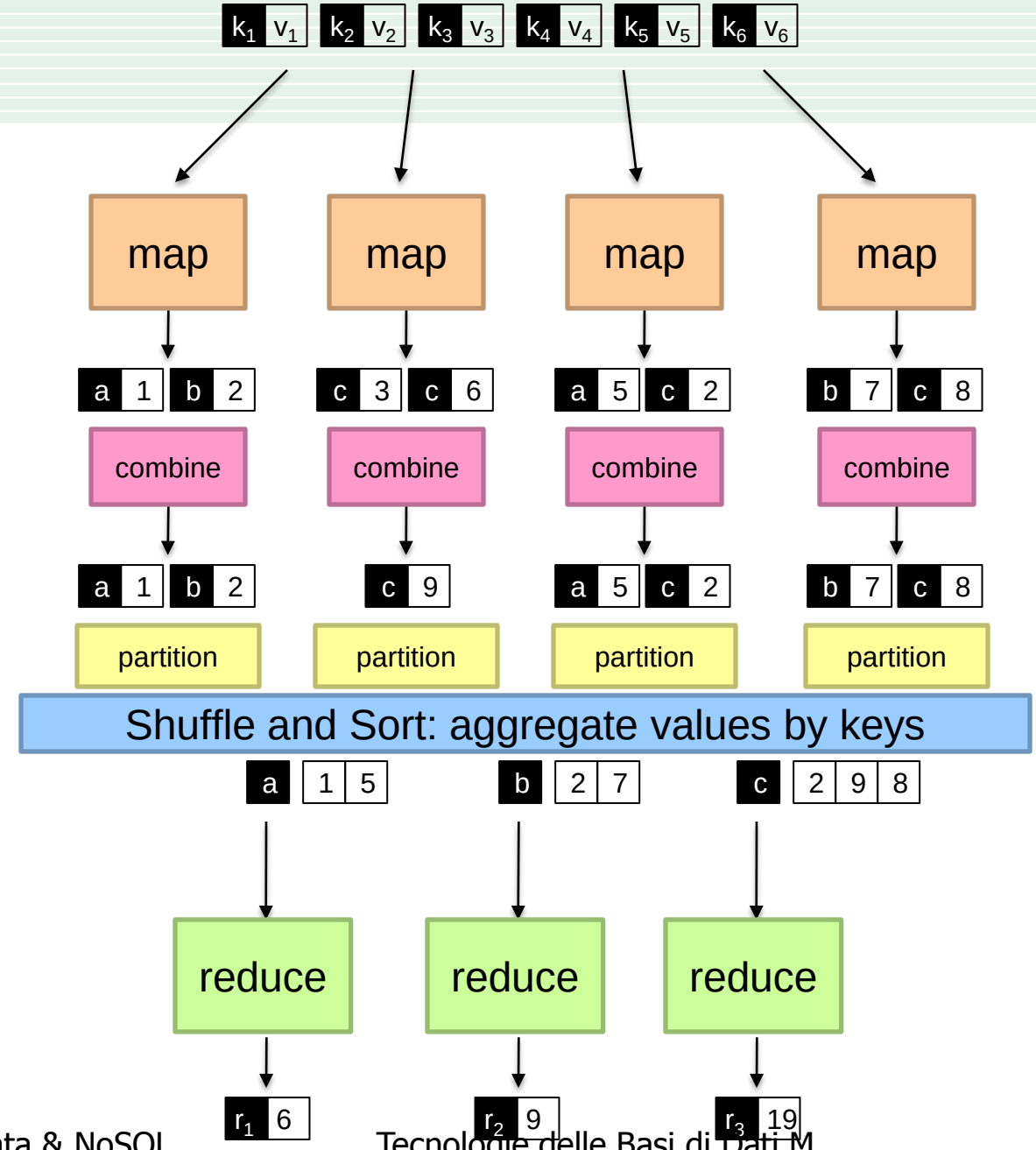
- **Reduce**: counts the number of words in each list and emits:

  (1, l) , . . . , (p, m)

- **Output**: (l,n) pairs, where l is a length and n is the total number of words of length l in the input documents

Introducing combiners

- When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks
- In this case we also apply a combiner to the Map function
- Grouping is still necessary!
- Advantages:
  - it reduces the amount of intermediate data
  - it reduces the network traffic


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r1 6
r2 9
r3 19

Example: Word Count with combiners

- **Input**: a repository of documents, each document is an element
- **Map**: reads a document and emits a sequence of key-value pairs where keys are words of the documents and values are equal to 1:

  (w1, 1), . . . , (wn, 1)

- **Combiner**: groups by key, adds up all the values and emits:

  (w1, i), . . . , (wn, j)

- **Grouping**: groups by key and generates pairs of the form

  (w1, [p, . . . , q]) , . . . , (wn, [r, . . . , s])

- **Reduce**: adds up all the values and emits:

  (w1, k) , . . . , (wn, l)

- **Output**: (w,m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents.

Introducing partitioners

- We can also specify a partitioner that:
  - divides up the intermediate key space
  - assigns intermediate key-value pairs to reducers
  - *n* partitions $\rightarrow$ *n* reducers
- The simplest partitioner assigns approximately the same number of keys to each reducer.
- But partitioner only considers the key and ignores the value
- An imbalance in the amount of data associated with each key is relatively common in many text processing applications
  - In texts the frequency of any word is inversely proportional to its rank in the frequency table
  - The most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

Partitioners

- The user tells the map-reduce system how many Reduce tasks there will be, say r.
- The master controller picks a hash function that applies to keys and produces a bucket number from 0 to r − 1.
- Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files
- Local files are organized as a sequence of (key,list-of-values) pairs
- Each file is destined for one of the Reduce tasks.
- Optionally, users can specify their own hash function or other method for assigning keys to Reduce tasks.
- However, whatever algorithm is used, each key is assigned to one and only one Reduce task.

Reduce Tasks, Compute Nodes, and Skew

- If we want maximum parallelism, then we could use one Reduce task to execute each single key and its associated value list and execute each Reduce task at a different compute node, so they would all execute in parallel.
- Skew:
  - there is often significant variation in the lengths of the value lists for different keys
  - different reducers take different amounts of time
  - significant difference in the amount of time each reduce task takes.
- Overhead and physical limitations:
  - There is overhead associated with each task we create
  - Often there are far more keys than there are compute nodes available.
- We can reduce the impact of skew by using fewer Reduce tasks.
- If keys are sent randomly to Reduce tasks, we can expect that there will be some averaging of the total time required by the different Reduce tasks.
- We can further reduce the skew by using more Reduce tasks than compute nodes. In that way, long Reduce tasks might occupy a compute node fully, while several shorter Reduce tasks might run sequentially at a single compute node.

MapReduce: the complete picture

- Programmers specify two functions:

  map (k1, v1) → [(k2, v2)]

  reduce (k2, [v2]) → [(k3, v3)]

  - All values with the same key are reduced together
- Usually, programmers also specify:

  combine (k2, [v2]) → [(k3, v3)]

  - Mini-reducers that run after the map phase
  - Used as an optimization to reduce network traffic

  partition (k2, number of partitions) → partition for k2

  - Divides up key space for parallel reduce operations
- The execution framework handles everything else…

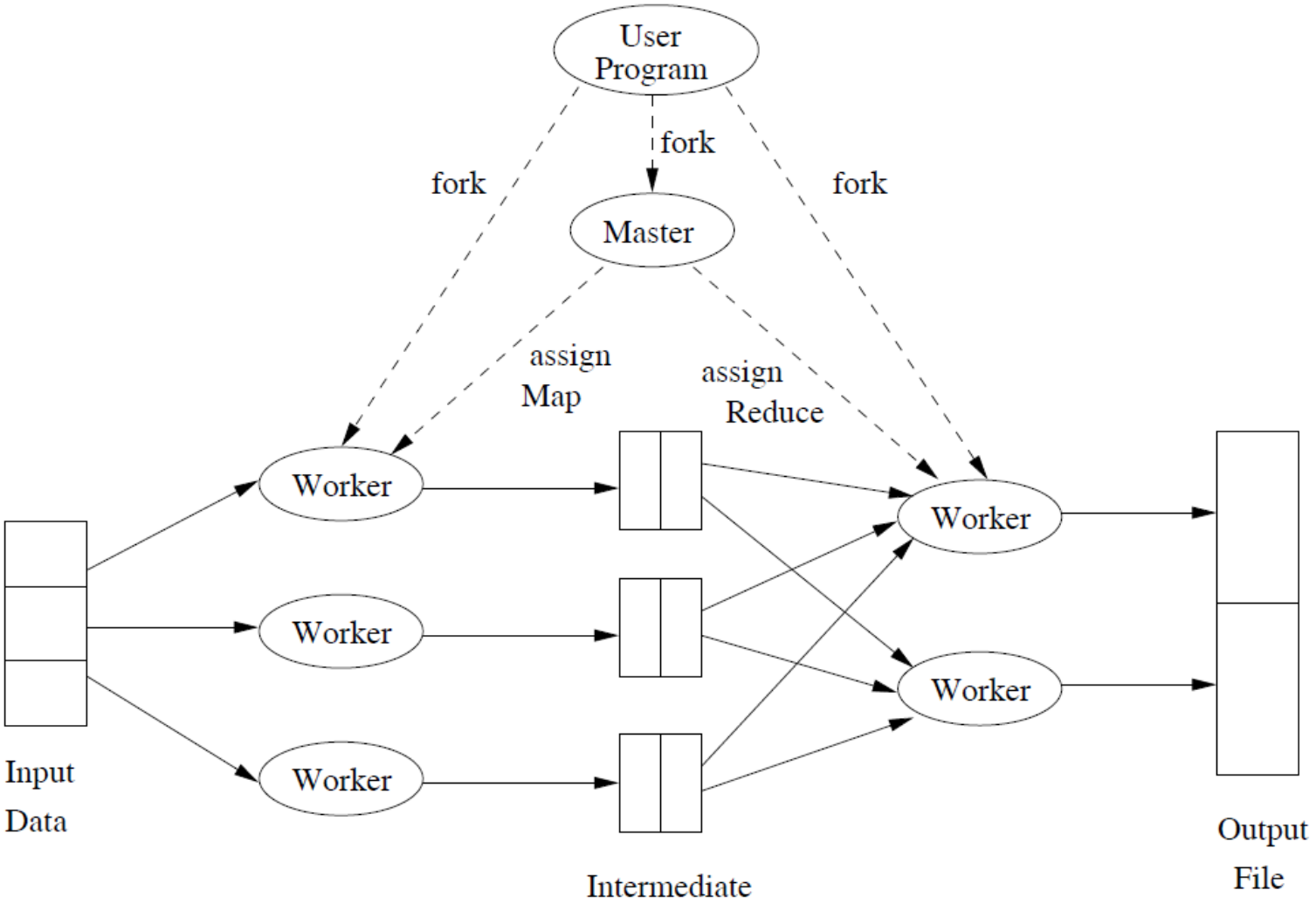
MapReduce runtime

- Important idea behind MapReduce is separating the what of distributed processing from the how
- The developer submits the job to the submission node of a cluster
- The execution framework (the "runtime") takes care of everything else:
  - it transparently handles all aspects of distributed code execution
  - on clusters ranging from a single node to a few thousand nodes

MapReduce at work

- Master-slave architecture
- Taking advantage of a library provided by a map-reduce system such as Hadoop, the user program generates:
  - a Master controller process (the jobtracker in Hadoop)
  - some number of Worker processes at different compute nodes (the tasktrackers in Hadoop).
- The jobtracker process coordinates all the jobs run on the system by scheduling tasks
- The tasktrackers processes run tasks and send progress reports to the jobtracker
- If a task fails, the jobtracker can reschedule it on a different tasktracker

Map-Reduce Execution



MapReduce execution

- Normally, a Worker handles either Map tasks (a Map worker) or Reduce tasks (a Reduce worker), but not both
- The Master:
  - creates a number of Map tasks and a number of Reduce tasks, as selected by the user program.
  - assigns tasks to Workers
  - keeps track of the status of each Map and Reduce task (idle, executing at a particular Worker, completed).
- The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task and informs the Master of the location and sizes of each of these files
- When a Reduce task is assigned by the Master to a Worker, that task is given all the files that form its input
- The Reduce task writes its output to a file of the distributed file system

MapReduce Job Run

- Four entities are involved
  - The client, which submits the MapReduce job
  - The jobtracker, which coordinates the job run
  - The tasktrackers, which run the tasks that the job has been split into
  - The distributed file system

MapReduce "Runtime"

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Get data to the workers
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS

Scheduling

- It is not uncommon for MapReduce jobs to have thousands of individual tasks that need to be assigned to nodes in the cluster
- In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently, making it necessary for the scheduler to maintain some sort of a task queue and to track the progress of running tasks so that waiting tasks can be assigned to nodes as they become available

How do we get data to the workers?

What's the problem here?

MapReduce Principles

- Data locality:
  - Data and workers must be close to each other
- Shared nothing architecture
  - Each node is independent and self-sufficient

Locality enforcement

- Don’t move data to workers… move workers to the data!
- Store data on the local disks of nodes in the cluster
- Start up the workers on the node that has the data local
- The tasks are created from the input splits in the shared file system
  - 1 map per split + N reduces determined by the configuration
- If this is not possible (e.g., a node is already running too many tasks)
  - new tasks will be started elsewhere
  - the necessary data will be streamed over the network
- Optimization
  - prefer nodes that are on the same rack in the datacenter as the node holding the relevant data block
  - inter-rack bandwidth is significantly less than intra-rack bandwidth!

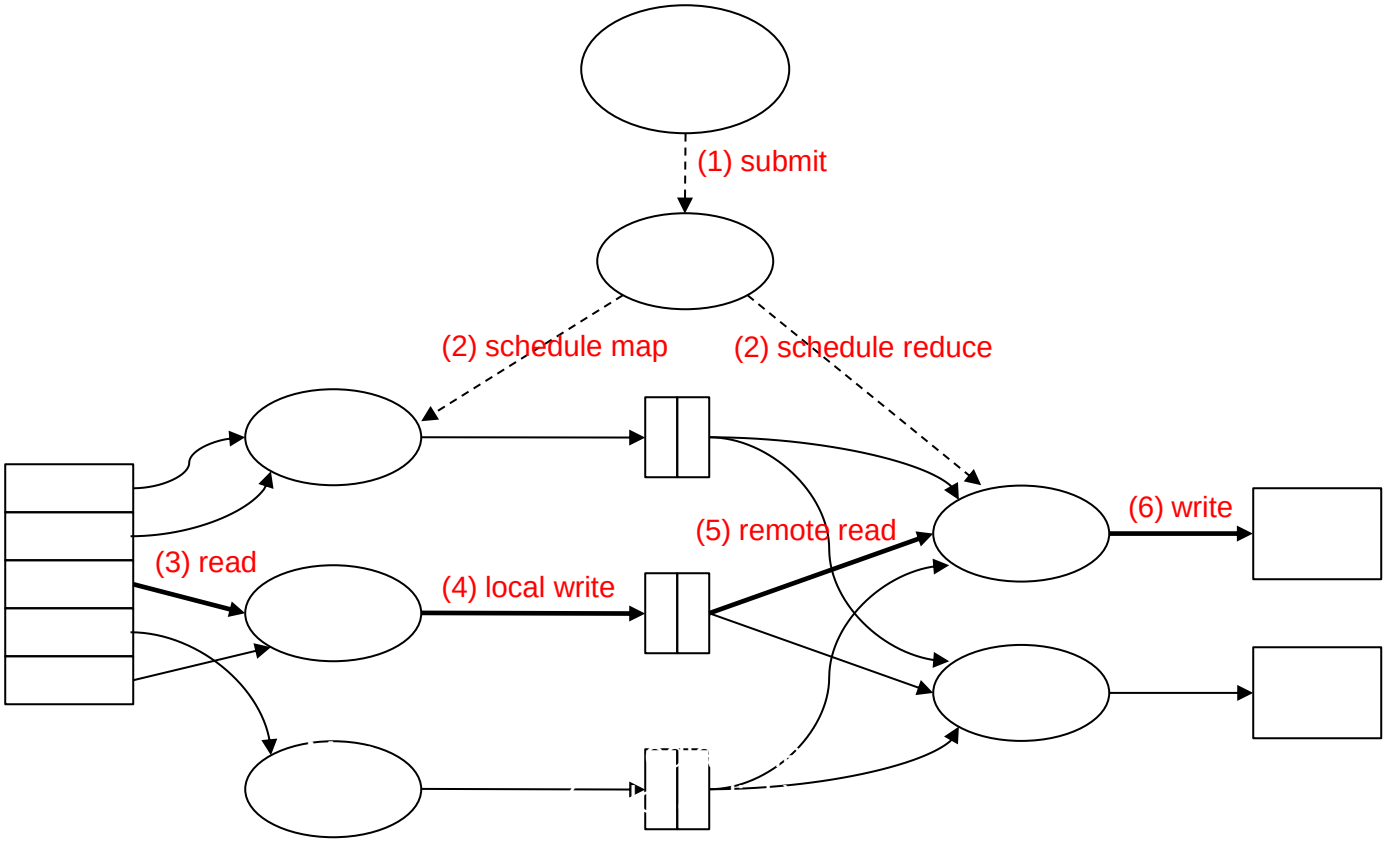
Shared-nothing architecture

- Coordinating the process in a distributed computation is hard
  - How to distribute the load over the available nodes?
  - How to serialize the data for the transmission?
  - How to handle an unresponsive process?
- MapReduce's shared-nothing architecture means that tasks have no dependence on one other
- Programmers do not worry about the distributed computation issues
- They need only to write two the Map and Reduce functions

Synchronization

- In general, synchronization refers to the mechanisms by which multiple concurrently running processes “join up”, for example, to share intermediate results or otherwise exchange state information
- In MapReduce, synchronization is accomplished by a “barrier” between the map and reduce phases of processing
- Intermediate key-value pairs must be grouped by key, which is accomplished by a large distributed sort involving all the nodes that executed map tasks and all the nodes that will execute reduce tasks
- This necessarily involves copying intermediate data over the network, and therefore the process is commonly known as “shuffle and sort”
- A MapReduce job with *m* mappers and *r* reducers involves up to *m x r* distinct copy operations, since each mapper may have intermediate output going to every reducer

MapReduce Execution



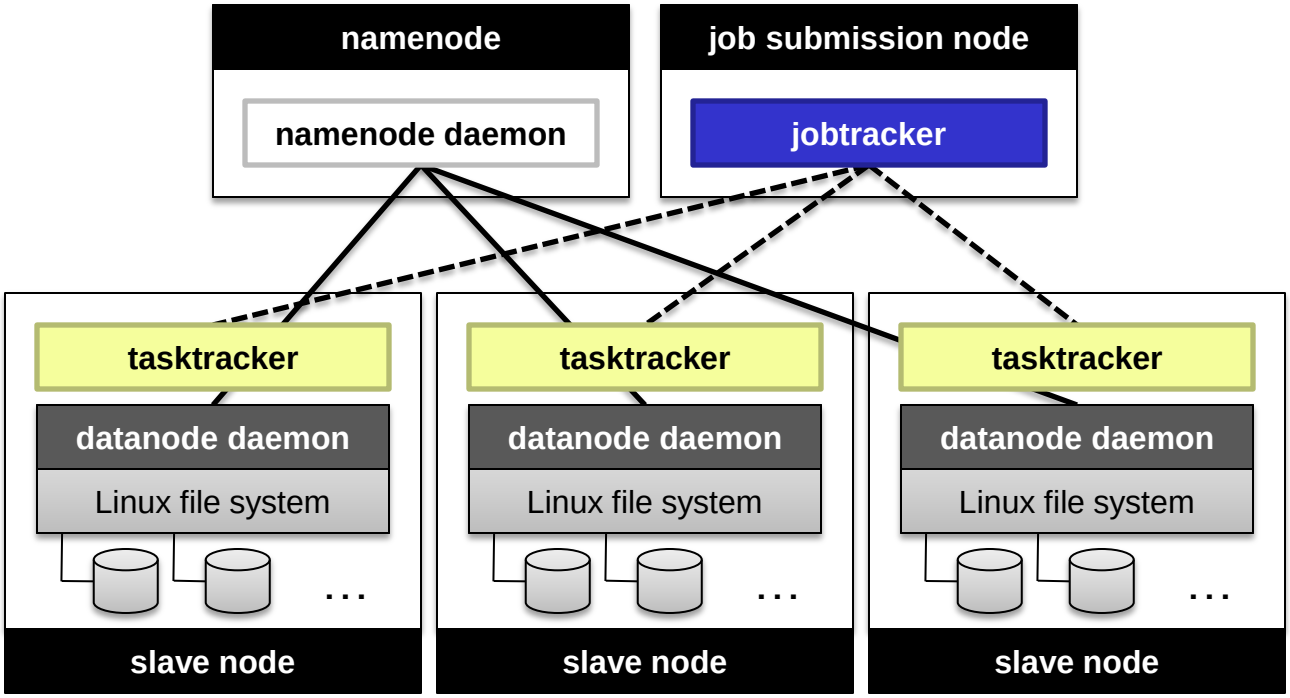
Coping With Node Failures

- The worst thing that can happen is that the compute node at which the Master is executing fails
  - In this case, the entire map-reduce job must be restarted
- Other failures will be detected and managed by the Master, and the map-reduce job will complete eventually
- Failure of a worker
  - the tasks assigned to this Worker will have to be redone
  - the Master:
    - sets the status of each failed tasks to idle
    - reschedules them on a Worker when one becomes available

Error and fault handling

- The MapReduce execution framework must accomplish all the tasks above in a hostile environment, where errors and faults are the norm, not the exception
- Since MapReduce was explicitly designed around low-end commodity servers, the runtime must be especially resilient. In large clusters, disk failures are common and RAM experiences more errors than one might expect
- Datacenters suffer from both planned outages (e.g., system maintenance and hardware upgrades) and unexpected outages (e.g., power failure, connectivity loss, etc.)
- And that's just hardware. No software is bug-free…
- Furthermore, any sufficiently large dataset will contain corrupted data or records that are mangled beyond a programmer's imagination, resulting in errors that one would never think to check for or trap

Putting everything together…



Algorithms for MapReduce

- Need to take the algorithm and break it into filter/collect/aggregate steps
  - Filter/collect becomes part of the map function
  - Collect/aggregate becomes part of the reduce function
- Note that sometimes we may need multiple map/reduce stages – chains of maps and reduces
- MapReduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel!
- It makes sense only when:
  - files are very large and are rarely updated
  - we need to iterate over all the files to generate some interesting property of the data in those files
- Let's see some examples

Filtering algorithms

- Goal: Find lines/files/tuples with a particular characteristic
- Examples:
  - grep Web logs for requests to *dia.uniroma3.it/*
  - find in the Web logs the hostnames accessed by 192.168.127.1
  - locate all the files that contain the words 'Apple' and 'Jobs'
- Generally: map does most of the work, reduce may simply be the identity

Aggregation algorithms

- Goal: Compute the maximum, the sum, the average, ..., over a set of values
- Examples:
  - count the number of requests to *.dia.uniroma3.it/*
  - find the most popular domain
  - average the number of requests per page per Web site
- Often: map may be simple or the identity

Union, intersections and joins

- Goal: Intersect multiple different inputs on some shared values
  - values can be equal, or meet a certain predicate
- Examples:
  - find all documents with the words “big” and/or “data” given an inverted index
  - find all professors and students in common courses and return the pairs <professor,student> for those cases

- Map generates a pair (k,e) for each element e in the input
- Reduce generates a sequence of pair [(k1,r1),..,(kn,rn)] for each k,[e1,..,ek] in the input

Example: Relational Union and Intersection

- Union
  - Map: turn each input tuple t into a key-value pair (t, t)
  - Reduce: associated with each key t there will be either one or two values
    - Produce output (t, t) in either case

- Intersection
  - Map: Turn each input tuple t into a key-value pair (t, t)
  - Reduce: Associated with each key t there will be either one or two values
    - Produce output (t, t) only if there are two values

Example: Relational Join

- R(AB) JOIN S(BC)
- Map: for each tuple (a, b) of R, produce the pair (b, (R, a)), for each tuple (b, c) of S, produce the pair (b, (S, c))
- Reduce: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c), for each pair (R, a) and (S, c) in the list generate a pair (k,(a, b, c)).

- Example:
  - R(AB)={(a,b),(c,b)} JOIN S(BC)={(b,e),(f,g)}
  - MAP: (b,(R,a)),(b,(R,c)),(b,(S,e)),(f,(S,g))
  - SHUFFLE AND SORT: (b,[(R,a),(R,c),(S,e)]) (f,[(S,g)])
  - REDUCE: (k1,(a,b,e)),(k2,(c,b,e))

Sorting

- Goal: Sort input
- Examples:
  - Return all the domains covered by Google's index and the number of pages in each, ordered by the number of pages
- The programming model does not support this per se,
  but the implementations do
  - The Shuffle stage groups and orders!

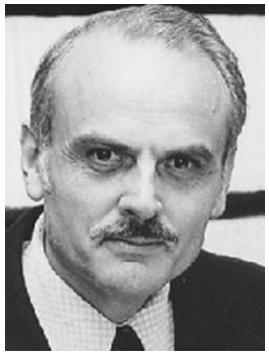
- The map does nothing
  - If we have a single reducer, we will get sorted output
  - If we have multiple reducers, we can get partly sorted output but it's quite easy to write a last-pass file that merges all of the files

Why NoSQL?

- In the last thirty years relational databases have been the default choice for serious data storage
- An architect starting a new project:
  - your only choice is likely to be which relational database to use
  - often not even that, if your company has a dominant vendor
- In the past, other proposals for database technology:
  - deductive databases in the 1980's
  - object databases in the 1990's
  - XML databases in the 2000's
  - these alternatives never got anywhere!

The Value of Relational Databases

- Effective and efficient management of persistent data
- Concurrency control
- Data integration
- A standard data model
- A standard query language



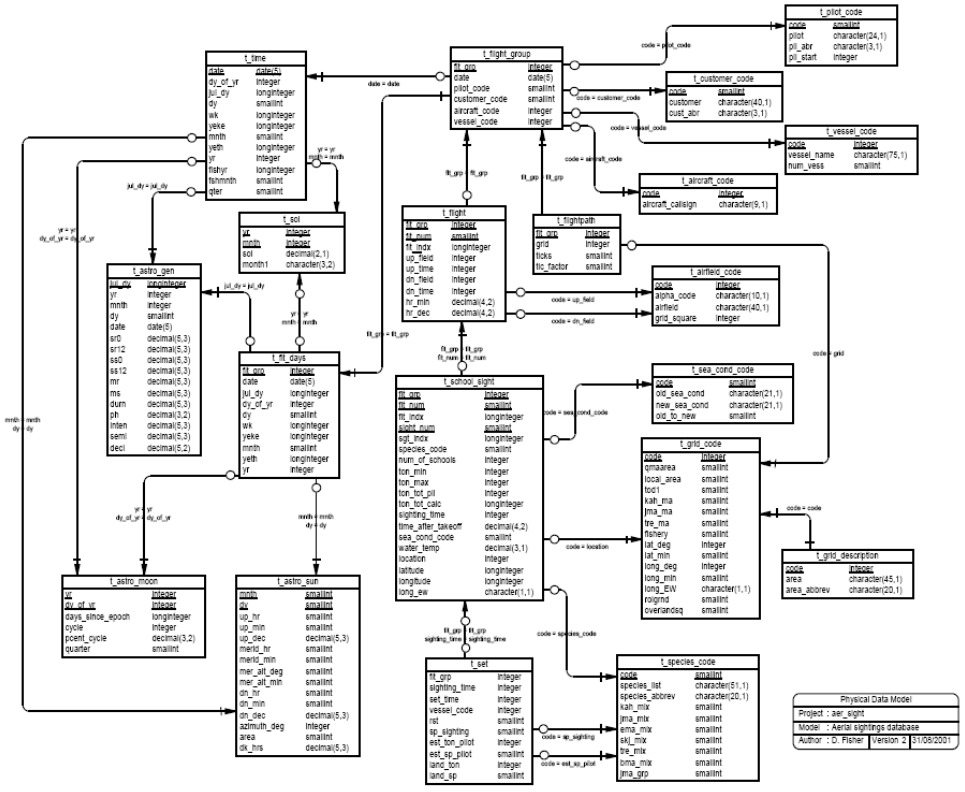
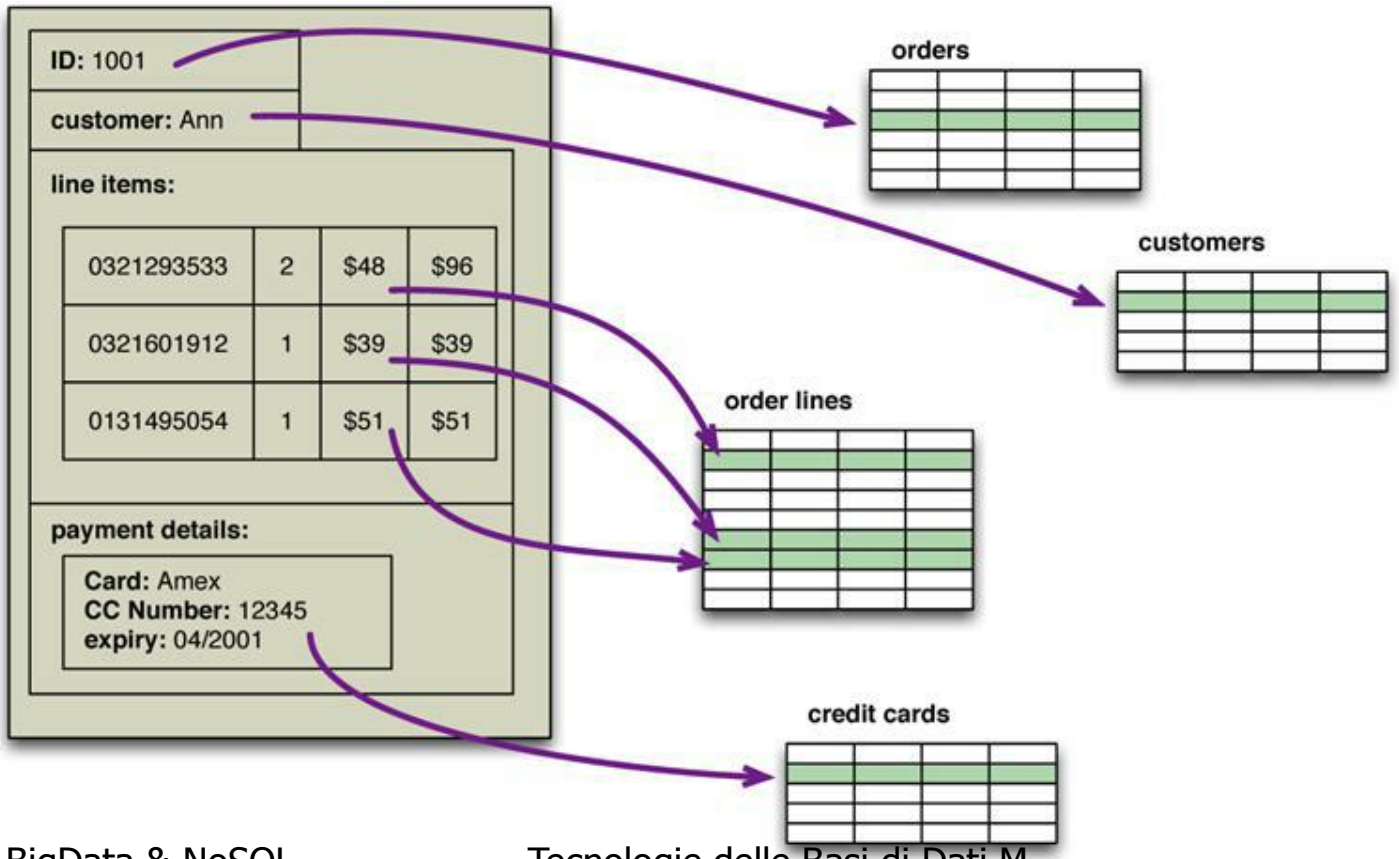


Figure 1: Entity Relationship Diagram (ERD) for the aer_sight database

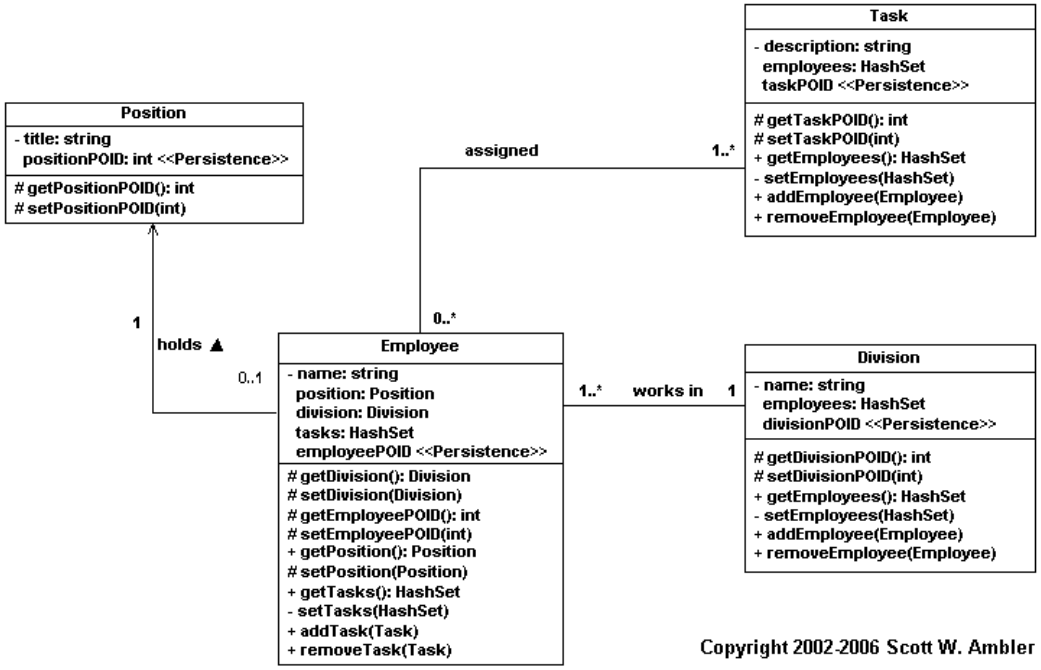
Impedance Mismatch

- Difference between the persistent data model and the in-memory data structures



A proposal to solve the problem (1990s)

- Databases that replicate the in-memory data structures to disk
- Object-oriented databases!



- Faded into obscurity in a few years..
- Solution emerged:
  - ORDBMS
  - object-relational mapping frameworks

Evolution of applications

- OO databases are dead. Why?
  - SQL provides an integration mechanism between applications
  - The database acts as an integration database
  - Multiple applications one database
- 2000s: a distinct shift to application databases (SOA)
  - Web services add more flexibility for the data structure being exchanged
  - richer data structures to reduce the number of round trips
    - nested records, lists, etc.
  - usually represented in XML or JSON
  - you get more freedom of choosing a database
    - a decoupling between your internal database and the services with which you talk to the outside world
  - despite this freedom, however, it wasn't apparent that application databases led to a big rush to alternative data stores

Relational databases are familiar and usually work very well
(or, at least, well enough)

Attack of the Clusters

- A shift from scale up to scale out
  - with the explosion of data volume the computer architectures based on cluster of commodity hardware emerged as the only solution
  - but relational databases are not designed to run (and do not work well) on clusters!
- The mismatch between relational databases and clusters
  led some organization to consider alternative solutions to data storage
- Google: BigTable
- Amazon: Dynamo

NoSQL

- Term appeared in the late 90s
  - open-source relational database [Strozzi NoSQL]
  - tables as ASCII files, without SQL
- Current interpretation
  - June 11, 2009: meetup in San Francisco
  - open-source, distributed, non-relational DBs
  - Hashtag chosen: #NoSQL
  - Main features:
    - Not using SQL and the relational model
    - Open-source projects (mostly)
    - Running on clusters
    - Schema-less
  - However, no accepted precise definitions
- Most people say that NoSQL means "Not Only SQL"

Key Points

- Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism
- Application developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures
- There is a movement away from using databases as integration points towards encapsulating databases within applications and integrating through services
- The vital factor for a change in data storage was the need to support large volumes of data by running on clusters
  - Relational databases are not designed to run efficiently on clusters
- NoSQL is an accidental neologism: There is no prescriptive definition
- All you can make is an observation of common characteristics:
  - Not using the relational model
  - Running well on clusters
  - Open-source
  - Built for the 21st century web estates
  - Schema-less

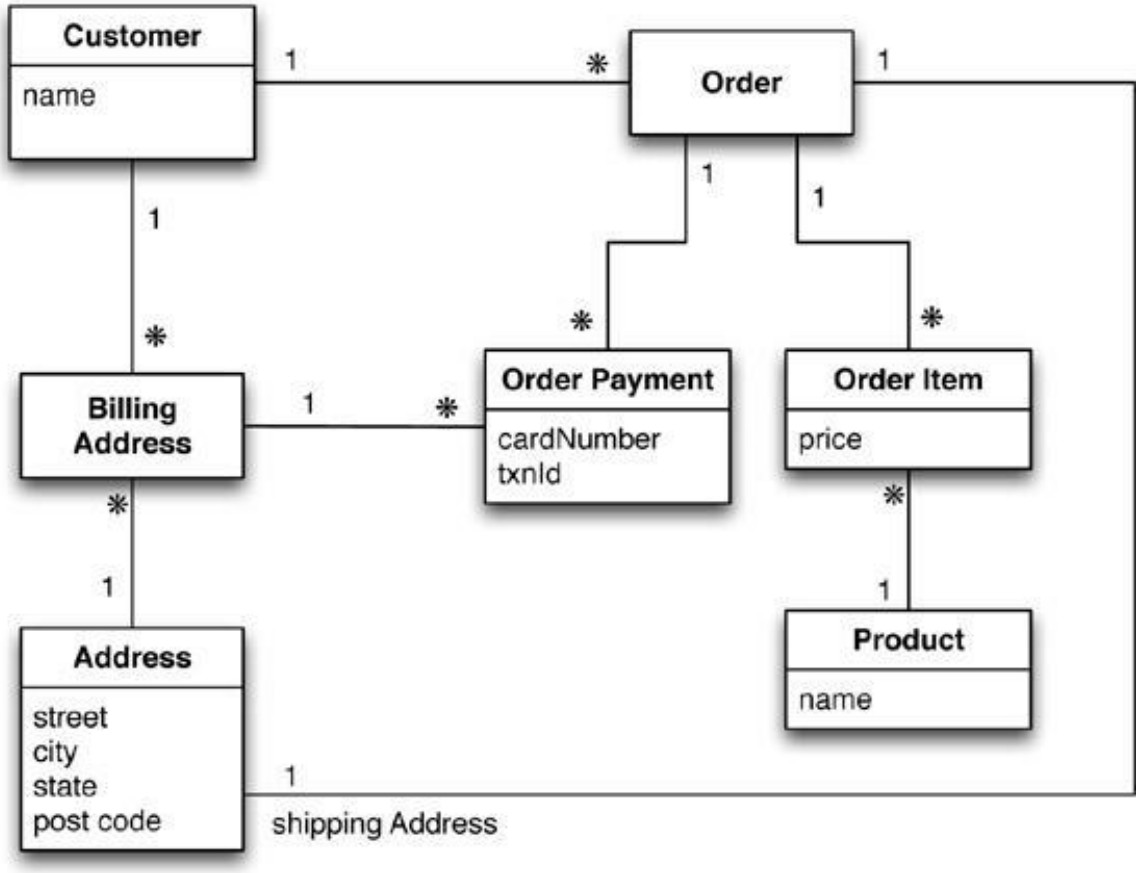
NoSQL Data Models

- A data model is a set of constructs for representing the information
  - Relational model: tables, columns and rows
- Storage model: how the DBMS stores and manipulates the data internally
- A data model is usually independent of the storage model
- Data models for NoSQL systems:
  - aggregate models
    - key-value
    - document
    - column-family
  - graph-based models

Aggregates

- Data as units that have a complex structure
  - more structure than just a set of tuples
  - example:
    - complex record with: simple fields, arrays, records nested inside
- Aggregate in Domain-Driven Design
  - a collection of related objects that we treat as a unit
  - a unit for data manipulation and management of consistency
- Advantages of aggregates:
  - easier for application programmers to work with
  - easier for database systems to handle operating on a cluster

Example


Customer
name
Order
Billing Address
Order Payment
cardNumber
txnId
Order Item
price
Address
street
city
state
post code
Product
name
shipping Address
1
*

Relational implementation

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

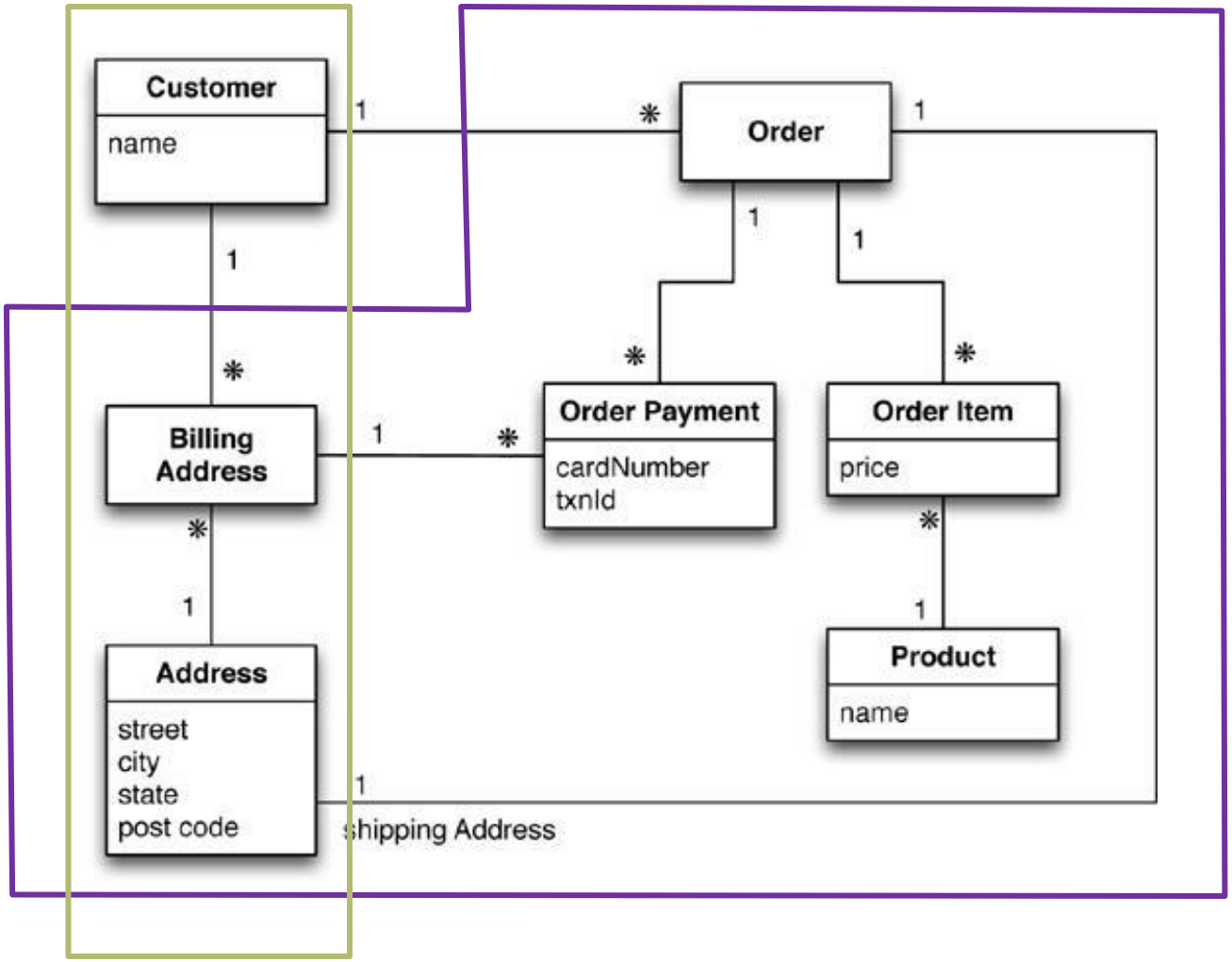
Address

Id	City
77	Chicago

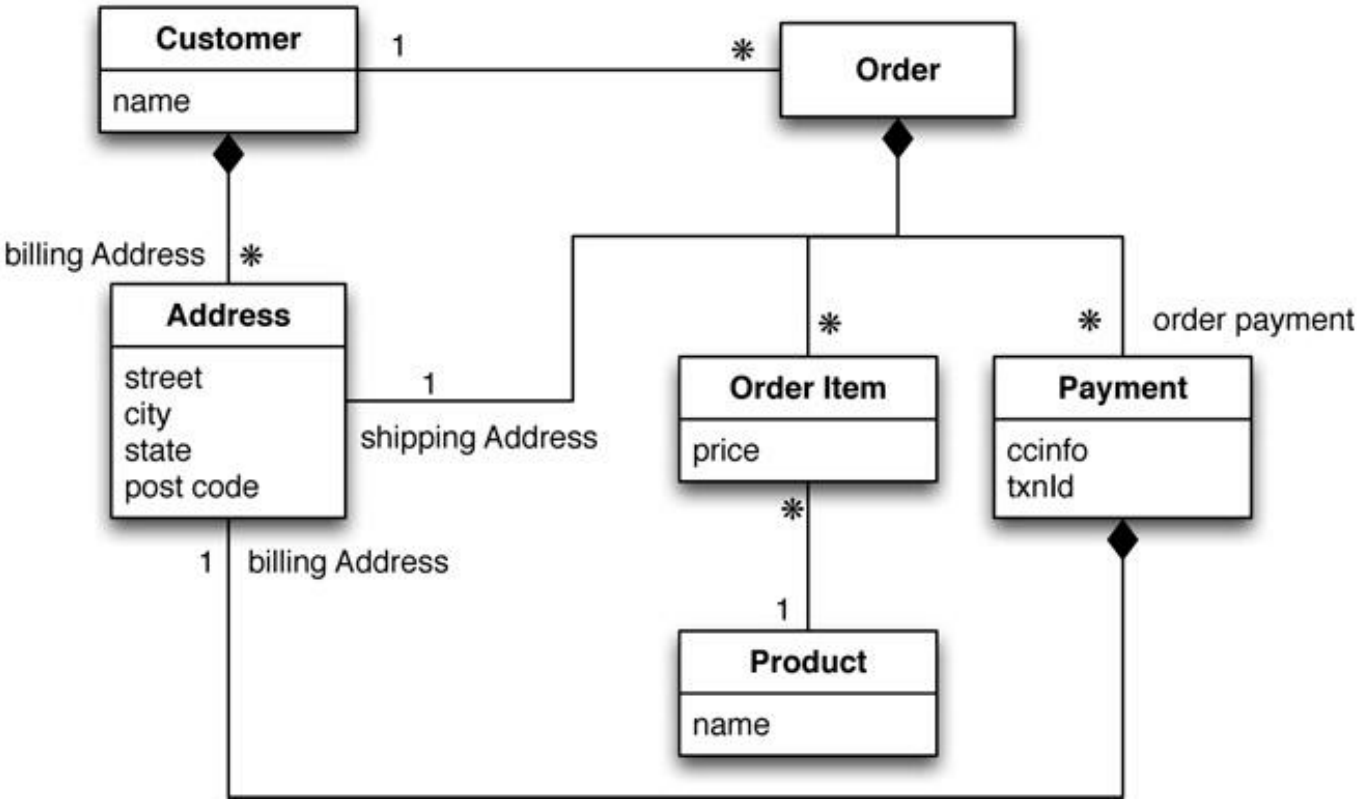
OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Aggregation



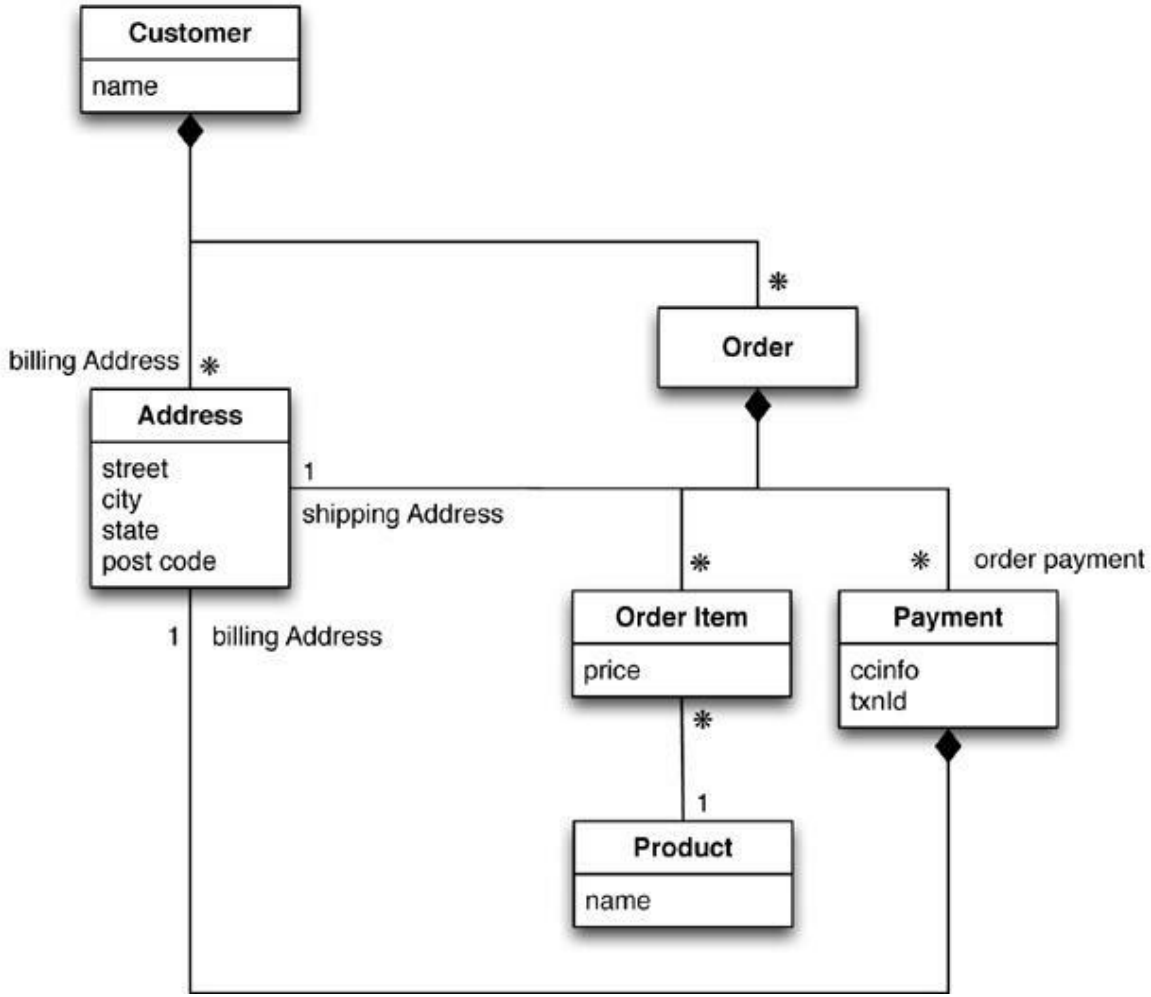
Aggregate representation



Aggregate implementation

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Another possible solution



Aggregate implementation (2)

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
```

Design strategy

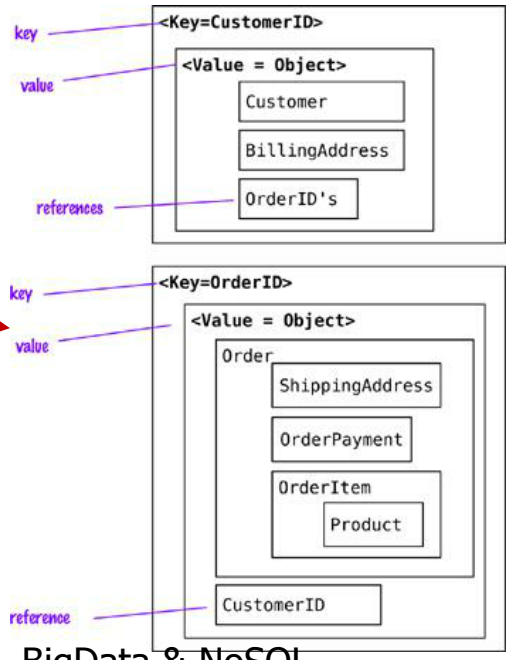
- No universal answer for how to draw aggregate boundaries
- It depends entirely on how you tend to manipulate data!
  - Accesses on a single order at a time: first solution
  - Accesses on customers with all orders: second solution
- Context-specific
  - some applications will prefer one or the other
  - even within a single system
- Focus on the unit of interaction with the data storage
- Pros: it helps greatly with running on a cluster
  - data will be manipulated together, and thus should live on the same node!
- Cons: an aggregate structure may help with some data interactions, but be an obstacle for others

Transactions

- Relational databases have ACID transactions
- Aggregate-oriented databases:
  - don’t have ACID transactions that span multiple aggregates
  - they support atomic manipulation of a single aggregate at a time
- Part of the consideration for deciding how to aggregate data

Key-Value and Document Data Models

- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key is used to get data
- Key-value database
  - The aggregate is opaque to the database
- Document database
  - A structure in the aggregate



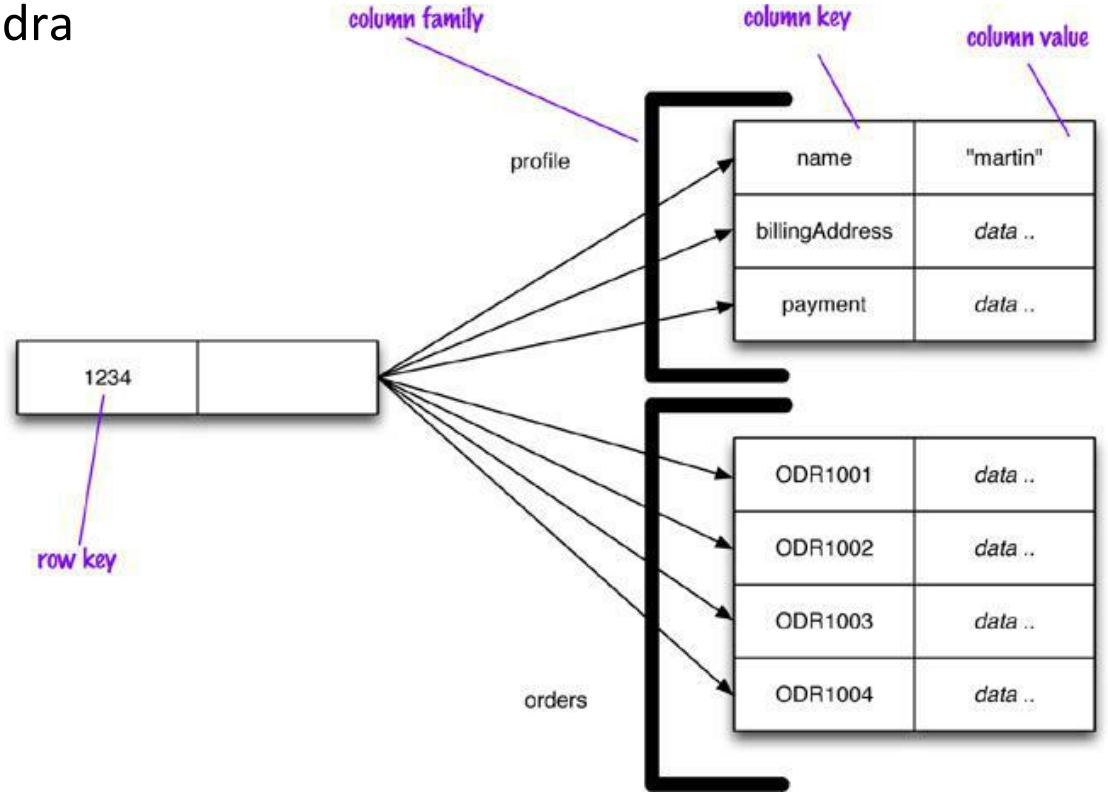

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}

# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Key-Value vs. Document stores

- Key-value database
  - A key plus a big BLOB of mostly meaningless bits
  - We can store whatever we like in the aggregate
  - We can only access an aggregate by lookup based on its key
- Document database
  - A key plus a structured aggregate
  - More flexibility in access
    - we can submit queries to the database based on the fields in the aggregate
    - we can retrieve part of the aggregate rather than the whole thing
  - Indices based on the contents of the aggregate

Column-Family Stores

- A two-level aggregate structure:
  - A key and a row aggregate
  - A row aggregate is a group of columns
- Bigtable, HBase, Cassandra



Properties of Column-Family Stores

- Operations also allow picking out a particular column
  - get('1234', 'name')
- Each column:
  - has to be part of a single column family
  - acts as unit for access
- You can add any column to any row, and rows can have very different columns
- You can model a list of items by making each item a separate column
- Two ways to look at data:
  - Row-oriented
    - Each row is an aggregate
    - Column families represent useful chunks of data within that aggregate
  - Column-oriented:
    - Each column family defines a record type
    - Row as the join of records in all column families

Cassandra

- Skinny row
  - few columns with the same columns used by many different rows
  - the column family defines a record type
  - each row is a record and each column is a field
- Wide row
  - many columns (perhaps thousands)
  - rows having very different columns
  - models a list, with each column being one element in that list
- A column family can contain both field-like columns and list-like columns

Key Points

- An aggregate is a collection of data that we interact with as a unit.
- Aggregates form the boundaries for ACID operations with the database
- Key-value, document, and column-family databases can all be seen as forms of aggregate-oriented database
- Aggregates make it easier for the database to manage data storage over clusters
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate
- Aggregate-ignorant databases are better when interactions use data organized in many different formations

Relationships

- Relationship between different aggregates:
  - Put the ID of one aggregate within the data of the other
  - Join: write a program that uses the ID to link data
  - The database is ignorant of the relationship in the data

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

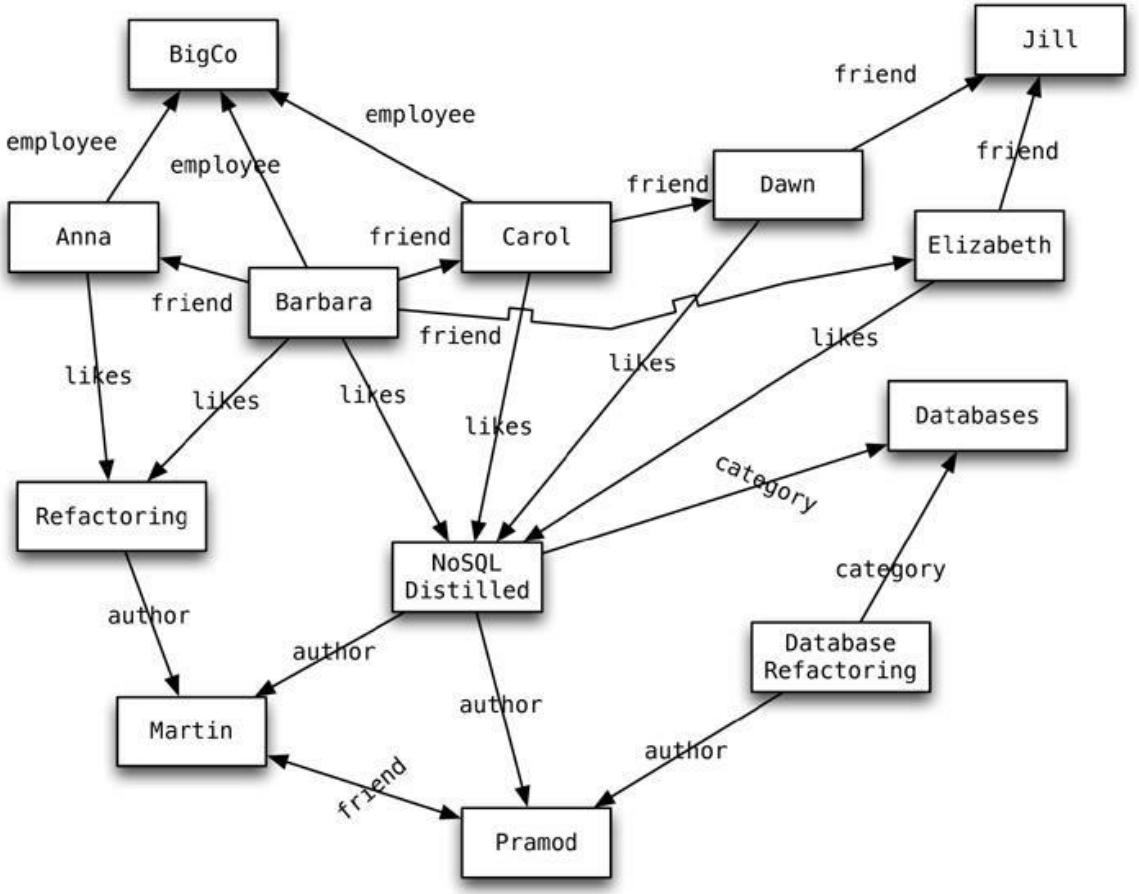
Relationship management

- Many NoSQL databases provide ways to make relationships visible to the database
  - Document stores makes use of indexes
  - Riak (key-value store) allows you to put link information in metadata
- But what about updates?
  - Aggregate-oriented databases treat the aggregate as the unit of data-retrieval
  - Atomicity is only supported within the contents of a single aggregate
  - Updates over multiple aggregates at once is a programmer's responsibility!
  - In contrast, relational databases provide ACID guarantees while altering many rows through transactions

Graph Databases

- Graph databases are motivated by a different frustration with relational databases
  - Complex relationships require complex join
- Goal:
  - Capture data consisting of complex relationships
  - Data naturally modelled as graphs
  - Examples: Social networks, Web data, product preferences

A graph database



Possible query: “find the books in the Databases category that are written by someone whom a friend of mine likes”

Data model of graph databases

- Basic characteristic: nodes connected by edges (also called arcs)
- Beyond this: a lot of variation in data models
  - FlockDB is simply nodes and edges with no mechanism for additional attributes
  - Neo4J stores Java objects to nodes and edges in a schema-less fashion
  - Infinite Graph stores Java objects, which are subclasses of built-in types, as nodes and edges
- Queries
  - Navigation through the network of edges
  - You do need a starting place
  - Nodes can be indexed by an attribute such as ID

Graph vs. Relational databases

- Relational databases
  - implement relationships using foreign keys
  - joins require to navigate around and can get quite expensive
- Graph databases
  - make traversal along the relationships very cheap
  - performance is better for highly connected data
  - shift most of the work from query time to insert time
  - good when querying performance is more important than insert speed

Graph vs. Aggregate-oriented databases

- Very different data models
- Aggregate-oriented databases
  - distributed across clusters
  - simple query languages
  - no ACID guarantees
- Graph databases
  - more likely to run on a single server
  - graph-based query languages
  - transactions maintain consistency over multiple nodes and edges

Schema-less Databases

- No fixed schema:
  - key-value store allows you to store any data you like under a key
  - document databases make no restrictions on the structure of the documents you store
  - column-family databases allow you to store any data under any column you like
  - graph databases allow you to freely add new edges and freely add properties to nodes and edges as you wish

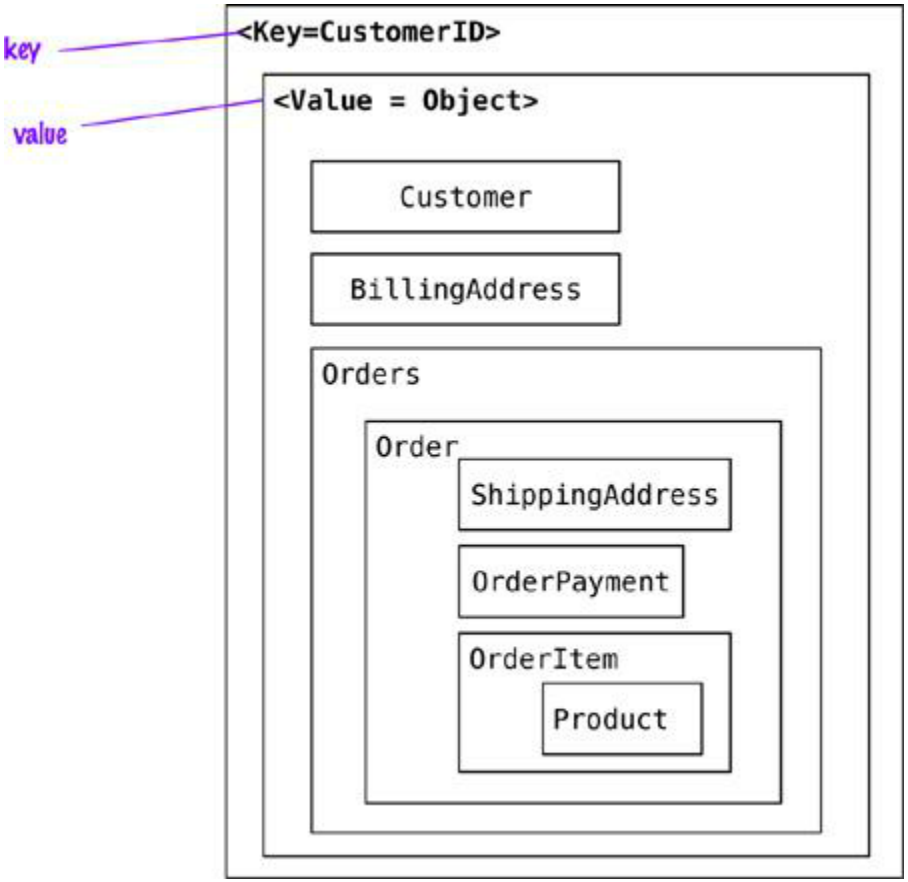
Pros and cons of schema-less data

- Pros:
  - More freedom and flexibility
  - you can easily change your data organization
  - you can deal with non-uniform data
- Cons:
  - A program that accesses data:
    - almost always relies on some form of implicit schema
    - it assumes that certain fields are present
    - carry data with a certain meaning
  - The implicit schema is shifted into the application code that accesses data
    - To understand what data is present you have look at the application code
  - The schema cannot be used to:
    - decide how to store and retrieve data efficiently
    - ensure data consistency
  - Problems if multiple applications, developed by different people, access the same database
- Relational schemas can be changed at any time with standard SQL commands!

Materialized Views

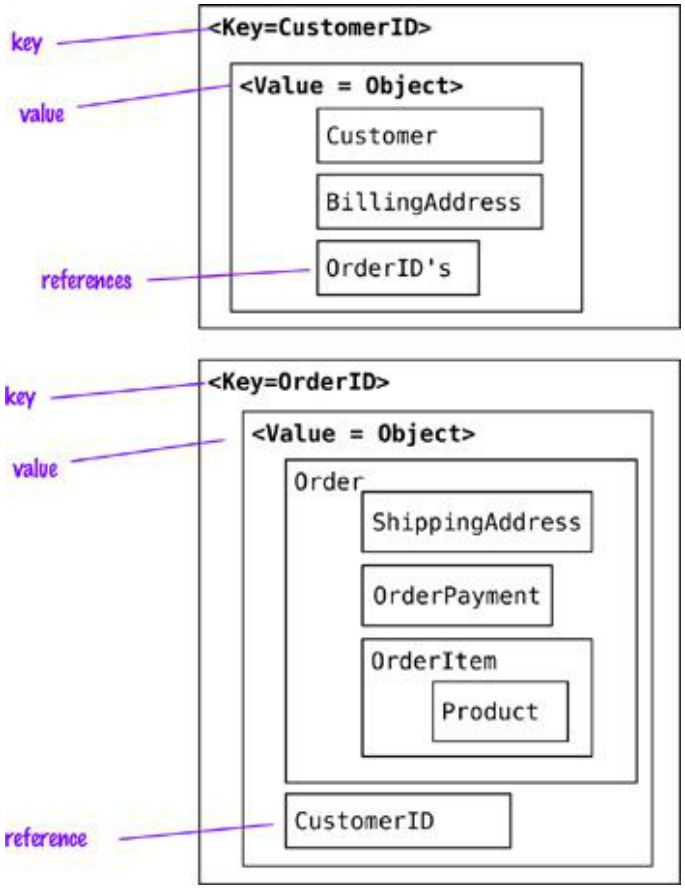
- A relational view is a table defined by computation over the base tables
- Materialized views: computed in advance and cached on disk
- NoSQL databases:
  - do not have views
  - have pre-computed and cached queries usually called "materialized view"
- Strategies to building a materialized view
  - Eager approach
    - the materialized view is updated at the same time of the base data
    - good when you have more frequent reads than writes
  - Detached approach
    - batch jobs update the materialized views at regular intervals
    - good when you don't want to pay an overhead on each update
- Materialized views can be used within the same aggregate

Data Accesses in key-value store



The application can read all customer's information by using the key

Splitting aggregates




```
# Customer object
{
"customerId": 1,
"customer": {
  "name": "Martin",
  "billingAddress": [{"city": "Chicago"}],
  "payment": [{"type": "debit","ccinfo": "1000-1000-1000-1000"}],
  "orders":[{"orderId":99}]
  }
}
```

```
# Order object
{
"customerId": 1,
"orderId": 99,
"order":{
  "orderDate":"Nov-20-2011",
  "orderItems":[{"productId":27, "price": 32.45}],
  "orderPayment":[{"ccinfo":"1000-1000-1000-1000",
          "txnId":"abelif879rft"}],
  "shippingAddress":{"city":"Chicago"}
  }
}
```

We can now find the orders independently from the Customer, and with the orderID reference in the Customer we can find all Orders for the Customer

Aggregates for analytics

- An aggregate update may store which Orders have a given Product in them
- Useful for Real Time Analytic

```
{
"itemid":27,
"orders":{99,545,897,678}
}
{
"itemid":29,
"orders":{199,545,704,819}
}
```

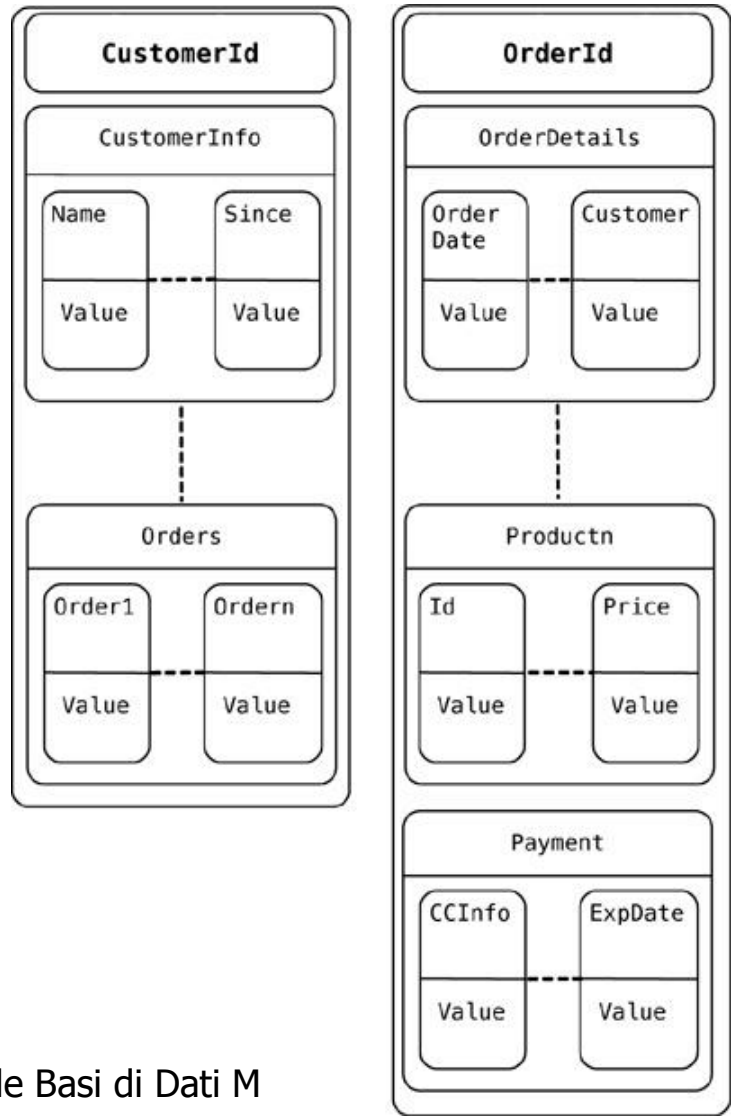
Data Accesses in document stores

- We can query inside documents: removing references is possible
- We do not need to update the Customer object when new orders are placed by the Customer

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}

# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

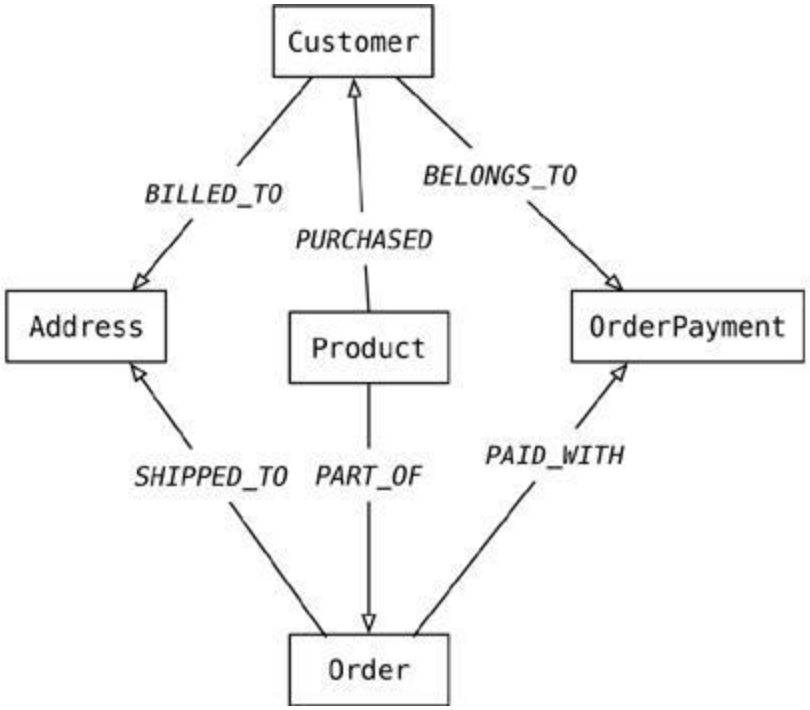
Data Accesses in column-family stores

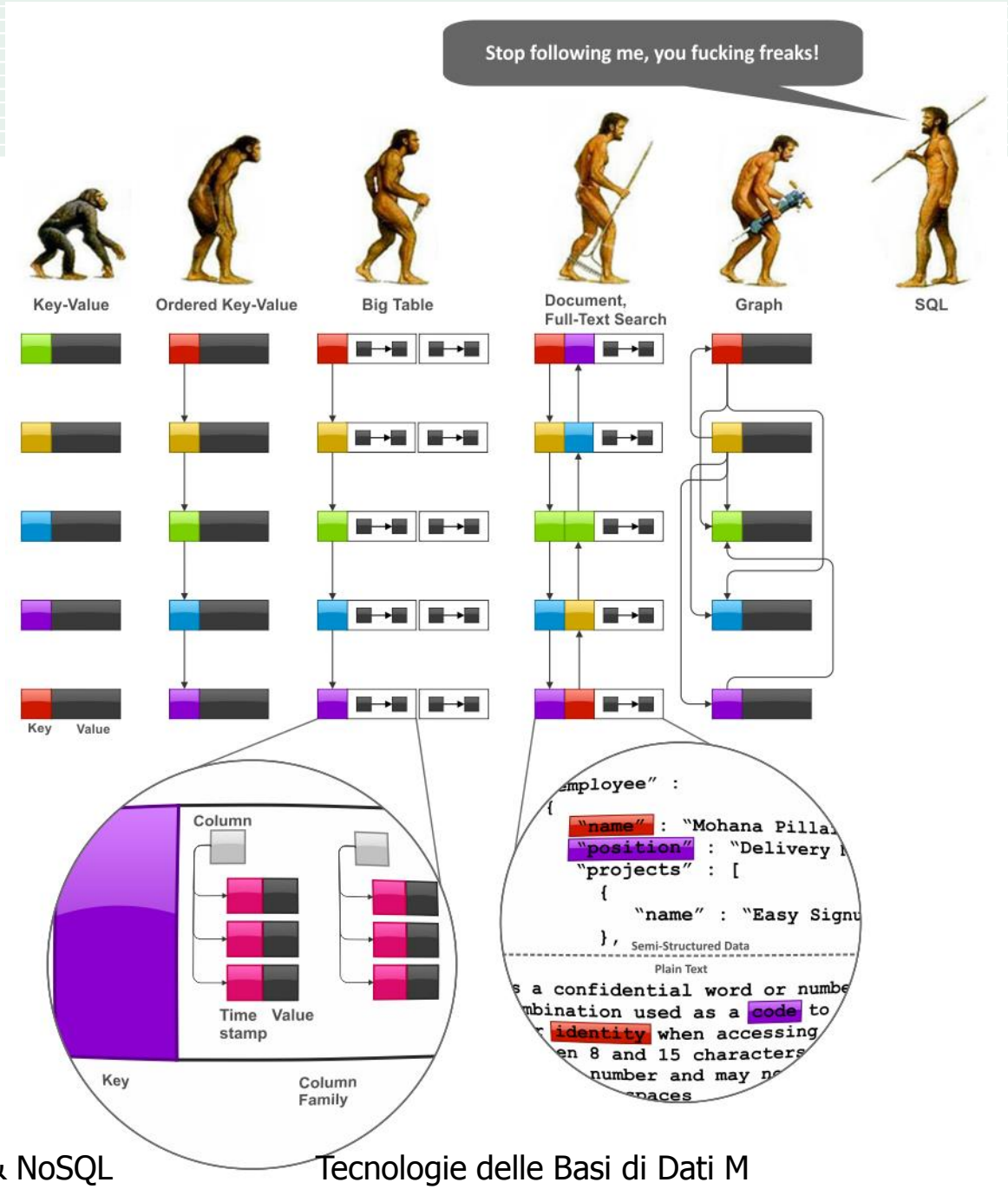
- The columns are ordered
- We can choose columns that are frequently used so that they are fetched first
- Splitting data in different column-family families can improve performance



Data Accesses in graph databases

- Each node has independent relationships with other nodes
- The relationships have names
- Relationship names let you traverse the graph




Stop following me, you fucking freaks!
Key-Value
Ordered Key-Value
Big Table
Document,
Full-Text Search
Graph
SQL
Key
Value
Column
Time stamp
Value
Key
Column Family
"name" : "Mohana Pilla
"position" : "Delivery
"projects" : [
{
"name" : "Easy Signu
},
Semi-Structured Data
Plain Text
a confidential word or numbe
mbination used as a code to
identity when accessing
en 8 and 15 characters
number and may n
spaces

Key Points

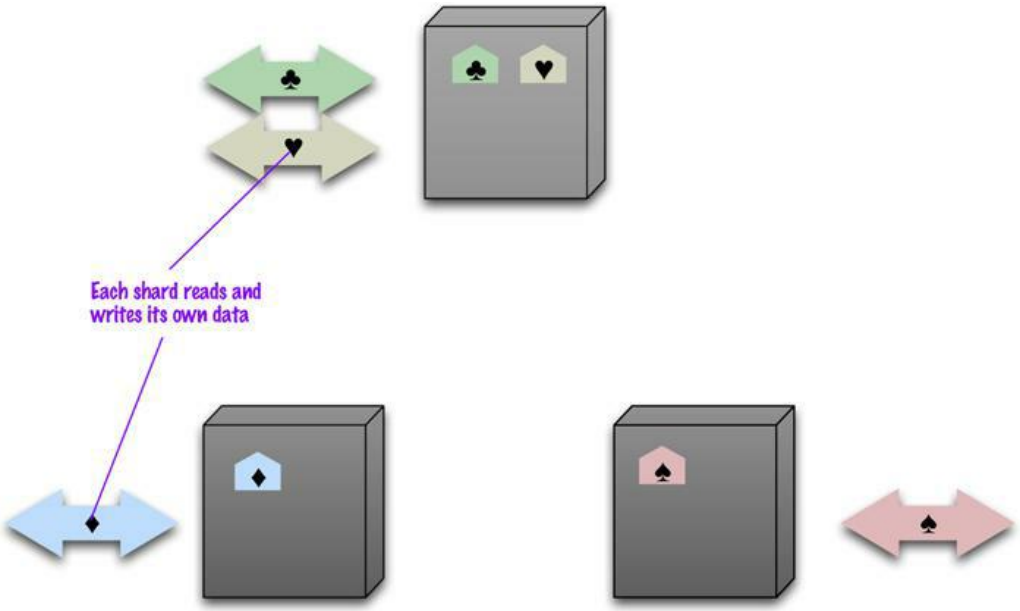
- Aggregate-oriented databases make inter-aggregate relationships more difficult to handle than intra-aggregate relationships
- Graph databases organize data into node and edge graphs
  - They work best for data that has complex relationship structures
- Schema-less databases allow you to freely add fields to records, but there is usually an implicit schema expected by users of the data
- Aggregate-oriented databases often compute materialized views to provide data organized differently from their primary aggregates
  - This is often done with map-reduce computations

Data distribution

- NoSQL systems: data distributed over large clusters
- Aggregate is a natural unit to use for data distribution
- Data distribution models:
  - Single server (is an option for some applications)
  - Multiple servers
- Orthogonal aspects of data distribution:
  - Sharding: different data on different nodes
  - Replication: the same data copied over multiple nodes
    - master-slave
    - peer-to-peer

Sharding

- Different parts of the data onto different servers
  - Horizontal scalability
  - Ideal case: different users all talking to different server nodes
  - Data accessed together on the same node – aggregate unit!
- Pros: it can improve both reads and writes
- Cons: Clusters use less reliable machines – resilience decreases



Improving performance

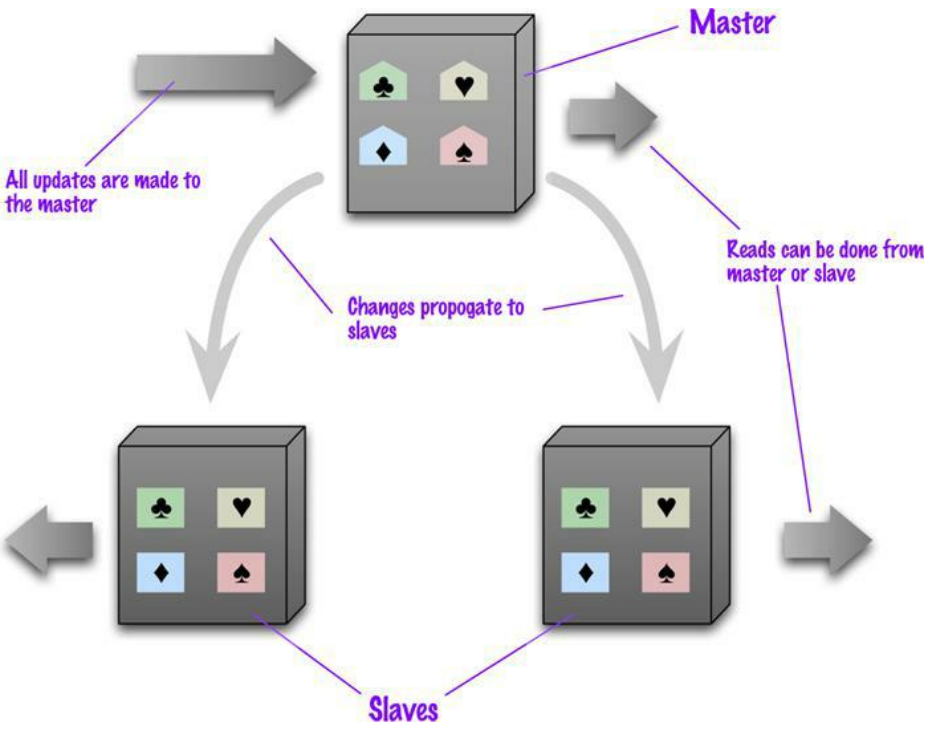
Main rules of sharding:

1. Place the data close to where it is accessed
   - Orders for Boston: data in your eastern US data center
2. Try to keep the load even
   - All nodes should get equal amounts of the load
3. Put together aggregates that may be read in sequence
   - Same order, same node

- Many NoSQL databases offer auto-sharding
  - the database takes on the responsibility of sharding

Master-Slave Replication

- Master
  - is the authoritative source for the data
  - is responsible for processing any updates to that data
  - can be appointed manually or automatically
- Slaves
  - A replication process synchronizes the slaves with the master
  - After a failure of the master, a slave can be appointed as new master very quickly

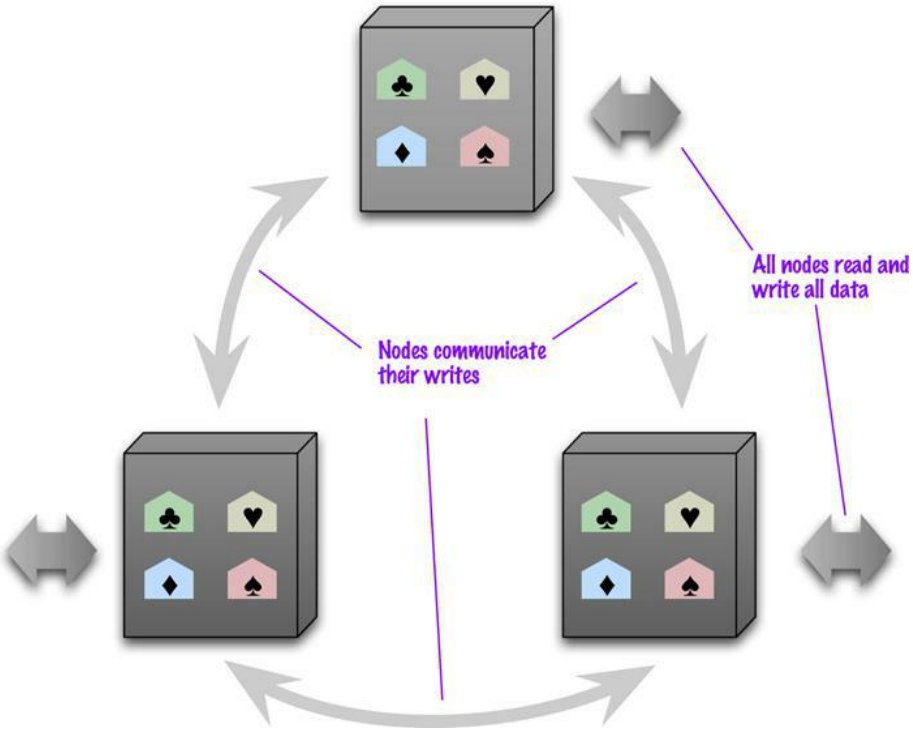


Pros and cons of Master-Slave Replication

- Pros
  - More read requests:
    - Add more slave nodes
    - Ensure that all read requests are routed to the slaves
  - Should the master fail, the slaves can still handle read requests
  - Good for datasets with a read-intensive dataset
- Cons
  - The master is a bottleneck
    - Limited by its ability to process updates and to pass those updates on
    - Its failure does eliminate the ability to handle writes until:
      - the master is restored or
      - a new master is appointed
  - Inconsistency due to slow propagation of changes to the slaves
  - Bad for datasets with heavy write traffic

Peer-to-Peer Replication

- All the replicas have equal weight, they can all accept writes
- The loss of any of them doesn't prevent access to the data store

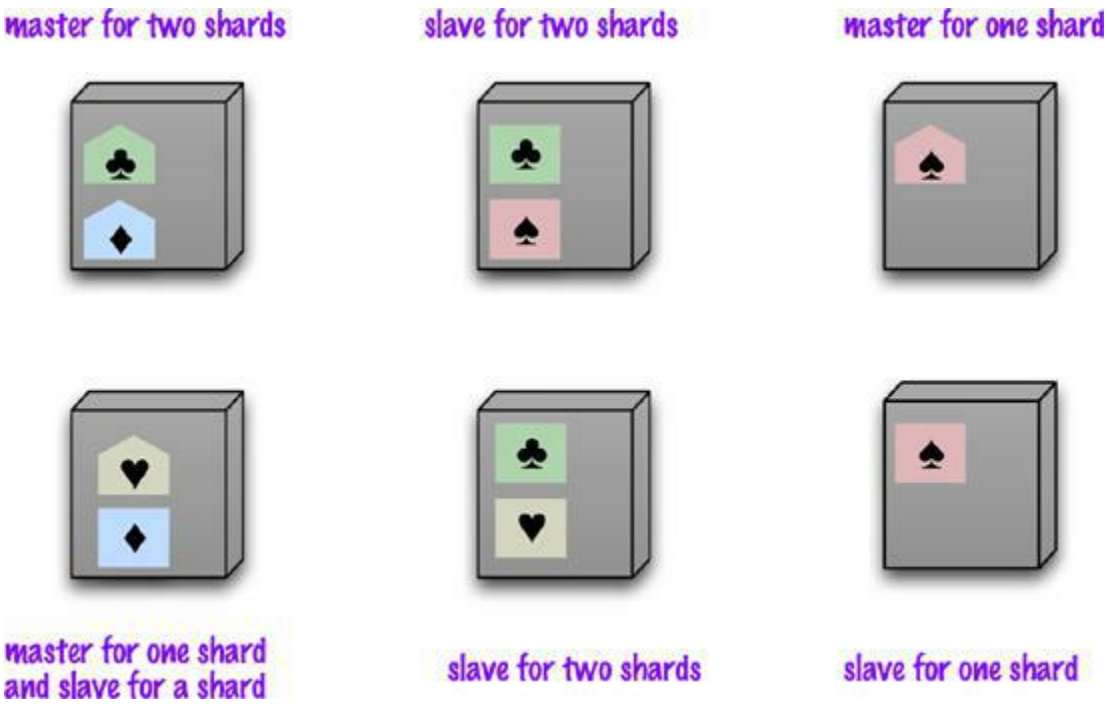


Pros and cons of peer-to-peer replication

- Pros:
  - you can ride over node failures without losing access to data
  - you can easily add nodes to improve your performance
- Cons:
  - Inconsistency!
    - Slow propagation of changes to copies on different nodes
      - Inconsistencies on read lead to problems but are relatively transient
    - Two people can update different copies of the same record stored on different nodes at the same time - a write-write conflict.
      - Inconsistent writes are forever

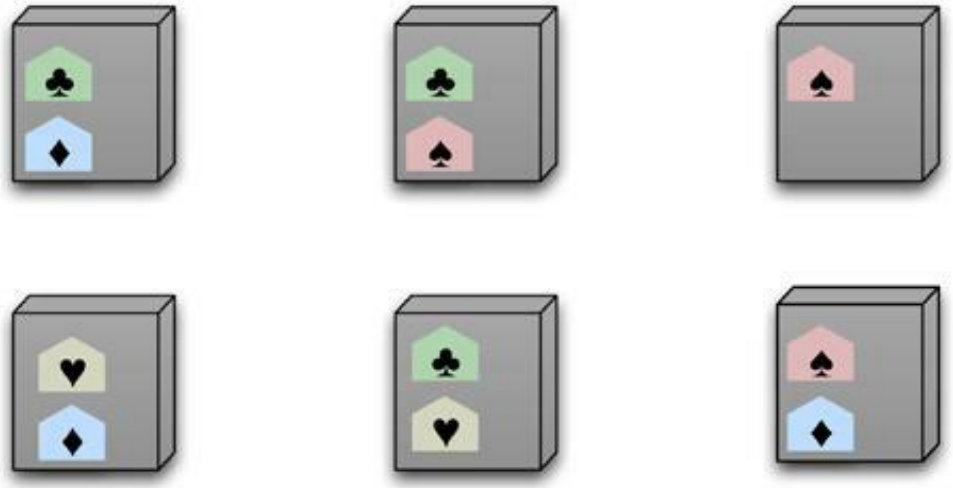
Sharding and Replication on MS

- We have multiple masters, but each data only has a single master
- Two schemes:
  - A node can be a master for some data and slaves for others
  - Nodes are dedicated for master or slave duties



Sharding and Replication on P2P

- Usually each shard is present on three nodes
- A common strategy for column-family databases

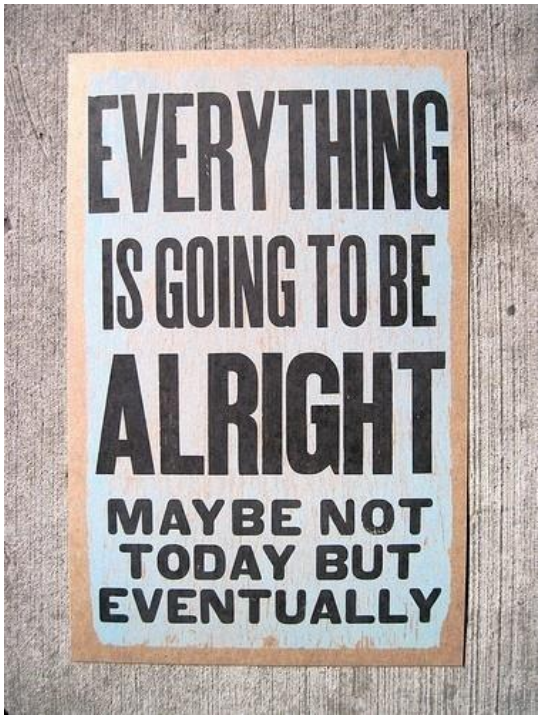


Key points

- There are two styles of distributing data:
  - Sharding distributes different data across multiple servers
    - each server acts as the single source for a subset of data
  - Replication copies data across multiple servers
    - each bit of data can be found in multiple places
- A system may use either or both techniques
- Replication comes in two forms:
  - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads
  - Peer-to-peer replication allows writes to any node
    - The nodes coordinate to synchronize their copies of the data
- Master-slave replication reduces the chance of update conflicts
- Peer-to-peer replication avoids loading all writes onto a single point of failure

Consistency

- Biggest change from a centralized relational database to a cluster-oriented NoSQL
  - Relational databases: **strong** consistency
  - NoSQL systems: mostly **eventual** consistency



Update Consistency

- **Write-write conflict**: two people updating the same data item at the same time
- If the server serialize them (sequential consistency):
  - One is applied and immediately overwritten by the other
  - Lost update

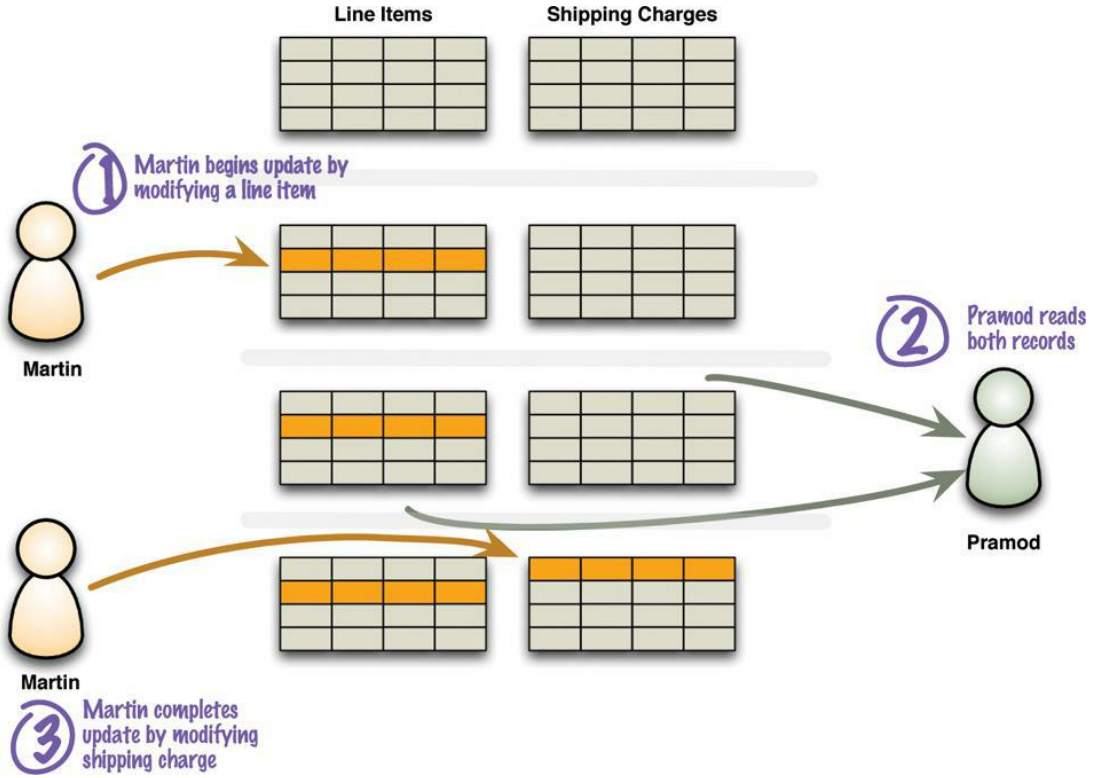
- Solutions:
  - **Pessimistic** approach
    - Prevent conflicts from occurring
    - Usually implemented with write locks managed by the system
  - **Optimistic** approach
    - Let conflicts occur, but detects them and takes action to sort them out
    - Approaches:
      - conditional updates: test the value just before updating
      - save both updates: record that they are in conflict and then merge them
- Do not work if there is more than one server (peer-to-peer replication)

Pessimistic vs optimistic approach

- Concurrency involves a fundamental tradeoff between:
  - safety (avoiding errors such as update conflicts), and
  - liveness (responding quickly to clients).
- Pessimistic approaches often:
  - severely degrade the responsiveness of a system
  - leads to deadlocks, which are hard to prevent and debug

Read Consistency (or read-write conflict)

- A read in the middle of two logically-related writes



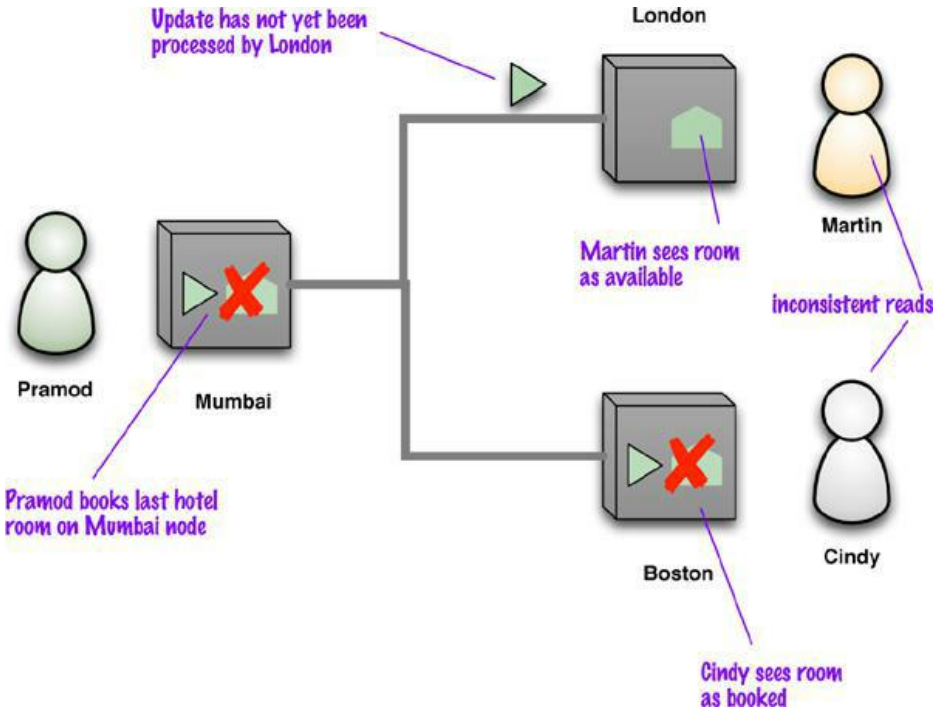
- **Logical consistency**: no read or read-write conflicts

Transactions on NoSQL databases

- Graph databases tend to support ACID transactions
- Aggregate-oriented NoSQL database:
  - Support atomic updates, but only within a single aggregate
  - To avoid inconsistency: orders in a single aggregate
  - Update over multiple aggregates: possible inconsistent reads
  - **Inconsistency window**: length of time an inconsistency is present
- Amazon's documentation:
  - "inconsistency window for SimpleDB service is usually less than a second"

Replication consistency

- Replication is another source of inconsistency



- **Eventual consistency:** at any time nodes may have replication inconsistencies but, if there are no further updates, eventually all nodes will be updated to the same value
- **Stale** data: out of date

Logical and eventual consistency

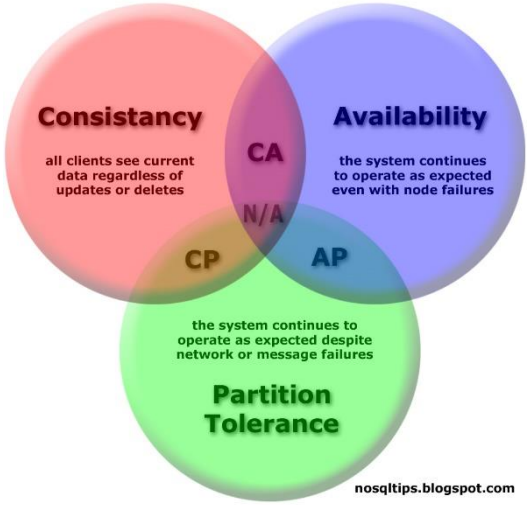
- Eventual consistency is independent from logical consistency but replication can lengthen the inconsistency window
  - Two updates on the master performed in rapid succession
  - Delays in networking can lengthen the inconsistency on a slave
- Consequences of inconsistency windows
  - different people see different data at the same time: usually tolerated
  - read-your-writes consistency: should be guaranteed
- **Session consistency**: within a user's session
  - Sticky session: tied to one node (session affinity)
  - Version stamps: every interaction latest version stamp

Version stamps

- Help you detect concurrency conflicts
- When you read data, then update it, you can check the version stamp to ensure nobody updated the data between your read and write
- Version stamps can be implemented using counters, GUIDs (a large random number that's guaranteed to be unique), content hashes, timestamps, or a combination of these
- With distributed systems, a vector of version stamps (a set of counters, one for each node) allows you to detect when different nodes have conflicting updates

The CAP Theorem

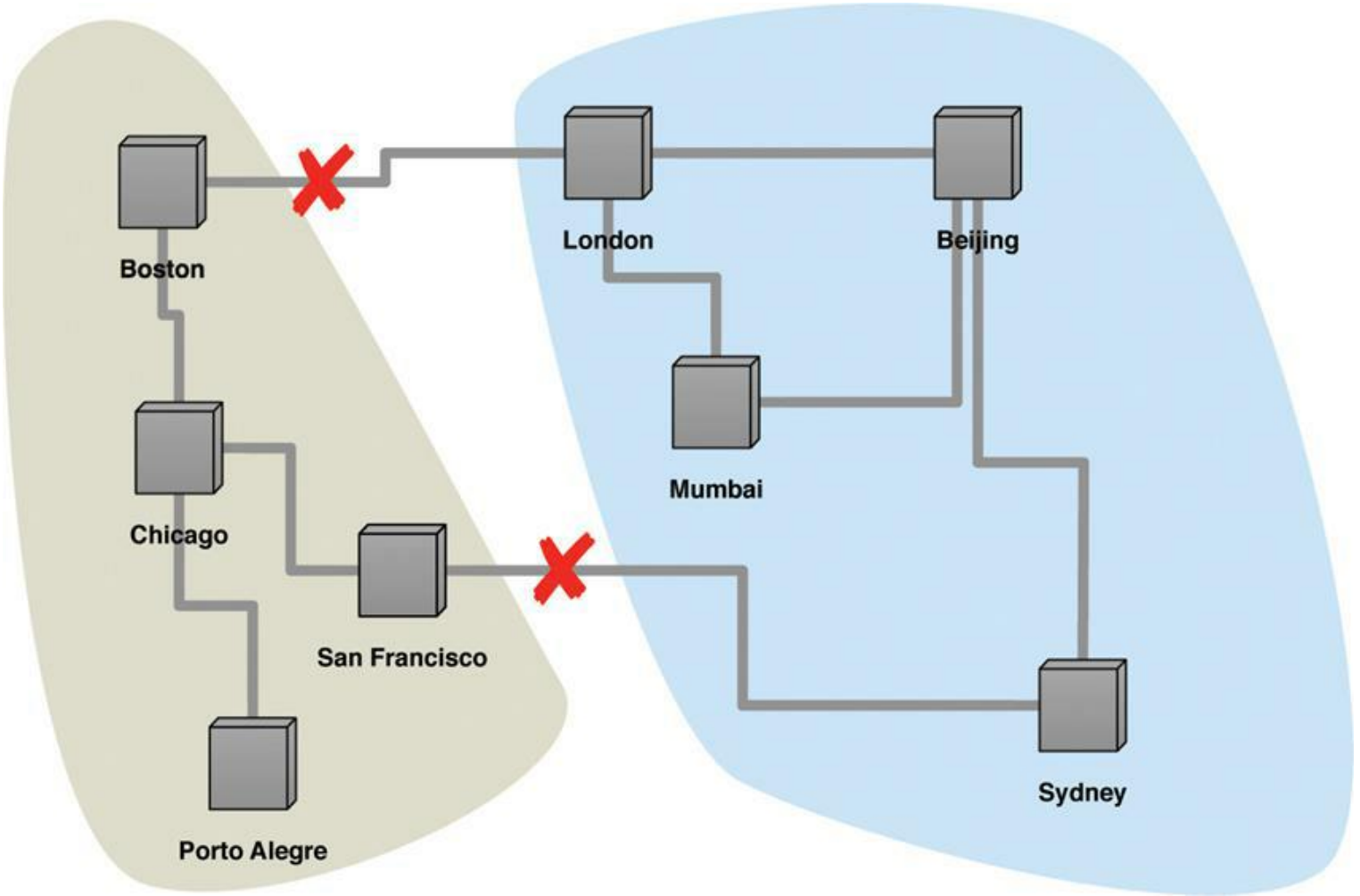
"Given the properties of Consistency, Availability, and Partition tolerance, you can only get two"



The CAP Theorem

- Why you may need to relax consistency
- Proposed by Eric Brewer in 2000
- Formal proof by Seth Gilbert and Nancy Lynch in 2002
- **Consistency**: all people see the same data at the same time
- **Availability**: if you can talk to a node in the cluster, it can read and write data
- **Partition tolerance**: the cluster can survive communication breakages that separate the cluster into partitions unable to communicate with each other

Network partition

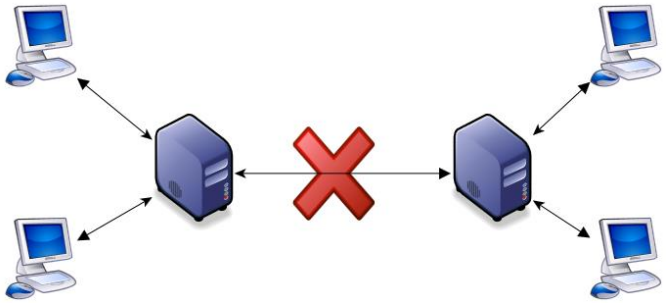


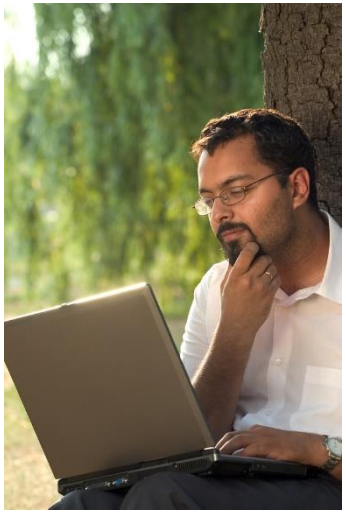
CA systems

- A single-server system is the obvious example of a CA system
- CA cluster: if a partition occurs, all the nodes would go down
  - A failed, unresponsive node doesn't infer a lack of CAP availability
- A system that suffer partitions: tradeoff consistency vs. availability
  - Give up to some consistency to get some availability

An example

- Ann is trying to book a room of the Ace Hotel in New York on a node located in London of a booking system
- Pathin is trying to do the same on a node located in Mumbai
- The booking system uses a peer-to-peer distribution
- There is only a room available
- The network link breaks





Possible solutions

- CP: Neither user can book any hotel room, sacrificing availability
- caP: Designate Mumbai node as the master for Ace hotel
  - Pathin can make the reservation
  - Ann can see the inconsistent room information
  - Ann cannot book the room
- AP: both nodes accept the hotel reservation
  - Overbooking!
- These situations are closely tied to the domain
  - Financial exchanges? Blogs? Shopping charts?
- Issues:
  - How tolerant you are of stale reads
  - How long the inconsistency window can be
- BASE approach (Basically Available, Soft state, Eventual consistency)

Durability

- You may want to trade off durability for higher performance
  - Main memory database: if the server crashes, any updates since the last flush will be lost
  - Keeping user-session states as temporary information
  - Capturing telemetric data from physical devices
- Replication durability
  - Occurs when a node processes an update
  - But fails before that update is replicated to the other nodes

Quorums

- How many nodes need to be involved to get strong consistency?
  - Write quorum: W > N/2
  - The number of nodes participating in the write (W) must be more than the half the replication factor (N)
- How many nodes you need to contact to be sure you have the most up-to-date change?
  - Read quorum: R + W > N
- In a master-slave distribution one R/W is enough
- A replication factor of 3 is usually enough to have good resilience

Key points

- Write-write conflicts occur when two clients try to write the same data at the same time
- Read-write conflicts occur when one client reads inconsistent data in the middle of another client's write
- Pessimistic approaches lock data records to prevent conflicts
- Optimistic approaches detect conflicts and fix them
- Distributed systems (with replicas) see read-write conflicts due to some nodes having received updates while other nodes have not
- Eventual consistency means that at some point the system will become consistent once all the writes have propagated to all the nodes
- Clients usually want read-your-writes consistency, which means a client can write and then immediately read the new value
  - This can be difficult if the read and the write happen on different nodes
- To get good consistency, you need to involve many nodes in data operations, but this increases latency
  - So you often have to trade off consistency versus latency
- The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency
- Durability can also be traded off against latency, particularly if you want to survive failures with replicated data
- You do not need to contact all replicants to preserve strong consistency with replication; you just need a large enough quorum