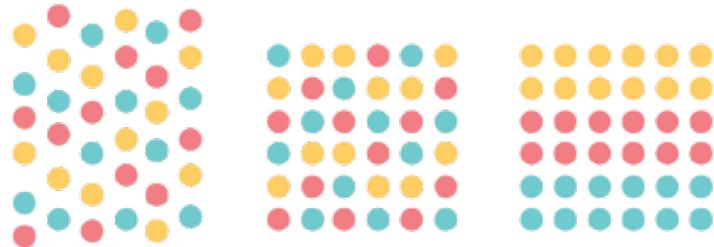


USDE - Notes

Unstructured Streaming and Data Engineering

Professors: Emanuele Della Valle - Marco Brambilla

Author: Simone Staffa



Released with Beerware License, Rev. 42 (<https://spdx.org/licenses/Beerware.html>)
“As long as you retain this notice you can do whatever you want with this stuff. If we meet some day,
and you think this stuff is worth it, you can buy me a beer in return”

November 6, 2020

Contents

1	Introduction (Course motivation)	4
1.1	Data-driven Decision Making for Data-driven Organizations	4
1.2	Solving problems with Big Data, Data Science and ... Data Engineering	4
1.2.1	What's Big Data?	5
1.2.2	What's Data Science	6
1.2.3	What's Data Engineering	7
2	No SQL Intro	9
2.1	Big Data Platforms: Architectures, Features and System	9
2.1.1	Big Data vs Traditional Data	9
2.1.2	The Concept of Data Lake	10
2.1.3	Scalability	11
2.2	ACID vs. BASE and SQL vs. NoSQL	12
2.2.1	Transactional Properties	12
2.2.2	CAP Theorem	13
2.2.3	ACID vs. BASE properties	14
2.2.4	The NoSQL World	15
3	Graph DB	17
3.1	Graph Theory	17
3.1.1	Useful definitions	18
3.1.2	Graph Abstract Data Type (ADT)	22
3.2	Graph Databases	22
3.2.1	Advantages of Graph Databases	23
3.3	Neo4J	24
3.3.1	Cypher	25
4	Key-Value DB	28
4.1	How does a key-value database work?	28
4.1.1	Key Features	28
4.2	Redis	29
4.2.1	Scaling Redis	31
4.2.2	Redis topologies	31
4.2.3	Redis Advantages	33
4.3	Key-Value and Caching	33
4.3.1	What is Caching?	33
4.3.2	Memcached	35
5	Big Column DB	36
5.1	Introduction	36
5.1.1	Column wise vs. Row wise database	36
5.1.2	Column storage	37
5.2	Cassandra	39
5.2.1	Cassandra Properties	40
5.2.2	Gossip Protocol	40
5.2.3	Replica Placement Strategies	40
5.2.4	Write operation	41
5.2.5	Read operation	42
5.2.6	Cassandra Quorums and Consistency Levels	42
5.2.7	Data model	43
5.2.8	What about...SQL?	44
5.3	Is Cassandra a good fit?	45

6 Document-oriented DB	47
6.1 Why document-based?	47
6.2 MongoDB	48
6.2.1 Facts	49
6.2.2 Data Model	49
6.2.3 Queries	49
6.2.4 CAP Theorem and Mongo	50
7 Streaming Data Engineering	51
7.1 The Solution Space	51
7.1.1 The Dimensions: Throughput vs. Latency vs. Message size	51
7.1.2 Three Cases along a continuum	52
7.2 The Batch Case	54
7.3 The Continuous Case	55
7.3.1 From Passive to Active DBMS and DSMS	56
7.3.2 Event-based systems	57
7.3.3 Service Oriented Architecture (SOA)	59
8 EPL	64
8.1 EPL and Esper	64
8.2 Processing Model	64
8.3 Event types and Query syntax	65
8.4 Pattern Matching	67
9 Kafka	70
9.1 Kafka Basics	70
9.1.1 Kafka in a nutshell	70
9.1.2 Main Concepts and Terminology	70
9.1.3 Kafka Internals	72
9.2 Avro and Schema Registry	73
9.2.1 Schema Evolution	74
9.3 Connect for Data Movement	75
9.3.1 Kafka Connect	75
9.4 Kafka Stream Processing	77

1 Introduction (Course motivation)

1.1 Data-driven Decision Making for Data-driven Organizations

In many organizations decisions are made by "questionable" methodologies such as

- **Highest Paid Person Opinion (HiPPO):** when Galileo tried to say that the earth cycles around the sun, the pope (the HiPPO) stated that heliocentrism was impossible.
- **Flipism:** all decisions are made by flipping a coin (randomly)

This could have been the right approach in the '70s... but in the Digital Era one can dream of data-driven organization, taking decisions using data.

"Decisions no longer have to be made in the dark or based on gut instinct; they can be based on evidence, experiments and more accurate forecasts", McKinsey

Data-driven organizations

- **perform better:** the data shows where they can streamline their processes
- **are operationally more predictable:** data insights fuel current and future decision making
- **are more profitable:** constant improvements and better predictions help to outsmart the competition and improve innovation.

1.2 Solving problems with Big Data, Data Science and ... Data Engineering

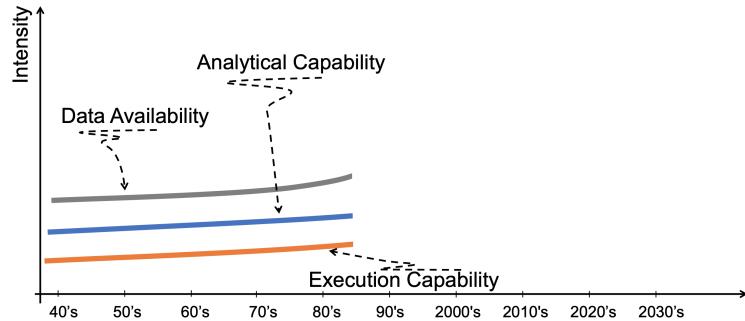


Figure 1: Up until '90s the data available was growing together with our analytical and execution capabilities.

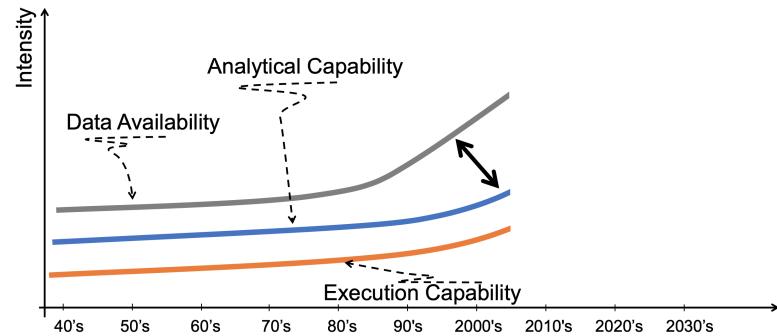


Figure 2: With the new millenium we have the appearance of Big Data. Data availability is growing fast and the digital revolution gap is growing.

1.2.1 What's Big Data?

Big data is a term that describes the large volume of data – both structured and unstructured – that inundates a business on a day-to-day basis.

IBM data scientists break big data into four dimensions:

- **Volume** (data at scale): volume is increasing, we have more and more data. Curiosity: in Italy is rare to find companies work with more than 20 Terabytes of data (only big customers)
- **Variety** (data in many form): structured, unstructured (or semi-structured e.g., graph), text, multimedia
- **Velocity** (data in motion): analysis of streaming data to enable decision within fractions of a second (real time decision and data analysis while data are coming). This is not a property of data but is specifically related to the kind of analysis we want to achieve.
- **Veracity** (data uncertainty): managing the reliability and predictability of inherently imprecise data type. This is not a property of data, it regards the quality of data. For 1 purpose the data are good, for another purpose the same data may not be good (or useful).

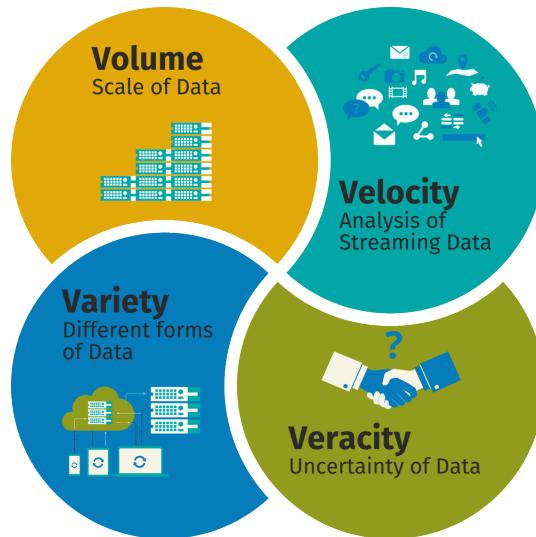


Figure 3: With the new millennium we have the appearance of Big Data. Data availability is growing fast and the digital revolution gap is growing.

Big Data techs are like "crude oil" that we have to:

- Extract
- Transport in mega-tankers
- Ship through pipelines
- Store in massive silos

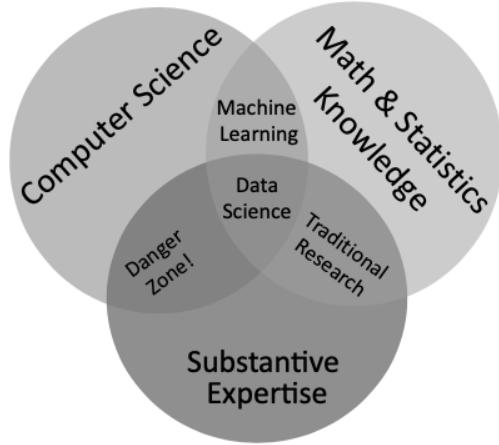


1.2.2 What's Data Science

The Science (and Art) of:

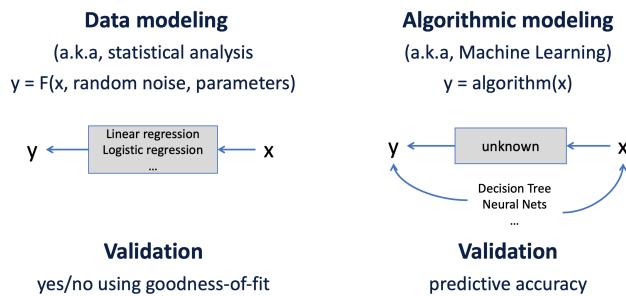
- **Discovering** what we don't know from data
- Obtaining **predictive, actionable insight** from data
- **Creating Data Products** that have business impact now
- **Communicating** relevant business stories from data
- **Building confidence** in decisions that drive business value

Data scientists are a new breed of analytical data expert who have the technical skills to solve complex problems – and the curiosity to explore what problems need to be solved. They're part mathematician, part computer scientist and part trend-spotter. And, because they straddle both the business and IT worlds, they're highly sought-after and well-paid.



We distinguish **two cultures** of Statistical Modeling:

- Data modeling (traditional research)
- Algorithmic modeling (more like machine learning)



The algorithmic modeling culture starts with data and has two main goals:

- Descriptions: describe how nature associates responses to inputs
- Predictions: predict response for future input variables

1.2.3 What's Data Engineering

Following with the crude oil example, data engineers build "the refinery".

"A scientist can discover a new star, but he cannot make one. He would have to ask an engineer to do it for him." - Gordon Lindsay Glegg

A data engineer is a specialist that **maintain data and models available and usable** by others (i.e., Data Scientists and Business Analysts). According to Google: "A professional data engineer enables data-driven decision making by collecting, transforming, and publishing data. He should also be able to leverage, deploy, and continuously train pre-existing machine learning models."

Data engineering purposes a paradigmatic shift, solving problems in new ways.

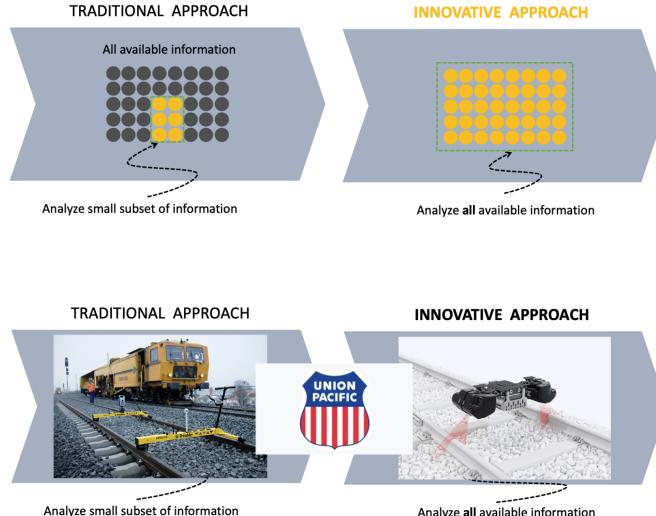


Figure 4: Instead of looking for the perfect exact data, measure everything and **leverage more of the data being captured**. With a large enough dataset at some point we reach the same result (Central Limit Theorem).

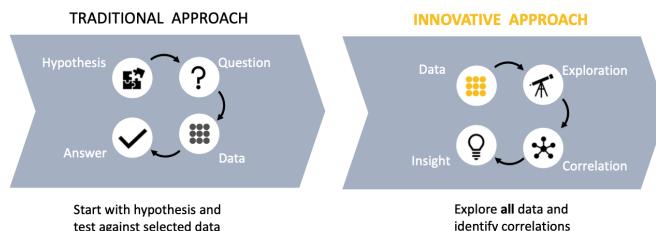


Figure 5: **Data-driven exploration looking for correlation**. For instance, your butcher sells both pure meat and semi-prepared dishes because he knows that if you see the variety of products that he prepares and sells, you will probably notice something that you like!

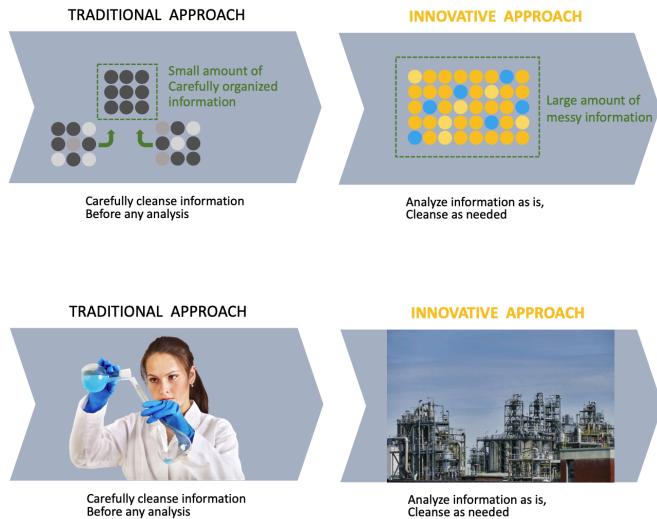
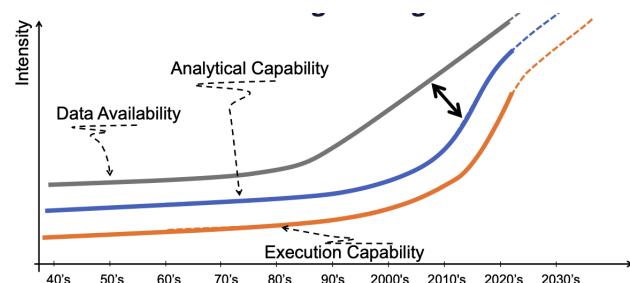


Figure 6: **Reduce effort required to leverage data.** If you can do it by hand it is not said that you can do it automatically. Is hard to do things at scale.



Figure 7: **Leverage data as it is captured.**

The gap is closing thanks to Big Data, Data Science and ... Engineering.



2 No SQL Intro

2.1 Big Data Platforms: Architectures, Features and System

2.1.1 Big Data vs Traditional Data

	Traditional	Big Data
Data Characteristics	 Relational (with highly modeled schema)	
Cost	 Expensive (storage and compute capacity)	 Commodity (storage and compute capacity)
Culture	 Rear-view reporting (using relational algebra)	 Intelligent action (using relational algebra AND ML, graph, streaming, image processing)

Figure 8: Big Data vs. Traditional Data

The first step towards Big Data and flexibility is to adopt a schema-less data storage. Indeed, we don't want to waste time designing complex and fixed schema.

- Aggregate-based: key-value, big-table, column-based, document-based
- Relationship-based: graph dbs are better than relational!

Even in this context we see a paradigmatic shift introduced by Big Data. From **schema on write** to **schema-on-read**.

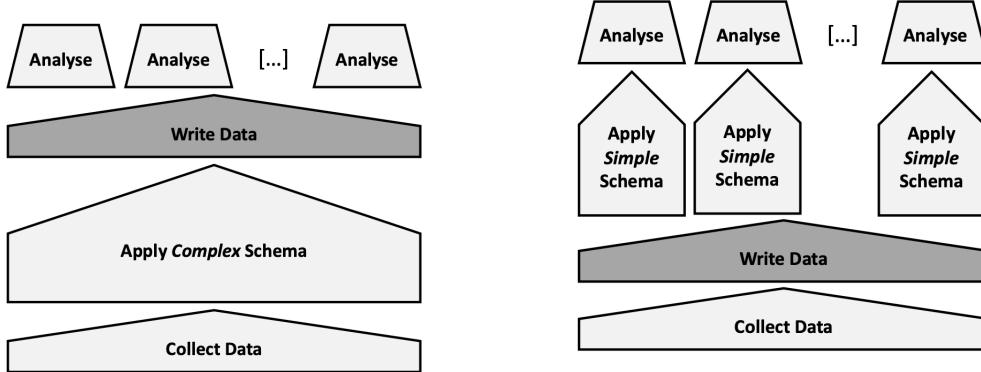


Figure 9: **Schema-on-write**: the rigid and traditional strategy (relational data) in which a complex schema is applied after a long lasting dis-lect and load data first and ask questions/queries cussing. Here we collect the data from different later. All data are kept and the minimal schema sources, ensuring that it is compatible with our analysis on that.

Figure 10: **Schema-on-read**: schema-less application strategy (document-based data) in which we collect data first and ask questions/queries later. All data are kept and the minimal schema sources, ensuring that it is compatible with our analysis on that. New analysis can then be introduced in any point in time.

2.1.2 The Concept of Data Lake

A Data Lake is a repository in which we store all the possible data that we need in our business. These raw data can be structured or unstructured, without any specific organization and they are there ready to be analyzed when needed. Indeed, there is a specific process that characterized the flow of Big Data into the Data Lake and the various transformation that are applied before analysis and visualization.

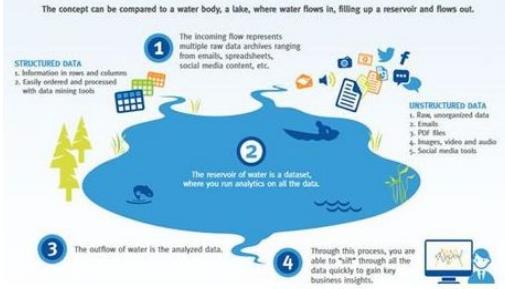


Figure 11: Data lake

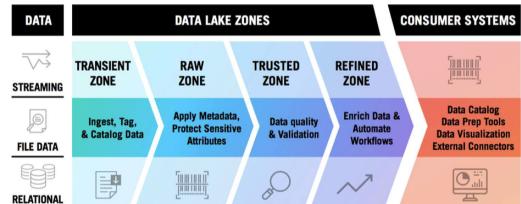


Figure 12: Data Lake in process

Data Ingestion is the process of importing, transferring and loading data for storage and later use. It involves loading data from a variety of sources. It can involve altering and modification of individual files to fit into a format that optimizes the storage. For instance, in Big Data small files are concatenated to form files of 100s of MBs and large files are broken down in files of 100s of MBs.

Data Wrangling: the process of cleansing "raw" data and transforming raw it into data that can be analysed to generate valid actionable insight. It includes understanding, cleansing, augmenting and shaping data. The results is data in the best format (e.g., columnar) for the analysis to perform.

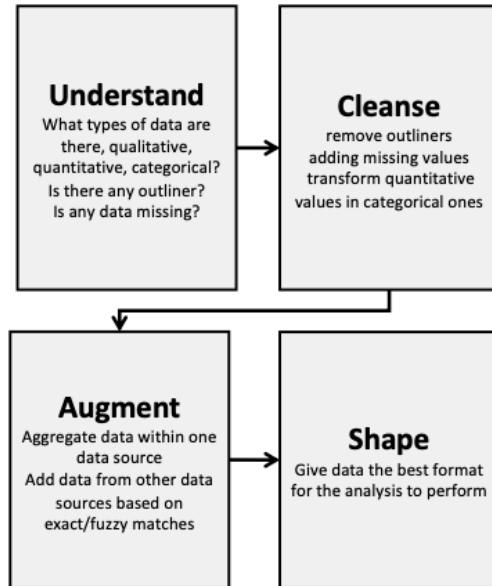


Figure 13: Data Wrangling

2.1.3 Scalability

Adding data to a system may degrade its performances.

- **”Traditional” SQL system scale vertically:** when the machine, where the SQL system runs, no longer performs as required, the solution is to **buy a better machine** (with more RAM, more cores and more disk).
- **Big Data solutions scale horizontally:** when the machines, where the big data solution runs, no longer performs as required, the solution is **to add another machine**.

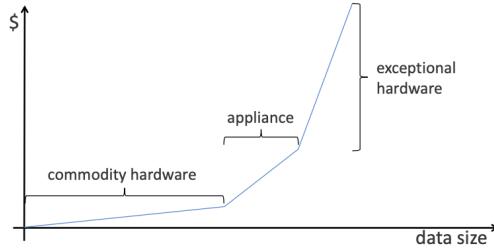


Figure 14: Vertical Scalability

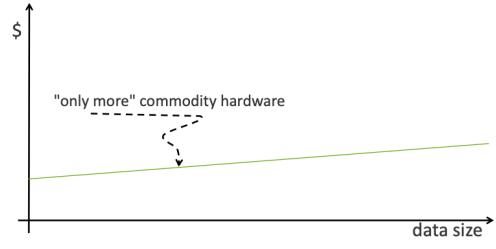


Figure 15: Horizontal Scalability

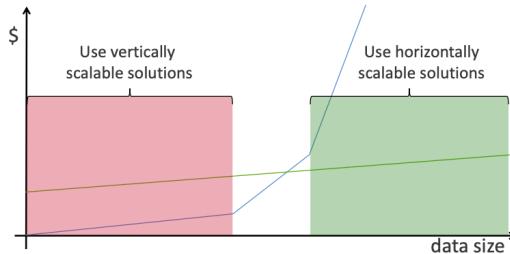


Figure 16: Vertical (Exponential) vs Horizontal (Linear) growth

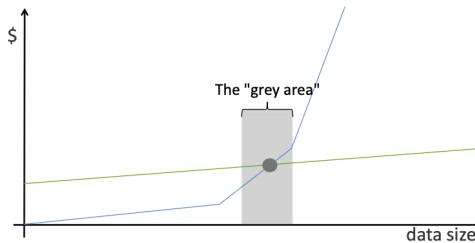
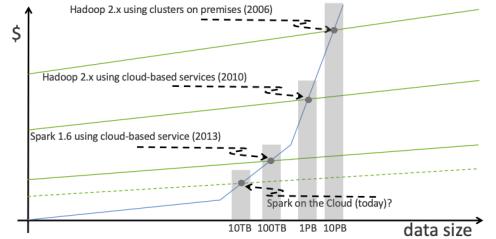


Figure 17: The space within vertical (blue) and horizontal (red) scalable solution. On the left we see an high price gap between the blue and the red line for which the preferred solution is vertical scalability. On the right is the contrary: the price is growing faster on the blue line while the horizontal scalable solution price is growing linearly.



2.2 ACID vs. BASE and SQL vs. NoSQL

2.2.1 Transactional Properties

Definition of Transaction: An elementary unit of work performed by an application. Each transaction is encapsulated within two commands: **begin transaction** (bot) and **end transaction** (eot).

Within a transaction of the commands below is executed *exactly once*: **commit work** (commit) and **rollback work** (abort).

A **Transactional System** (OLTP) is a system capable of providing the definition and execution of transactions on behalf of multiple, concurrent applications.

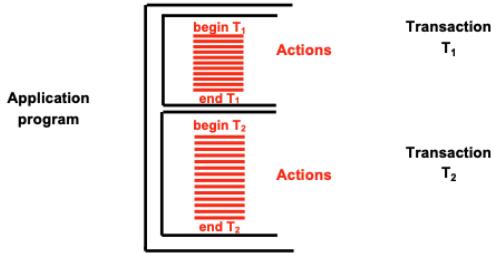


Figure 19: Application and Transactions

```

begin transaction;
update Account
    set Balance = Balance + 10
where AccNum = 12202;
update Account
    set Balance = Balance - 10
where AccNum = 42177;
commit work;
end transaction;

```

Figure 20: Transaction example

```

begin transaction;
update Account
    set Balance = Balance + 10 where AccNum =
12202;
update Account
    set Balance = Balance - 10 where AccNum =
42177;
select Balance into A from Account
    where AccNum = 42177;
if (A>=0) then commit work
else rollback work;
end transaction;

```

Figure 21: Another Transaction example with rollback

ACID Properties of Transactions A transaction is a unit of work enjoying the following properties:

- **Atomicity:** a transaction is an atomic transformation from the initial state to the final state. Three possible behaviors:
 - Commit work: SUCCESS
 - Rollback work or error prior to commit: UNDO
 - Fault after commit: REDO

Abort-rollback restart and Commit protocols

- **Consistency:** the transaction satisfies the integrity of constraints on data. As a consequence, if the initial state is consistent, then the final state is also consistent. **Integrity checking of DBMS**
- **Isolation:** a transaction is not affected by the behavior of other, concurrent transactions. As a consequence, its intermediate states are not exposed and the "domino effect" is avoided. **Concurrency control**
- **Durability:** the effect of a transaction that has successfully committed will last "forever" independently of any system fault. **Recovery management**

These properties characterizes relational DBMS and for this reason such systems offer very expensive and rigid solutions.

2.2.2 CAP Theorem

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- **Consistency**: all nodes see the same data at the same time
- **Availability**: node failures do not prevent other survivors from continuing to operate (a guarantee that every request receives a response about whether it succeeded or failed)
- **Partition tolerance**: the system continues to operate despite arbitrary partitioning due to network failures (e.g., message loss)

A distributed system can satisfy any two of these guarantees at the same time but not all three. In a distributed system, a network (of networks) is inevitable (by definition). We can't avoid to deal with partition tolerance, we need to cover that. Indeed. Failures can, and will, occur to a networked system. Then, the only option left is choosing between **Consistency** and **Availability**. This because CA doesn't make any sense, because is what traditional centralized database are guaranteeing.

We have two solutions:

- AP: a partitioned node returns
 - a correct value, if in a consistent state;
 - a timeout error or an error, otherwise;
 - e.g., DynamoDB, CouchDB, and Cassandra
- CP: a partitioned note returns the most recent version of the data, which could be stale
 - e.g., MongoDB, Redis, AppFabric Caching and MemcacheDB

By the way, Consistency and Availability should not necessarily be guaranteed in a mutually exclusive manner, but possibly by partial accomodation of both. We need to do some trade-off analyses.

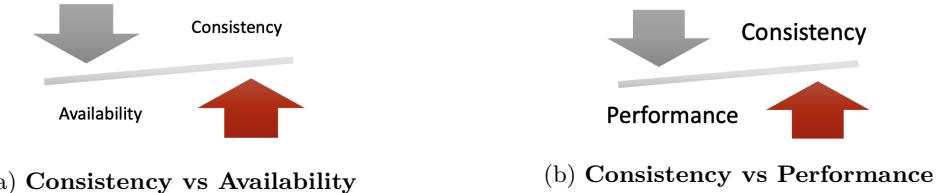


Figure 22: We need to choose between consistency and availability. According to the use case scenario, we can choose which one to favour. For example, consistency should be preferred in banking applications, where the transactions of money should be carefully saved and stored in a rigid flow to allow the correct functioning of the system. While almost all the social media apps or streaming platforms may concentrate on availability since if some data in the communication is lost or some user content are not presented in the latest version, the app can continue providing the service without creating any big issues to the user. Talking about performance, high consistency usually results in low performance while high performance results in low consistency.

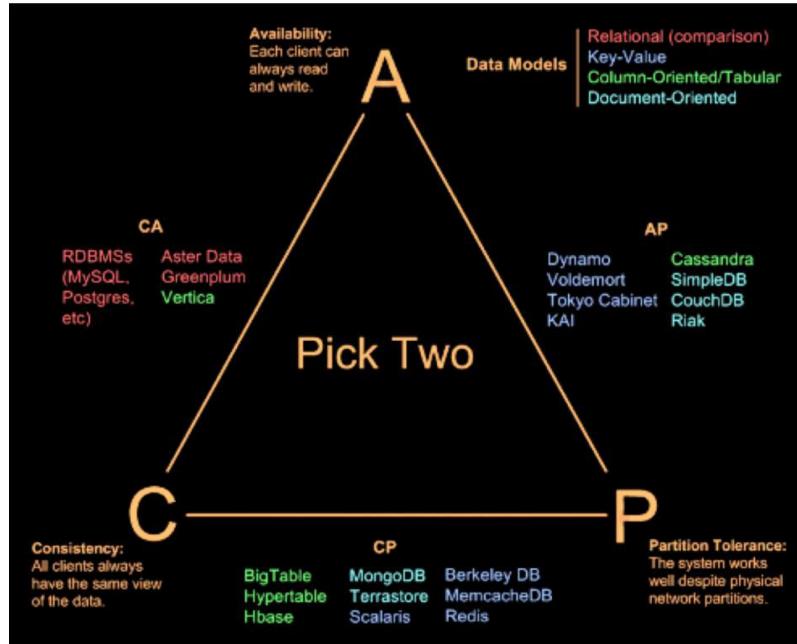


Figure 23: Visual Guide to CAP Theorem

2.2.3 ACID vs. BASE properties

SQL databases:

- Structured query language
- Traditional relational databases (unique keys, single valued, no update/insertion/deletion anomalies)
- Well structured data
- ACID properties should hold

NoSQL (Not Only SQL) databases:

- Triggered by the storage needs of Web 2.0 companies such as Facebook, Google and Amazon.com
- Not necessarily well structured - e.g., pictures, documents, web page description, video clips, etc.
- ACID properties may not hold, but this does not mean that there are no properties at all (there are new properties)
- focuses on availability of data even in the presence of multiple failures
- spread data across many storage systems with a high degree of replication

BASE properties are much weaker properties w.r.t. to ACID ones. The rationale behind them is that it's ok to use stale data and it's okay to give approximate answers.

- Basic Availability: fulfill request, even in partial consistency. Basic functionalities are always provided.
- Soft State: abandon the consistency requirements of the ACID model pretty much completely.
- Eventual consistency: at some point in the future, data will converge to a consistent state; delayed consistency, as opposed to immediate consistency of the ACID properties.
 - purely a liveness guarantee (reads eventually return the requested value);
 - no safety guarantees, i.e., an eventually consistent system can return any value before it converges

2.2.4 The NoSQL World

Google, Amazon, Facebook, and DARPA all recognized that when you scale systems large enough, you can never put enough iron in one place to get the job done (and you wouldn't want to, to prevent a single point of failure). Once you accept that you have a distributed system, you need to give up consistency or availability, which the fundamental transactionality of traditional RDBMSs cannot abide. - Cedric Beust

The acronym **NoSQL** was first used in 1998 by Carlo Strozzi while naming his lightweight, open-source "relational" database that did not use SQL. NoSQL term was used to say that he was not using an SQL interface.

The term was then reintroduced in early 2009, when Eric Evans and Johan Oskarsson used it to describe non-relational databases (which are often referred to as SQL systems). In that case the term was meaning "not only SQL" to emphasize the fact that some systems might even support SQL-like query languages.

Kind of NoSQL NoSQL solutions fall into two major areas:

- **Key/Value** or "the big hash table"
 - Amazon S3 (Dynamo)
 - Voldemort
 - Scalaris
 - Memcache DB
 - Azure Table Storage
 - Redis
 - Riak
- **Schema-less**
 - Cassandra (column-based)
 - CouchDB (document-based)
 - Neo4J
 - HBase

Different types of NoSQL

- **Key-Value Store:** A key that refers to a payload (actual content /data).
MemcacheDB, Azure Table Storage, Redis
- **Column Store:** column data is saved together, as opposed to row data. Super useful for data analytics.
Hadoop, Cassandra, Hypertable
- **Document / XML / Object Store:** key (and possibly other indexes) point at serialized object. DB can operate against values in document.
MongoDB, CouchDB, RavenDB
- **Graph Store:** nodes are stored independently, and the relationship between nodes (edges) are stored with data.
Neo4J

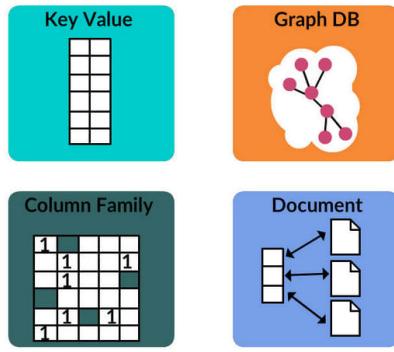


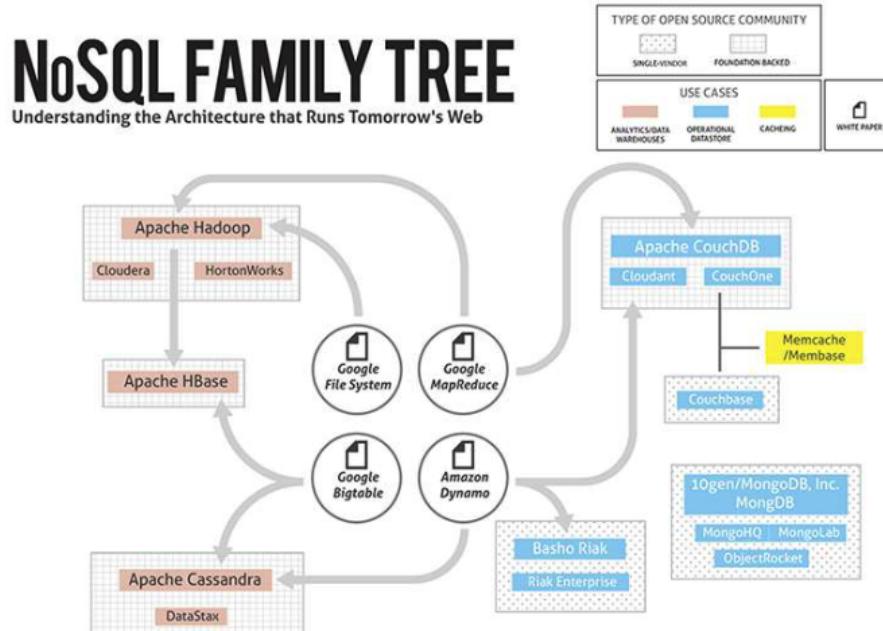
Figure 24: NoSQL types

Most of NoSQL databases are open source projects started and/or supported by the most famous companies in the world, cause they needed to create custom solutions to improve their performance.

- Google → BigTable, LevelDB
- LinkedIn → Voldemort
- Facebook → Cassandra
- Twitter → Hadoop/HBase, FlockDB, Cassandra
- Netflix → SimpleDB, Hadoop/HBase, Cassandra
- CERN → CouchDB

This is a big shift from traditional SQL-based produced by companies such as Oracle, IBM and Microsoft. They are still selling traditional, relational and transactional DBMS. Indeed, their projects are not open source.

In conclusion, there is no general answer to whether your application needs an ACID versus BASE consistency model. Given BASE's loose consistency, developers need to be more **knowledgeable and rigorous about consistent data** if they choose a BASE store for their application. Planning around BASE limitations can sometimes be a major disadvantage when compared to the simplicity of ACID transactions. A fully ACID database is the perfect fit for use cases where data reliability and consistency are essential.



3 Graph DB

Graph databases address one of the great macroscopic business trends of today: leveraging complex and dynamic relationships in highly connected data to generate insight and competitive advantage. For data of any significant size or value, graph databases are the best way to represent and query connected data. Connected data is data whose interpretation and value requires us first to understand the ways in which its constituent elements are related.

Although large corporations realized this some time ago and began creating their own proprietary graph processing technologies, we are now in an era where that technology has rapidly become democratized. Today, general-purpose graph databases are a reality, enabling mainstream users to experience the benefits of connected data without having to invest in building their own graph infrastructure.

Graph theory was pioneered by Euler in the 18th century, and has been actively researched and improved by mathematicians, sociologists, anthropologists, and other practitioners ever since. However, it is only in the past few years that graph theory and graph thinking have been applied to information management. In that time, graph databases have helped solve important problems in the areas of social networking, master data management, geospatial, recommendations, and more. This increased focus on graph databases is driven by two forces: by the massive commercial success of companies such as Facebook, Google, and Twitter, all of whom have centered their business models around their own proprietary graph technologies; and by the introduction of general-purpose graph databases into the technology landscape.

3.1 Graph Theory

Formally, a graph is just a collection of *vertices* and *nodes* – or, in other words, a set of *nodes* and the *relationships* that connect them. Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships. This general-purpose expressive structure allows us to model all kinds of scenarios that we can imagine. Indeed, graphs are extremely useful in understanding a wide diversity of datasets in fields such as science, government, and business. For example, Twitter’s data is represented as a graph. In Figure 25 we see a small network of Twitter users. Each node is labeled *User*, indicating its role in the network. These nodes are then connected with relationships, which help further establish the semantic context: namely, that Billy follows Harry, and that Harry, in turn, follows Billy. Ruth and Harry likewise follow each other, but sadly, although Ruth follows Billy, Billy hasn’t (yet) reciprocated.

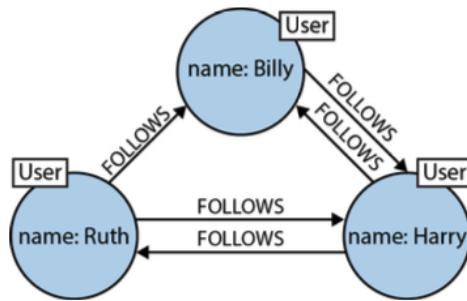


Figure 25: A small social graph

Of course, Twitter’s real graph is hundreds of millions of times larger than the example in Figure 25, but it works on precisely the same principles. In Figure 26 we’ve expanded the graph to include the messages published by Ruth.

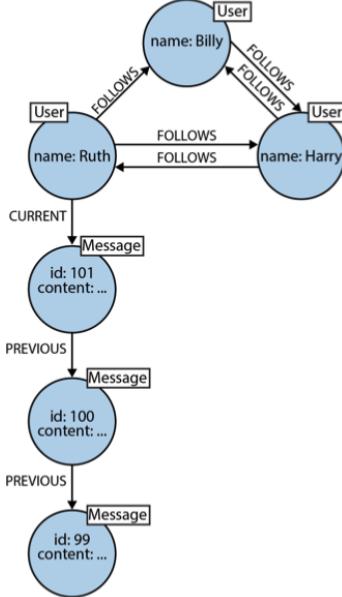


Figure 26: Publishing messages

Though simple, Figure 26 shows the expressive power of the graph model. It's easy to see that Ruth has published a string of messages. Her most recent message can be found by following a relationship marked CURRENT. The PREVIOUS relationships then create Ruth's timeline.

In discussing Figure 26 we've also informally introduced the most popular form of graph model, the **labeled property graph**. A labeled property graph has the following characteristics:

- It contains nodes and relationships
- Nodes contain properties (key-value pairs)
- Nodes can be labeled with one or more labels
- Relationships are named and directed, and always have a start and end node
- Relationships can also contain properties

3.1.1 Useful definitions

Vertex

- Basic element
- Drawn as a node or a dot
- Vertex set of G is usually denoted by $V(G)$, or V

Edge

- A set of two elements
- Drawn as a line connecting two vertices, called end vertices, or endpoints
- The edge set of G is usually denoted by $E(G)$, or E

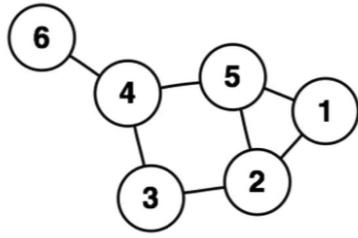


Figure 27: $V := \{1, 2, 3, 4, 5, 6\} - E : \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}$

Simple graphs: simple graphs are graphs without multiple edges or self-loops.

Path: a path is a sequence of vertices such that there is an edge from each vertex to its successor. A path is simple if each vertex is distinct.

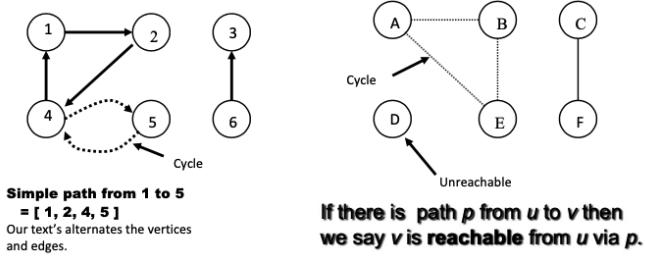


Figure 28: Graph path and reachability

Cycle: a path from a vertex to itself is called cycle. A graph is called cyclic if it contains a cycle; otherwise it is called acyclic

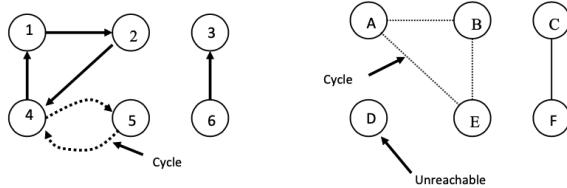


Figure 29: Graph cycle

Connectivity: a graph is connected if and only if

- you can get from any node to any other by following a sequence of edges OR
- any two nodes are connected by a path

A directed graph is strongly connected if there is a directed path from any node to any other node.

Sparse/Dense

- A graph is sparse if $|E| \approx |V|$ (same number of edges and vertices \rightarrow very few connections)
- A graph is dense if $|E| \approx |V|^2$ (graph full of connections \rightarrow with n vertices, there can be a max of $n(n - 1)$ edges)

Weighted Graph: is a graph for which each edge has an associated weight, usually given by a weight function $w : E \rightarrow R$.

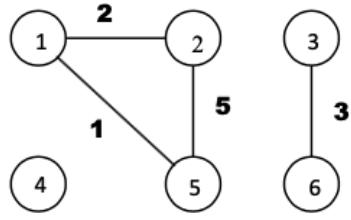


Figure 30: For example, in a GPS navigator we could use weight to specify duration, distance or traffic. The shortest path between two nodes is then calculated selecting the edges with the lowest weight.

Directed Graph: edges have directions and the arch can be followed only in that direction

Bipartite Graph: V can be partitioned into 2 sets V_1 and V_2 such that $(u, v) \in E$ implies:

- either $u \in V_1$ and $v \in V_2$
- OR $v \in V_1$ and $u \in V_2$

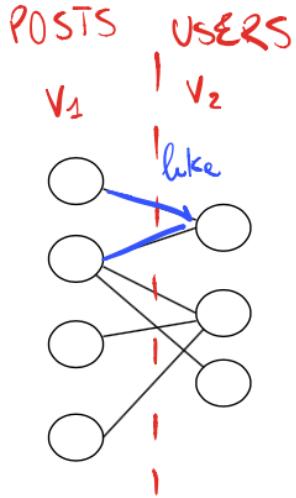


Figure 31: Nodes in V_1 connects only with nodes in V_2 (usually those are different categories of nodes e.g., Users and Posts)

Complete Graph: denoted by K_n , in a complete graph every pair of vertices are adjacent with a total of $n(n - 1)$ edges.

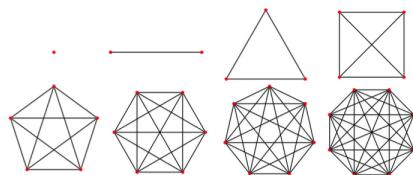


Figure 32: Exponential growth

Planar Graph: can be drawn on a plane such that no two edges intersect.

Tree: is a connected acyclic graph where two nodes have exactly one path between them.

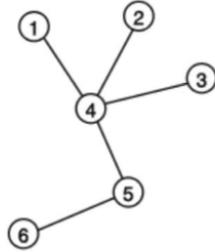


Figure 33: Example of Tree

Degree: number of edges incident on a node

Degree (directed graph):

- In degree: number of edges entering the node
- Out degree: number of edges leaving the node
- $Degree = indegree + outdegree$

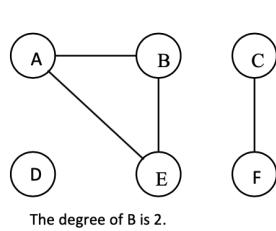


Figure 34: Node degree

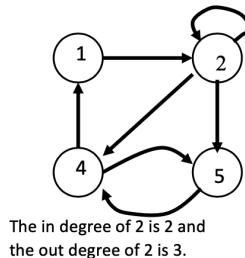


Figure 35: Node degree in directed graph

Subgraph: vertex and edge sets are subsets of those of G; a supergraph of a graph G is a graph that contains G as a subgraph.

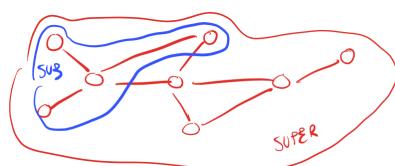


Figure 36: Subgraph and Supergraph

3.1.2 Graph Abstract Data Type (ADT)

In computer science, a graph is an abstract data type (ADT) that consists of:

- a set of nodes
- a set of edges (establish relationships/connections between the nodes)

The graph ADT follows directly from the graph concept from mathematics. We can implement a graph as a:

- Matrix
 - Incidence Matrix - [edge, vertex] contains the edge's data
 - Adjacency Matrix - [vertex, vertex] boolean values (adjacent or not) or edge weights
- List
 - Edge List - pairs (ordered if directed) of vertices and optionally weight and other data
 - Adjacency List

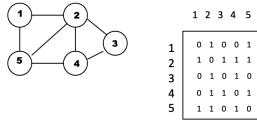


Figure 37: $|V| \times |V|$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise.

3.2 Graph Databases

A **graph database management system** (henceforth, a graph database) is an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

There are two properties of graph databases we should consider when investigating graph database technologies:

- *The underlying storage:*
Some graph databases use native graph storage that is optimized and designed for storing and managing graphs. Not all graph database technologies use native graph storage, however. Some serialize the graph data into a relational database, an object-oriented database, or some other general-purpose data store.
- *The processing engine:*
Some definitions require that a graph database use index-free adjacency, meaning that connected nodes physically “point” to each other in the database. Here we take a slightly broader view: any database that from the user’s perspective behaves like a graph database (i.e., exposes a graph data model through CRUD operations) qualifies as a graph database. We do acknowledge, however, the significant performance advantages of index-free adjacency, and therefore use the term native graph processing to describe graph databases that leverage index-free adjacency.

It’s important to note that native graph storage and native graph processing are neither good nor bad – they’re simply classic engineering trade-offs. The benefit of native graph storage is that its purpose-built stack is engineered for performance and scalability. The benefit of nonnative graph storage, in contrast, is that it typically depends on a mature nongraph backend (such as MySQL) whose production characteristics are well understood by operations teams. Native graph processing (index-free adjacency) benefits traversal performance, but at the expense of making some queries that don’t use traversals difficult or memory intensive.

Relationships are first-class citizens of the graph data model. This is not the case in other database management systems, where we have to infer connections between entities using things like foreign keys or out-of-band processing such as map-reduce. By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build arbitrarily sophisticated models that map closely to our problem domain. The resulting models are simpler and at the same time more expressive than those produced using traditional relational databases and the other NoSQL (Not Only SQL) stores.

3.2.1 Advantages of Graph Databases

Performance: One compelling reason, then, for choosing a graph database is the sheer performance increase when dealing with connected data versus relational databases and NoSQL stores. In contrast to relational databases, where join-intensive query performance deteriorates as the dataset gets bigger, with a graph database performance tends to remain relatively constant, even as the dataset grows. This is because queries are localized to a portion of the graph. As a result, the execution time for each query is proportional only to the size of the part of the graph traversed to satisfy that query, rather than the size of the overall graph.

Flexibility: As developers and data architects, we want to connect data as the domain dictates, thereby allowing structure and schema to emerge in tandem with our growing understanding of the problem space, rather than being imposed upfront, when we know least about the real shape and intricacies of the data. Graph databases address this want directly.

Graphs are naturally additive, meaning we can add new kinds of relationships, new nodes, new labels, and new subgraphs to an existing structure without disturbing existing queries and application functionality. These things have generally positive implications for developer productivity and project risk. Because of the graph model's flexibility, we don't have to model our domain in exhaustive detail ahead of time – a practice that is all but foolhardy in the face of changing business requirements. The additive nature of graphs also means we tend to perform fewer migrations, thereby reducing maintenance overhead and risk.

Agility: We want to be able to evolve our data model in step with the rest of our application, using a technology aligned with today's incremental and iterative software delivery practices. Modern graph databases equip us to perform frictionless development and graceful systems maintenance. In particular, the schema-free nature of the graph data model, coupled with the testable nature of a graph database's application programming interface (API) and query language, empower us to evolve an application in a controlled manner.

At the same time, precisely because they are schema free, graph databases lack the kind of schema-oriented data governance mechanisms we're familiar with in the relational world. But this is not a risk; rather, it calls forth a far more visible and actionable kind of governance. Governance is typically applied in a programmatic fashion, using tests to drive out the data model and queries, as well as assert the business rules that depend upon the graph.

Sailor		Reserve		Boat	
sid	sname	rating	age	bld	bname
22	dustin	7	45.0	22	101
31	lubber	8	55.5		10/10/96
58	rusty	10	35.0	58	103
					11/12/96
				bld	color
				101	Interlake
				102	Clipper
				103	Marine
					red
					green
					red

Figure 38: Relational DB with intermediate join table



(:Sailor) -[:reserves]-> (:Boat)

Figure 39: Graph DB model. Much simpler!

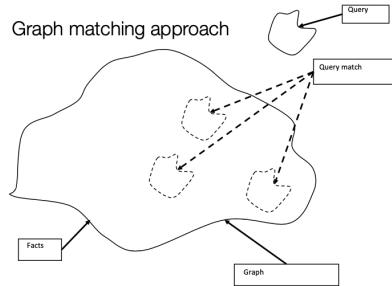


Figure 40: Querying a graph is similar to pattern matching. First, we define a pattern (shape of (sub)graph that we are looking for) and then we look in the graph for that shape.

3.3 Neo4J

Neo4J is the most popular graph database, developed by Neo Technologies and implemented in Java. It is fully open source.

Salient features

- **Neo4J is schema free:** data does not have to adhere to any convention
- **ACID:** atomic, consistent, isolated and durable for logical units of work (fully transactional solution)
- Easy to get started and use
- Well documented and large developer community
- Support for wide variety of languages (Java, Python, Perl, Scala, Cypher, etc.)

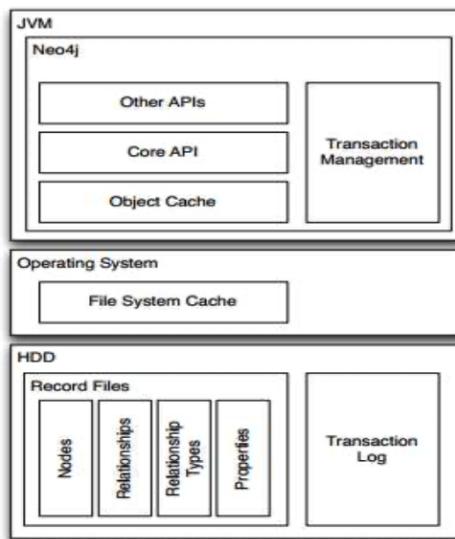


Figure 41: Neo4J software architecture. Each type of record (e.g., nodes, relationships) is stored in a separate and dedicated file. Traditional **Consistency** and **Availability** support (no partitioning).

Neo4J is meant to be an operational DB, not specifically for analytics. Thus, it is efficient on nodes and patterns, while is not so efficient in whole-graph analysis.

The data model is composed by

- Nodes – with labels (type) and attributes
- Edges
- Indexes (different from the ones in standard relational db) indexe

3.3.1 Cypher

Cypher is an expressive (yet compact) graph database query language. Cypher is arguably the easiest graph query language to learn, and is a great basis for learning about graphs. Cypher is designed to be easily read and understood by developers, database professionals, and business stakeholders. Its ease of use derives from the fact that it is in accord with the way we intuitively describe graphs using diagrams.

Cypher enables a user (or an application acting on behalf of a user) to ask the database to find data that matches a specific pattern. Colloquially, we ask the database to “find things like this.” And the way we describe what “things like this” look like is to draw them, using ASCII art.

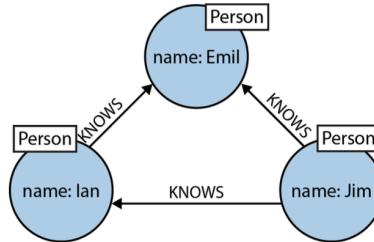


Figure 42: This pattern describes three mutual friends. Here’s the equivalent ASCII art representation in Cypher: `(emil)←[:KNOWS]−(jim)−[:KNOWS]→(ian)−[:KNOWS]→(emil)`

Cypher patterns follow very naturally from the way we draw graphs on the whiteboard.

The previous Cypher pattern describes a simple graph structure, but it doesn’t yet refer to any particular data in the database. To bind the pattern to specific nodes and relationships in an existing dataset we must specify some property values and node labels that help locate the relevant elements in the dataset. For example:

```

(emil:Person {name:'Emil'})
  <-[:KNOWS]-(jim:Person {name:'Jim'})
  -[:KNOWS]->(ian:Person {name:'Ian'})
  -[:KNOWS]->(emil)

```

Figure 43: Here we have bound each node to its identifier using its *name* property and *Person* label. The *emil* identifier, for example, is bound to a node in the dataset with a label *Person* and a *name* property whose value is *Emil*. Anchoring parts of the pattern to real data in this way is normal Cypher practice.

Like most query languages, Cypher is composed of clauses. The simplest queries consist of a *MATCH* clause followed by a *RETURN* clause (we’ll describe the other clauses you can use in a Cypher query later in this chapter). Here’s an example of a Cypher query that uses these three clauses to find the mutual friends of a user named Jim:

```

MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c),
      (a)-[:KNOWS]->(c)
RETURN b, c

```

Other Cypher clauses:

- **WHERE**: provides criteria for filtering pattern matching results.
- **CREATE** and **CREATE UNIQUE**: create nodes and relationships.
- **MERGE**: ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates, or by creating new nodes and relationships.
- **DELETE**: removes nodes, relationships, and properties.

- **SET**: sets property values.
- **FOREACH**: performs an updating action for each element in a list.
- **UNION**: merges results from two or more queries.
- **WITH**: chains subsequent query parts and forwards results from one to the next. Similar to piping commands in Unix.
- **START**: specifies one or more explicit starting points – nodes or relationships – in the graph. (START is deprecated in favor of specifying anchor points in a MATCH clause.)

```
Query:
MATCH (n:Crew)-[r:KNOWS*]-m
WHERE n.name='Neo'
RETURN n AS Neo,r,m
```

Figure 44: **Example query.** Find a node n of type *Crew* connected to m with relations r of type *Knows* (from 1-step to $*$ -steps in the relation)

- MATCH (user)-[:FRIEND]-{friend}
- WITH user, count(friend) AS friends
- ORDER BY friends DESC
- SKIP 1 LIMIT 3
- RETURN user

Figure 45: **Example query.** Aggregation can be used (count). WITH separates query parts explicitly, to declare the variables for the next part. SKIP skips results at the top and LIMIT limits the number of results.

(n:Person)	Node with Person label.
(n:Person:Swedish)	Node with both Person and Swedish labels.
(n:Person {name: \$value})	Node with the declared properties.
()-[r {name: \$value}]-()	Matches relationships with the declared properties.
(n)-->(m)	Relationship from n to m .
(n)--(m)	Relationship in any direction between n and m .
(n:Person)-->(m)	Node n labeled Person with relationship to m .

(m)<-[:KNOWS]-(n)	Relationship of type KNOWS from n to m .
(n)-[:KNOWS :LOVES]->(m)	Relationship of type KNOWS or of type LOVES from n to m .
(n)-[r]->(m)	Bind the relationship to variable r .
(n)-[*1..5]->(m)	Variable length path from 1 to 5 rels. from n to m .
(n)-[*]->(m)	Variable length path of any number of rels. from n to m
(n)-[:KNOWS]->(m {property: \$value})	A relationship of type KNOWS from a node n to a node m with the declared property.

Figure 46: List of patterns

Stored procedures Cypher support stored procedures. It allows you to add JAVA functions or simply move a JAR to a folder to add new functions. Of course, there are also some predefined function such as *shortestPath*, *allShortestPaths*, *size*.

Hints

- **Use parameters instead of literals** when possible. This allows Cypher to re-use your queries instead of having to parse and build new execution plans.
- **Always set an upper limit for your variable length patterns.** It's easy to have a query touch all nodes in a graph by mistake.
- **Return only the data you need.** Avoid returning whole nodes and relationships
- Use **PROFILE / EXPLAIN** to analyze the performance of your queries.

4 Key-Value DB

The main motivation behind Key-Value databases is performance. Indeed, there are certain organizations/companies that cannot accept low performances:

- Amazon - Every 1/10 second delay resulted in 1% loss of sales.
- Google - Half a second delay caused a 20% drop in traffic
- Industrial Group - 1-second delay in page-load time
 - 11% fewer page views
 - 15% decrease in customer satisfaction
 - 7% loss in conversions

Search by ID is usually built on top of a key-value store.

- (Business) Key → Value (follow this schema)
- (twitter.com) tweet → information about tweet
- (kayak.com) flight number → information about flight
- (yourbank.com) account number → information about it
- (amazon.com) item number → information about it

4.1 How does a key-value database work?

A key-value database, aka key-value store, associates a value (which can be anything from a number or simple string, to a complex object) with a key, which is used to keep track of the object. In its simplest form, a key-value store is like a dictionary/array/map object as it exists in most programming paradigms, but which is stored in a persistent way and managed by a Database Management System (DBMS).

Key-value databases use compact, efficient index structures to be able to quickly and reliably locate a value by its key, making them ideal for systems that need to be able to find and retrieve data in constant time. Redis, for instance, is a key-value database that is optimized for tracking relatively simple data structures (primitive types, lists, heaps, and maps) in a persistent database. By only supporting a limited number of value types, Redis is able to expose an extremely simple interface to querying and manipulating them, and when configured optimally is capable of extremely high throughput.

4.1.1 Key Features

A key-value database is defined by the fact that it allows programs or users of programs to retrieve data by keys, which are essentially names, or identifiers, that point to some stored value. Because key-value databases are defined so simply, but can be extended and optimized in numerous ways, there is no global list of features, but there are a few common ones:

- **Retrieving a value** (if there is one) stored and associated with a given key
- **Deleting the value** (if there is one) stored and associated with a given key
- **Setting, updating, and replacing the value** (if there is one) associated with a given key

4.2 Redis

REmote DIctionary Server (REDIS) introduced their key-value database in 2009.

Redis is an advanced key-value store, where keys can contain data structures such as strings, hashes, lists, sets, and sorted sets. Supporting a set of atomic operations on these data types. Redis is a different evolution path in the key-value databases where values are complex data types that are closely related to fundamental data structures and are exposed to the programmer as such, without additional abstraction layers.

It can be used as:

- **Database** - Redis can persist data to disk
- **Caching layer** - Redis is fast
- **Message Broker** - Redis is not only a key-value store

What is NOT Redis:

- **Redis is not a replacement for Relational Databases nor Document Stores.**
- **It might be used complementary to a SQL relational store, and/or NoSQL document store.**
- Even when Redis offers configurable mechanisms for persistency, increased persistency will tend to increase latency and decrease throughput.
- **Best used for rapidly changing data** with a foreseeable database size (should fit mostly in memory).

Redis use cases:

- Caching
- Counting things
- Blocking queues
- Pub/Sub (service bus)
- MVC Output Cache provider
- ASP.NET Session State provider
- Online user data (e.g., shopping cart, ...)
- ... any real-mine cross-platform, cross-application communication

When to consider Redis:

- **Speed is critical**
- More than just key-value pairs
- Dataset can fit in memory
- Dataset is not critical

Redis Data Type	Contains	Read/write ability
String	I'm a string! 0 1 1 0 0 0 0 ...	Binary-safe strings (up to 512 MB), Integers or Floating point values, Bitmaps. Operate on the whole string, parts, increment/decrement the integers and floats, get/set bits by position.
Hash	Key1 Value1 Key2 Value2	Unordered hash table of keys to string values Add, fetch, or remove individual items by key, fetch the whole hash.
List	A ← C → B → C	Doubly linked list of strings Push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value.
Set	D B C A	Unordered collection of unique strings Add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items.
Sorted Set	B: 0.1 D: 0.3 A: 250 C: 250	Ordered mapping of string members to floating-point scores, ordered by score Add, fetch, or remove individual items, fetch items based on score ranges or member value.
Geospatial index	Value Lat: 20.63373 Lon: -103.55328	Sorted set implementation using geospatial information as the score Add, fetch or remove individual items, search by coordinates and radius, calculate distance.
HyperLogLog	0 1 1 0 0 0 0 1 0 1 ...	Probabilistic data structure to count unique things using 12kb of memory Add individual or multiple items, get the cardinality.

Figure 47: Redis data types.

Strings	
Get/Set strings redis> SET foo "hello!" OK redis> GET foo "hello!"	SET [key value] / GET [key] O(1)
Increment numbers redis> SET bar 223 OK redis> INCRBY bar 1000 (integer) 1223	INCRBY [key increment] O(1)
Get multiple keys at once redis> MGET foo bar 1. "hello!" 2. "1223"	MGET [key key ...] O(N) : N=# of keys.
Set multiple keys at once > MSET foo "hello!" bar 1223 OK	MSET [key value key value ...] O(N) : N=# of keys.
Get the length of a string redis> STRLEN foo (integer) 6	STRLEN [key] O(1)

Keys	
Key removal redis> DEL foo (integer) 1	DEL [key ...] O(1)
Test for existence redis> EXISTS foo (integer) 1	EXISTS [key ...] O(1)
Get the type of a key redis> TYPE foo string	TYPE [key] O(1)
Rename a key redis> RENAME bar new_bar OK redis> EXPIRE foo 10 (integer) 1	RENAME [key newkey] O(1) O(1)
Get key time-to-live redis> TTL foo (integer) 10	TTL [key] O(1)

Figure 48: Redis commands. [Full command reference here](#)

4.2.1 Scaling Redis

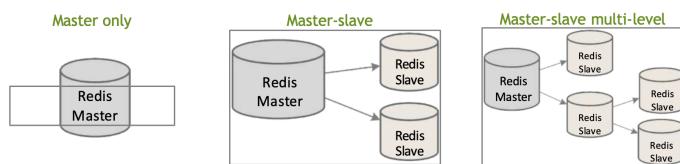
- **Persistence:** how can we be sure that we have some persistent storage of data (since Redis works in main memory)?
For this reason, REDIS provides two mechanisms to deal with persistence (very basic options):
 - Redis Database Snapshots (RDB): save memory snapshot on disk (like a backup)
 - append-only files (AOF): store in append mode the evolution of data
- **Replication:** a Redis instance known as the *master*, ensures that one or more instances known as the *slaves*, become exact copies of the master. Clients can connect to the master or to the slaves. Slaves are read only by default, while master allows both read and write operations.
- **Partitioning:** we need to deal with data separation, breaking up data and distributing it across different hosts in a cluster. It can be implemented in different layers:
 - Client: partitioning on client-side code
 - Proxy: an extra layer that proxies all redis queries and performs partitioning (i.e. [Twemproxy](#))
 - Query Router: instances will make sure to forward the query to the right node (i.e. [Redis Cluster](#))
- **Failover:** replace possible masters that are broken with a slave
 - Manual
 - Automatic with Redis Sentinel (for master-slave topology)
 - Automatic with Redis Cluster (for cluster topology)

4.2.2 Redis topologies

1. Standalone
2. Sentinel (automatic failover)
3. Twemproxy (distribute data)
4. Cluster (automatic failover and distribute data)

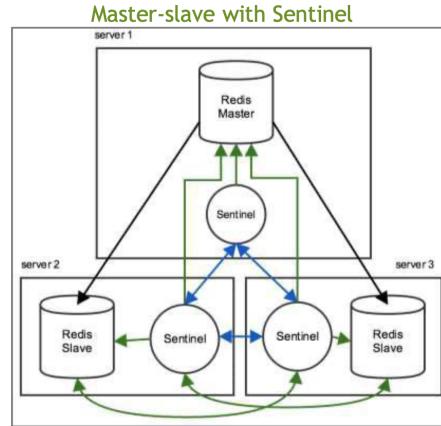
I - Standalone

- The master data is optionally replicated to slaves.
- The slaves provides data redundancy, reads offloading and save-to-disk offloading.
- Clients can connect to the Master for read/write operations or to the Slaves for read operations.
- Slaves can also replicate to its own slaves.
- There is no automatic failover.



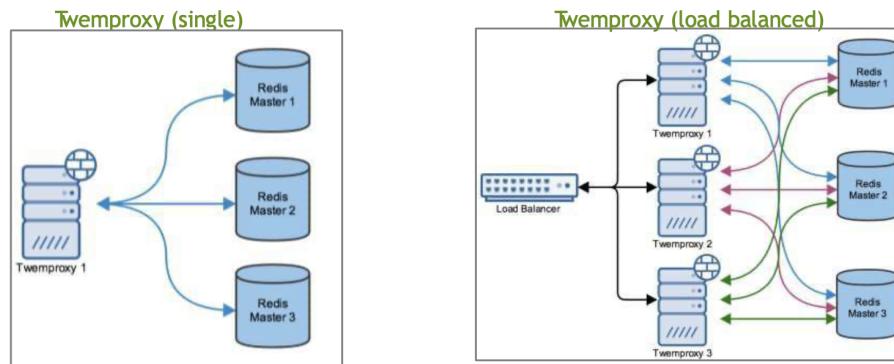
II - Sentinel

- Redis Sentinel provides a reliable **automatic failover** in a master/slave topology, automatically promoting a slave to master if the existing master fails.
- Every deployment (master or slave) have a sentinel component that is able to check the status of the other servers. If the master goes down, a slave is selected to become a new master automatically.
- Sentinel does not distribute data across nodes.



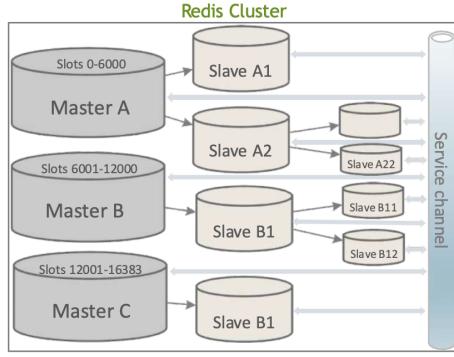
III - Twemproxy

- Twemproxy** (a project by twitter) works as a proxy between the clients and many Redis instances.
- Is able to **automatically distribute data** among different standalone Redis instances.
- Supports consistent hashing with different strategies and hashing functions
- Multi-key commands and transactions are not supported.



IV - Cluster

- Redis Cluster **distributed data** across different Redis instances and **perform automatic failover** if any problem happens to any master instance.
- All nodes are directly connected with a service channel.
- The keyspace is divided into hash slots. Different nodes will hold a subset of hash slot.
- Multi-key commands are only allowed for keys in the same hash slot.



4.2.3 Redis Advantages

- Performance
- Availability
- Fault-Tolerance
- Scalability (adaptability)
- Portability

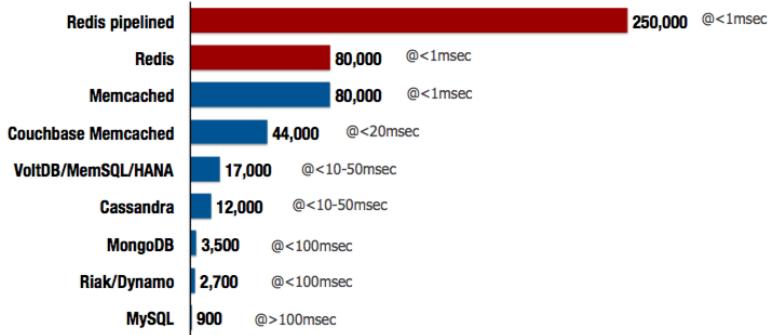


Figure 49: NoSQL & SQL response performance comparison.

Num of operations per unit of time - Average query time

4.3 Key-Value and Caching

4.3.1 What is Caching?

From Wikipedia:

"A cache is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (owing to longer access time) or to compute, compared to the cost of reading the cache."

Anatomy:

- Simple key/value storage
- Simple operations
 - save
 - get
 - delete

Terminology:

- Storage cost
- Retrieval cost (network load / algorithm load)
- Invalidation (keeping data up to date / removing irrelevant data)
- Replacement policy:
 - FIFO - First in First Out
 - LFU - Least Frequently Used (replace the cache entry used the least often in the recent past)
 - LRU - Least Recently Used (replace the least recently used items first)
 - MRU - Most Recently Used (replace, in contrast to LRU, the most recently used items first)
 - Random vs. Belady's algorithm (predicting the information that will not be needed for the longest time in the future and replace it)
- Cache concepts:
 - Cold Cache: when the cache is empty or has irrelevant data, so that CPU needs to do a slower read from main memory for your program data requirement.
 - Warm Cache: when the cache contains relevant data, and all the reads for your program are satisfied from the cache itself.
- Cache Hit and Cache Miss
 - Hit: when an application needs data and finds that data in the cache (avoiding to look for it in main memory)
 - Miss: when an application needs data and doesn't find it in the cache, so then it has to go and find the data on disc (takes more time).
- Typical stats:
 - $hit_ratio = hits/(hits + misses)$
 - $miss_ratio = 1 - hit_ratio$

When to cache?

- Caches are only efficient when the benefits of faster access outweighs the overhead of checking and keeping your cache up to date
- More cache hits than cache misses

Where are caches users?

- At hardware level (CPU, HDD)
- Operating systems (RAM)
- Web stack (browser cache, DNS cache, CDNs cache, application level)
- Applications

4.3.2 Memcached

- Free & open-source, high-performance, distributed memory object caching system
- Generic in nature, intended for use in speeding up dynamic web applications by alleviating database load.
- Key/Value dictionary
- Now used by Netlog, Facebook, Flickr, Wikipedia, Twitter, Youtube ...

Technically, Memcached is a server, where client access over TCP or UDP. Servers can run in pools and are independent, clients manage the pool (e.g., 3 servers with 64GB mem each give you a single pool of 192GB storage for caching).

What to store in a memcache?

- High demand (data used often)
- Expensive (data hard to compute)
- Common (data shared across users)
- typical examples:
 - user sessions (often)
 - user data (often, shared)
 - homepage data (often, shared, expensive)

Memcached principles

- Very simple version of a data store
- Lightweight technology with high-performance (comes at a cost)
- Fast network access: memcached servers close to other application servers
- No persistency: if your server goes down, data in memcached is gone
- No redundancy / fail-over
- No replication: single item in cache lives on one server only
- No authentication: not used in shared environments
- 1 key is maximum 1MB
- Keys are string of 250 characters
- No enumeration of keys: thus no list of valid keys in cache at certain moment)
- No active clean-up (only clean up when more space needed, LRU policy)

```
<?php

Memcached::add — Add an item under a new key
Memcached::addServer — Add a server to the server pool

Memcached::decrement — Decrement numeric item's value
Memcached::delete — Delete an item
Memcached::flush — Invalidate all items in the cache
Memcached::get — Retrieve an item
Memcached::getMulti — Retrieve multiple items
Memcached::getStats — Get server pool statistics
Memcached::increment — Increment numeric item's value
Memcached::set — Store an item

function getUserData($UID)
{
    $key = 'user_' . $UID;
    $userData = $cache->get($key);
    if (!$userData)
    {
        $queryResult = Database::query("SELECT * FROM USERS
WHERE uid = " . (int) $UID);
        $userData = $queryResult->getRow();
        $cache->set($userData);
    }
    return $userData;
}
?>
```

Figure 50: Memcached PHP Client functions.

Figure 51: Code for explicitly implementing caching.

5 Big Column DB

5.1 Introduction

5.1.1 Column wise vs. Row wise database

A **columnar database stores data by columns** rather than by rows, which makes it suitable for analytical query processing, and thus for data warehouses. They're often used in data warehouses, the structured data repositories that businesses use to support corporate decision-making.

The major difference in both the datastores (row- vs column-based) lies in the way they physically store the data on the disk. We know that persistent storage disks (hard disks) are organized in blocks and have following usual properties for reading/write operations. Head Seek operation is expensive in disks due to mechanical movement required. Read/Write is quite fast.

1. The whole block with data is loaded into the memory for reading by the operating system. Any further read for data for this block will happen from memory and will be super fast.
2. Read/Writing operations on disks are not slow. Only the seek operation is slow. i.e. to move the head to the correct block to perform the operation.
3. Due to the above point — sequential read/writes are much faster on disks rather than the random access.

Here comes the main difference between row and columnar DBs.

Row oriented database tries to store whole row of the database in the same block but columnar database stores the values of the columns of subsequent in the same block

Indeed, columnar storage for database tables is an important factor in optimizing analytic query performance because it drastically reduces the overall disk IO requirements and reduces the amount of data you need to load from disk.

The following series of illustrations (from [Amazon Redshift documentation](#)) describe how columnar data storage implements efficiencies and how that translates into efficiencies when retrieving data into memory.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

101259797|SMITH|88|899 FIRST ST|JUNO|AL 892375862|CHIN|37|16137 MAIN ST|POMONA|CA 318370701|HANDU|12|42 JUNE ST|CHICAGO|IL

Block 1 Block 2 Block 3

Figure 52: Row-wise

In a typical relational database table, each row contains field values for a single record. In row-wise database storage, data blocks store values sequentially for each consecutive column making up the entire row. In online transaction processing (OLTP) applications, most transactions involve frequently reading and writing all of the values for entire records, typically one record or a small number of records at a time. As a result, row-wise storage is optimal for OLTP databases.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

101259797 | 892375862 | 318370701 | 468248180 | 378568310 | 231346875 | 317346551 | 770336528 | 277332171 | 455124598 | 735885647 | 387586301

Block 1

Figure 53: Column-wise

Using columnar storage, each data block stores values of a single column for multiple rows. This means that reading the same number of column field values for the same number of records requires a third of the I/O operations compared to row-wise storage. In practice, using tables with very large numbers of columns and very large row counts, storage efficiency is even greater. An added advantage is that, since each block holds the same type of data, block data can use a compression scheme selected specifically for the column data type, further reducing disk space and I/O.

5.1.2 Column storage

Issues with today's workloads Column data storage were born to address the need of large scale data analysis.

- Data large and unstructured
- Lots of random reads and writes
- Foreign keys rarely needed (we use more complex data structures)
- Actual needs:
 - Incremental scalability
 - Speed
 - No single point of failure
 - Low cost (TCO) and admin
 - Scale out, not up

Recalling the CAP Theorem, for which we can achieve at most 2 out of the 3 guarantees (Consistency, Availability and Partition-tolerance), usually column databases (e.g., Cassandra) focus on Availability and Partition-tolerance, supporting only Eventual (weak) Consistency. Indeed, they are mainly used in OLAP systems (online analytical processing) and data mining operations.

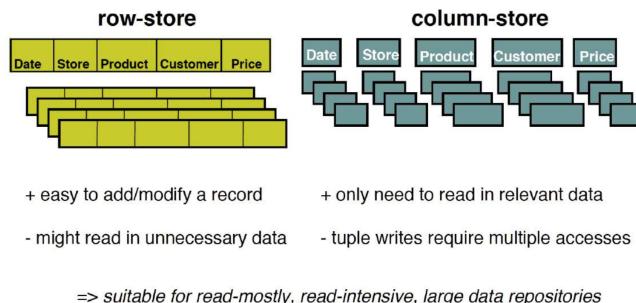


Figure 54: Column vs. Row data storage

Pros:

- Data compression (1000 TB compression come handy)
- Improved Bandwidth Utilization
- Improved Code Pipelining
- Improved Cache Locality

Cons

- Increased Disk Seek Time
- Increased cost of Inserts
- Increased tuple reconstruction costs

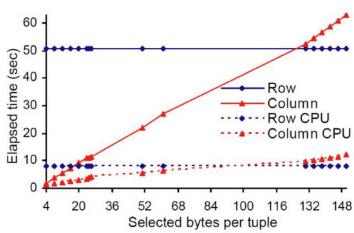


Figure 55: Row based always read the entire row (constant time). Column based instead are more efficient when few bytes have to be read. Then there is a breakeven point after which row based databases become more efficient to read data.

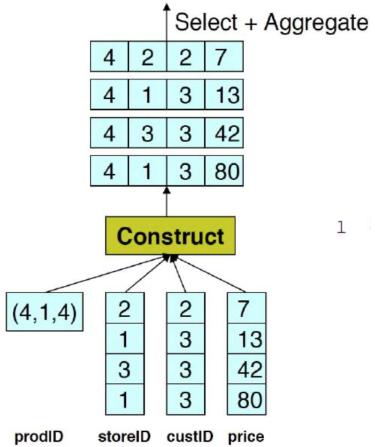


Figure 56: Tuple reconstruction.

Compression

Compression: find a better encoding without losing information

- Trades I/O for CPU
- Increased column-store opportunities:
 - Higher data value locality (spatial and temporal) in column stores, saving space and performance
 - Techniques such as *run length encoding* is far more useful
 - Can use extra space to store multiple copies of data in different sort orders

Example:

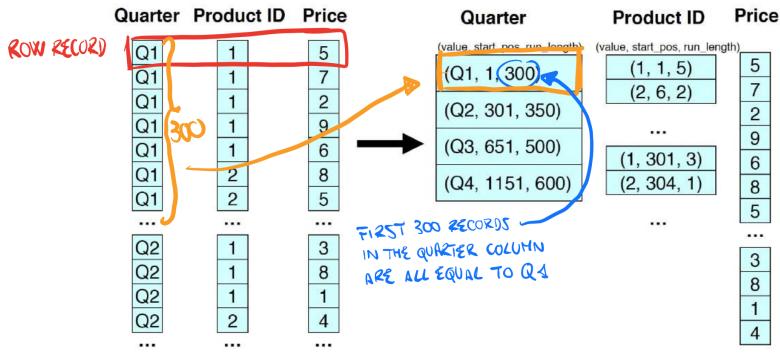


Figure 57: Column compression example using Run Length encoding

5.2 Cassandra

Cassandra is a columnar database which was originally designed at Facebook and then open-sourced (now within Apache foundation). Many big companies use Cassandra: IBM, eBay, twitter, Adobe, Netflix, Spotify ...
Cassandra can be considered an hybrid of Google's Bigtable (columnar) and Amazon's Dynamo (key-value). By the way it emphasizes a lot on columnar features.

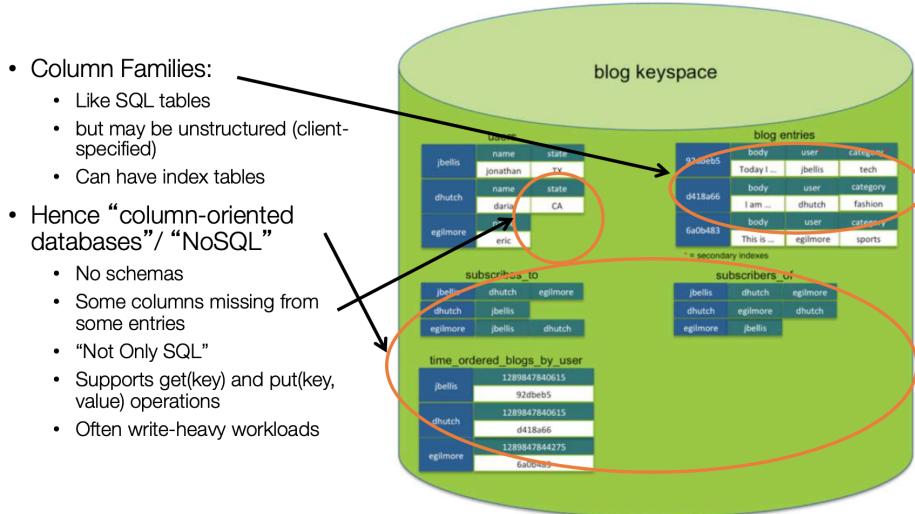


Figure 58: Cassandra Data Model

Property	Cassandra	RDBMS
Core Architecture	Masterless (no single point of failure)	Master-slave (single points of failure)
High Availability	Always-on continuous availability	General replication with master-slave
Data Model	Dynamic; structured and unstructured data	Legacy RDBMS; Structured data
Scalability Model	Big data/Linear scale performance	Oracle RAC or Exadata
Multi-Data Center Support	Multi-directional, multi-cloud availability	Nothing specific
Enterprise Search	Integrated search on Cassandra data.	Handled via Oracle search
In-Memory Database Option	Built-in in-memory option	Columnar in-memory option

Property	Cassandra	RDBMS
Joining	Doesn't support joining	Supports joining
Referential Integrity	Cassandra has no concept of referential integrity across tables. No cascading deletes.	Supports foreign keys in a table to reference the primary key of a another table. Supports cascading delete.
Normalization	Tables contain duplicate denormalize data.	Tables are normalized to avoid redundancy.

Figure 59: Cassandra properties w.r.t. RDBMS (e.g., Oracle)

5.2.1 Cassandra Properties

- highly available
- fault tolerant
- *tunably* consistent: allows to have some levels of consistency which can be tuned dynamically (trade-off with performance)
- very fast writes
- linear, elastic scalability
- **decentralized/symmetric** (no master/slave)
- automatic provisioning of new nodes
- $O(1)$ DHT: key-based query → constant complexity

5.2.2 Gossip Protocol

How does the Cassandra Cluster know which members are online and working?
It uses a [gossip protocol](#).

Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about. The gossip process runs every second and exchanges state messages with up to three other nodes in the cluster. The nodes exchange information about themselves and about the other nodes that they have gossiped about, so all nodes quickly learn about all other nodes in the cluster.

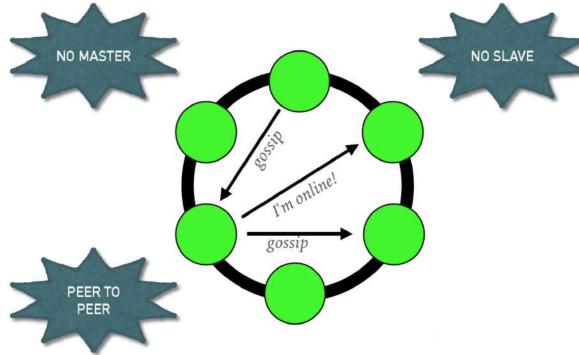


Figure 60: Cassandra Gossip Protocol

5.2.3 Replica Placement Strategies

As hardware problem can occur or link can be down at any time during data process, a solution is required to provide a backup when the problem has occurred. So data is replicated for assuring no single point of failure.

Cassandra places replicas of data on different nodes based on these two factors.

- Where to place next replica is determined by the **Replication Strategy**.
- While the total number of replicas placed on different nodes is determined by the **Replication Factor**.

One Replication factor means that there is only a single copy of data while three replication factor means that there are three copies of the data on three different nodes. For ensuring there is no single point of failure, **replication factor must be three**.

There are two kinds of replication strategies in Cassandra.

- **Simple Strategy:** used when you have just one data center. SimpleStrategy places the first replica on the node selected by the partitioner. After that, remaining replicas are placed in clockwise direction in the Node ring.

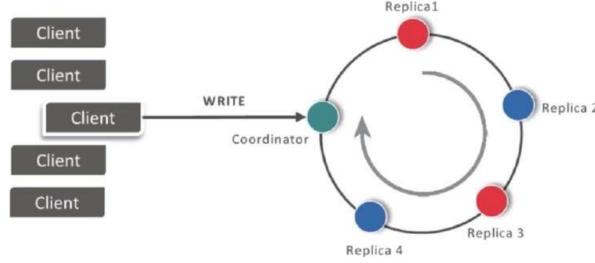
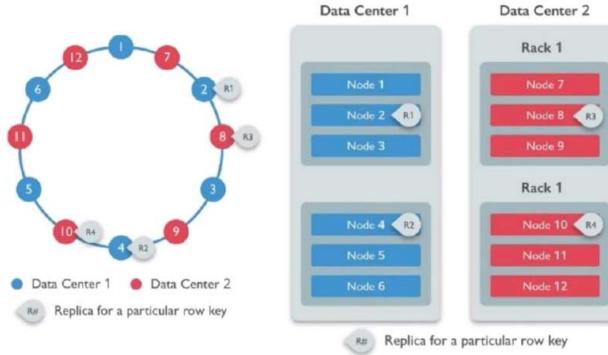


Figure 61: Cassandra's topology can be imagined as a ring, in which nodes are all equivalent (master-less). The process of replication is managed by a coordinator (who changes each time), who's in charge of handling the a write operation and then start propagating the new value to the others. The replicas in turn, will continue propagating the new value in an asynchronous way, following the clockwise direction.

- **Network Topology Strategy:** is used when you have more than two data centers. In NetworkTopologyStrategy, replicas are set for each data center separately. NetworkTopologyStrategy places replicas in the clockwise direction in the ring until reaches the first node in another rack. This strategy tries to place replicas on different racks in the same data center. This is due to the reason that sometimes failure or problem can occur in the rack. Then replicas on other nodes can provide data.



5.2.4 Write operation

For performance reasons, write operations need to be lock-free and fast (no reads or disk seeks). A Client sends a write to one front-end node in Cassandra cluster (coordinator). The coordinator forwards the request to replicas that are responsible for that key.

- Always writable: Hinted Handoff
 - If any replica is down, the coordinator writes to all other replicas, and keeps the write until down replica comes back up.
 - When all replicas are down, the Coordinator buffers writes (up to an hour).
- Provides Atomicity for a given key (i.e., within ColumnFamily)

Consistency level determines how many nodes will respond back with the success acknowledgment. The node will respond back with the success acknowledgment if data is written successfully to the commit log and **mem-table**.

For example, in a single data center with replication factor equals to three, three replicas will receive write request. If consistency level is one, only one replica will respond back with the success acknowledgment, and the remaining two will remain dormant. Suppose if remaining two replicas lose data due to node downs or some other problem, Cassandra will make the row consistent by the built-in repair mechanism in Cassandra.

Here it is explained, how write process occurs in Cassandra:

1. When write request comes to the node, first of all, it logs in the commit log.
2. Then Cassandra writes the data in the mem-table. Data written in the mem-table on each write request also writes in commit log separately. Mem-table is a temporarily stored data in the memory while Commit log logs the transaction records for back up purposes.
3. When mem-table is full or old, data is flushed to the SSTable data file.

5.2.5 Read operation

Are we sure that copies in replicas are aligned? That's why read operations may be slower than writes because they need to touch log and multiple SSTables to check if the data is correct.

There are three types of read requests that a coordinator sends to replicas.

- Direct request
- Digest request
- Read repair request

The coordinator sends direct request to one of the replicas. After that, the coordinator sends the digest request to the number of replicas specified by the consistency level and checks whether the returned data is an updated data.

After that, the coordinator sends digest request to all the remaining replicas. If any node gives out of date value, a background read repair request will update that data. This process is called read repair mechanism.

5.2.6 Cassandra Quorums and Consistency Levels

What if we have different values for the same data? Play with **majority quorums**.

Cassandra's tunable consistency comes from the fact that it allows per-operation tradeoff between consistency and availability through consistency levels.

The following consistency levels are available:

- *ONE* – Only a single replica must respond.
- *TWO* – Two replicas must respond.
- *THREE* – Three replicas must respond.
- *QUORUM* – A majority ($n/2 + 1$) of the replicas must respond.
- *ALL* – All of the replicas must respond.
- *LOCAL_QUORUM* – A majority of the replicas in the local datacenter (whichever datacenter the coordinator is in) must respond.
- *EACH_QUORUM* – A majority of the replicas in each datacenter must respond.
- *ANY* – A single replica may respond, or the coordinator may store a hint. If a hint is stored, the coordinator will later attempt to replay the hint and deliver the mutation to the replicas. This consistency level is only accepted for write operations.

Level	Description
ZERO	Good luck with that
ANY	1 replica (hints count)
ONE	1 replica. read repair in bkgnd
QUORUM (DCQ for RackAware)	(N /2) + 1
ALL	N = replication factor

Figure 62: Cassandra write consistency

Level	Description
ZERO	Ummm...
ANY	Try ONE instead
ONE	1 replica
QUORUM (DCQ for RackAware)	Return most recent TS after (N /2) + 1 report
ALL	N = replication factor

Figure 63: Cassandra read consistency

5.2.7 Data model

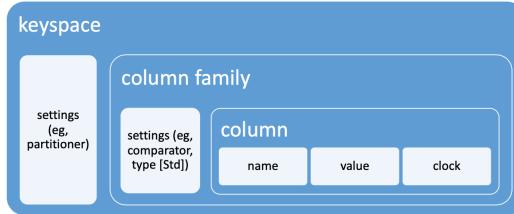


Figure 64: Cassandra data model

The **keyspace** is the entire DB, the space in which we define the keys for the elements and typically there is one keyspace per applications. Each **column** consists of a name (key), a value and a clock timestamp which indicates the update time. Furthermore every column is part of a **column family**. Indeed, a column family is a group of records of *similar* kind of elements (not the *same* kind, because CFs are **sparse tables**). Some examples of column families are: User, Address, Tweet, PointOfInterest, HotelRoom.

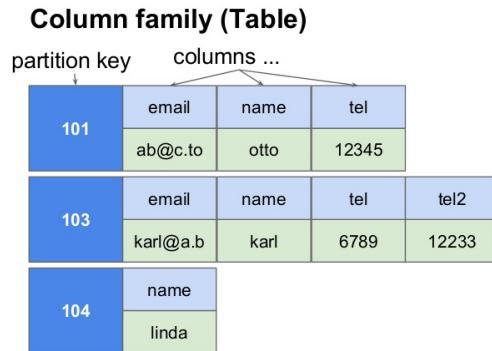


Figure 65: Cassandra column families are sparse tables. Some rows may contain all columns, while in other rows there could be only some of them. As in this case, row 104 has only one column while row 103 has 4 columns.

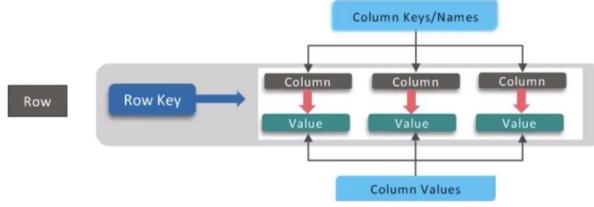


Figure 66: Cassandra: hybrid key-value based + column based database.

Cassandra, as we mentioned earlier, is an hybrid hybrid key-value based and column based database. Indeed, we can see that each row has a key and is composed by a set of columns (the value). Somehow we could say that Cassandra is a key-value based database in which the value is internally structured as columns.

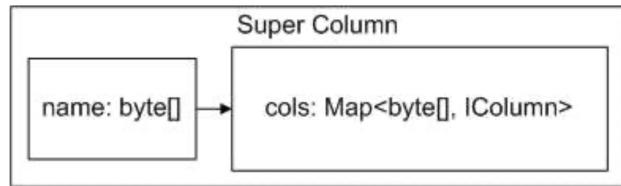


Figure 67: Cassandra supercolumn

In Cassandra there's also the concept of **super column**. Super columns group columns under a common name.

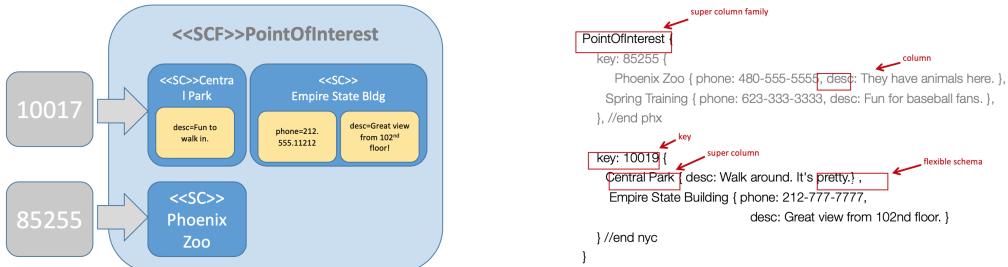


Figure 68: Supercolumn families

5.2.8 What about...SQL?

- **RDBMS:** domain-based-model → *what answers do I have?* (schema on write)
 1. Create the schema
 2. Perform query
 3. Check answer/result
- **Cassandra:** *query-based model* → *what questions do I have?* (schema on read)
 1. Plan the query
 2. Create the schema
 3. Check answer/result

In Cassandra we start defining queries and then we design the data model. Indeed, we define Cassandra as an *index factory*.

```
<<cf>>USER
Key: UserID
Cols: username, email, birth date, city, state

How to support this query?

SELECT * FROM User WHERE city = 'Scottsdale'

Create a new CF called UserCity:
<<cf>>USERCITY
Key: city
Cols: IDs of the users in that city.
Also uses the Valueless Column pattern
```

Figure 69: Given the schema defined above, how could we support the presented query? We just create a new column family *UserCity* which supports our query. In this case we are querying on the city attribute, that's why the new column family contains all the IDs of the users in that city.

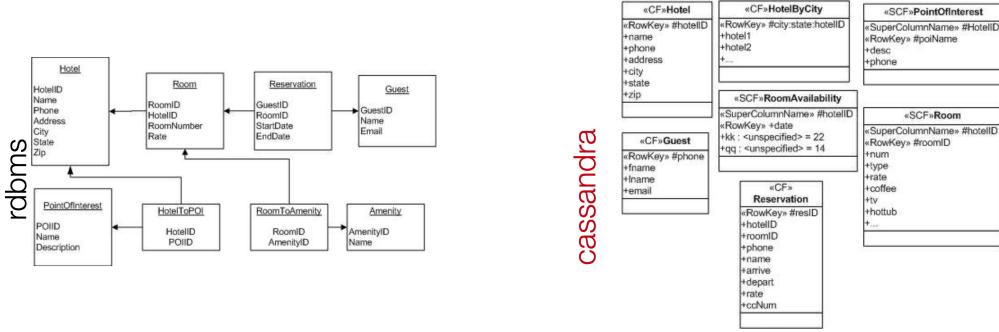


Figure 70: RDBMS schema are rigid schema defined when building up the database. Cassandra schema is more flexible because new column families are continuously added according to query needs.

5.3 Is Cassandra a good fit?

Cassandra is a good fit when:

- you need really fast writes
- you need durability
- you have lots of data
 - $> GBs$ (e.g., billions of tweets)
 - more than three servers)
- your app is evolving (startup mode, fluid data structure)
- loose domain data ("point of interest")
- your programmers can deal with
 - documentation
 - complexity
 - consistency model
 - change
 - visibility tools

Vs. SQL With more than 50GB of data:

- MySQL
 - Writes 300ms avg
 - Reads 350ms avg
- Cassandra
 - Writes 0.12ms avg
 - Reads 15ms avg

6 Document-oriented DB

6.1 Why document-based?

What makes document databases different from relational databases?

1. Intuitive Data Model: Faster and Easier for Developers

Documents map to the objects in your code, so they are much more natural to work with. There is no need to decompose data across tables, run expensive JOINs, or integrate a separate ORM layer. Data that is accessed together is stored together, so you have less code to write and your users get higher performance.

2. Flexible Schema: Dynamically Adapt to Change

A document's schema is dynamic and self-describing, so you don't need to first pre-define it in the database. Fields can vary from document to document and you modify the structure at any time, avoiding disruptive schema migrations. Some document databases offer JSON Schema so you can optionally enforce rules governing document structures.

3. Universal: JSON Documents are Everywhere

Lightweight, language-independent, and human readable, JSON has become an established standard for data interchange and storage. Documents are a superset of all other data models so you can structure data any way your application needs – rich objects, key-value pairs, tables, geospatial and time-series data, and the nodes and edges of a graph. You can work with documents using a single query language, giving you a consistent development experience however you've chosen to model your data.

4. Powerful: Query Data Anyway You Need

An important difference between document databases is the expressivity of the query language and richness of indexing. The MongoDB Query Language is comprehensive and expressive. Ad hoc queries, indexing, and real time aggregations provide powerful ways to access, transform, and analyze your data. With ACID transactions you maintain the same guarantees you're used to in SQL databases, whether manipulating data in a single document, or across multiple documents living in multiple shards.

5. Distributed: Resilient and Globally Scalable

Unlike monolithic, scale-up relational databases, document databases are distributed systems at their core. Documents are independent units which makes it easier to distribute them across multiple servers while preserving data locality. Replication with self-healing recovery keeps your applications highly available while giving you the ability to isolate different workloads from one another in a single cluster. Native sharding provides elastic and application-transparent horizontal scale-out to accommodate your workload's growth, along with geographic data distribution for data sovereignty.

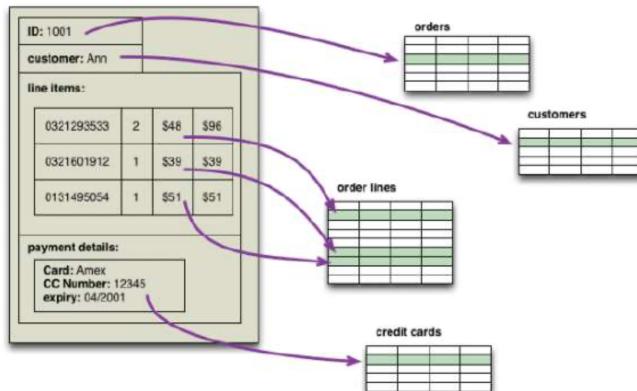


Figure 71: Example of document (invoice) stored in a document-based database.

The document in Figure 71 is made of nested data objects which (hypothetically) corresponds to different database tables. We decide to use document-based databases if we usually want to

retrieve the data in an aggregate way, avoiding complex joins between table. Indeed, we query the database and we obtain a document with all its subdocuments included.

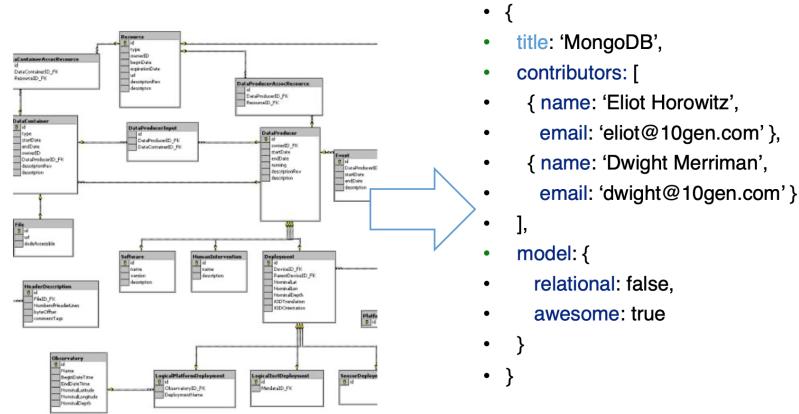


Figure 72: Example of JSON document

6.2 MongoDB

- An open source and document-oriented database
- Data is stored in JSON-like documents
- Designed with both scalability and developer agility
- Dynamic schemas
- Automatic data sharding

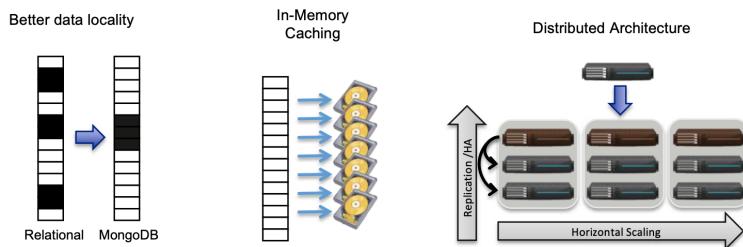


Figure 73: MongoDB features.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document
column	field
index	index
table joins (e.g. select queries)	embedded documents and linking
Primary keys	_id field is always the primary key
Aggregation (e.g. group by)	aggregation pipeline

Figure 74: MongoDB terminology vs SQL.

6.2.1 Facts

- No schemas
- No transactions
- No joins
- Max document size of 16MB (larger documents are handled with GridFS)
- Runs on most common OSs (Windows, Linux, Mac)
- Data stored as BSON (Binary JSON)
 - used for speed
 - translation handled by language drivers

6.2.2 Data Model

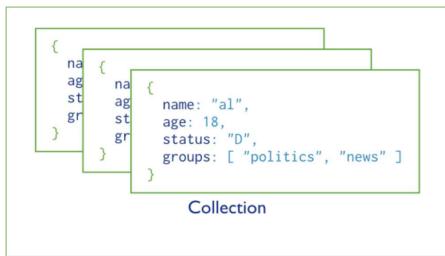


Figure 75: A collection includes a set of documents.

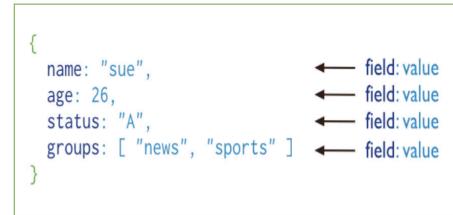


Figure 76: Structure of a JSON-document. The value of field could be one of: native data types, arrays, other documents. Rule: every document must have an `_id`.



Figure 77: Embedded documents.

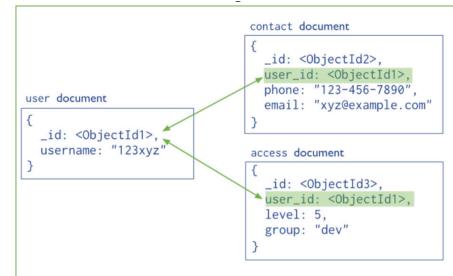


Figure 78: Reference documents or linking documents.

6.2.3 Queries

SQL Statement	MongoDB commands
SELECT * FROM table	db.collection.find()
SELECT * FROM table WHERE artist = 'Nirvana'	db.collection.find({Artist:"Nirvana"})
SELECT* FROM table ORDER BY Title	db.collection.find().sort>Title:1
DISTINCT	.distinct()
GROUP BY	.group()
>=, <	\$gte, \$lt

Figure 79: Read queries compared to SQL.

Name	Description
\$eq	Matches value that are equal to a specified value
\$gt, \$gte	Matches values that are greater than (or equal to) a specified value
\$lt, \$lte	Matches values less than or (equal to) a specified value
\$ne	Matches values that are not equal to a specified value
\$in	Matches any of the values specified in an array
\$nin	Matches none of the values specified in an array
\$or	Joins query clauses with a logical OR returns all
\$and	Join query clauses with a logical AND
\$not	Inverts the effect of a query expression
\$nor	Join query clauses with a logical NOR
\$exists	Matches documents that have a specified field

Figure 80: Comparison operators.

6.2.4 CAP Theorem and Mongo

Relative to the CAP theorem, MongoDB is a **CP** data store—it resolves network partitions by maintaining consistency, while compromising on availability.

MongoDB is a single-master system—each replica set can have only one primary node that receives all the write operations. All other nodes in the same replica set are secondary nodes that replicate the primary node's operation log and apply it to their own data set. By default, clients also read from the primary node, but they can also specify a read preference that allows them to read from secondary nodes.

When the primary node becomes unavailable, the secondary node with the most recent operation log will be elected as the new primary node. Once all the other secondary nodes catch up with the new master, the cluster becomes available again. As clients can't make any write requests during this interval, the data remains consistent across the entire network.

7 Streaming Data Engineering

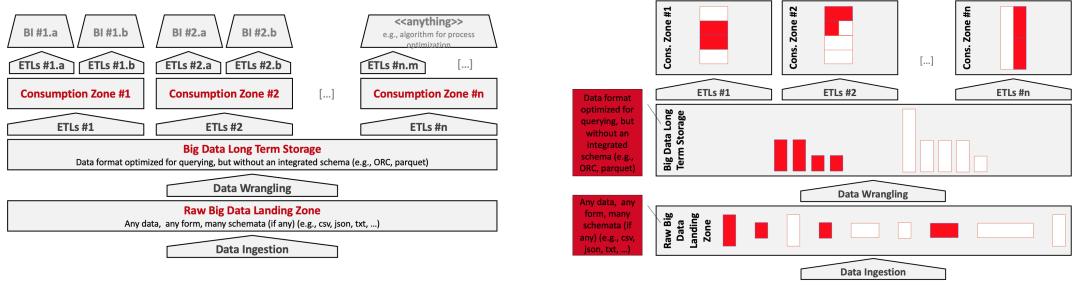
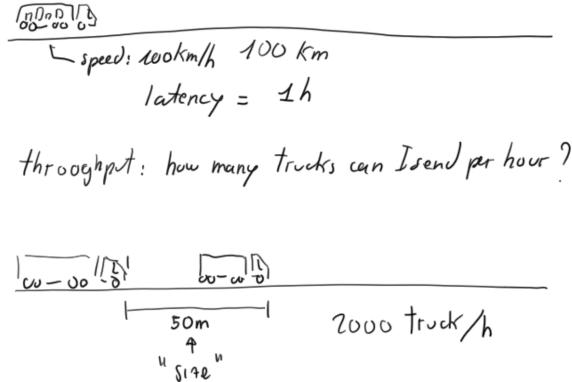


Figure 81: Logical architecture of a Big Data Platform.

Big data architecture is the foundation for big data analytics. It is the overarching system used to manage large amounts of data so that it can be analyzed for business purposes, steer data analytics, and provide an environment in which big data analytics tools can extract vital business information from otherwise ambiguous data. The big data architecture framework serves as a reference blueprint for big data infrastructures and solutions, **logically defining how big data solutions will work, the components that will be used, how information will flow, and security details**.

7.1 The Solution Space

7.1.1 The Dimensions: Throughput vs. Latency vs. Message size



In Figure 82, trucks are transporting data (they can carry a certain amount of data). Data can be ingested in our system at a certain speed, expecting different latency according to the size to be ingested.

Throughput indicates how much data we are able to ingest in a given time interval. It is the main performance indicator that we aim to optimize.

We have three options to optimize throughput:

- Increase trucks' speed
- Add lanes to the streets (parallelize)
- Reduce trucks' size (compress data)

- o Drive faster
 $100 \text{ km/h} \rightarrow 150 \text{ km/h}$
 - latency: 40 min
 - throughput: 3000 truck/h
- o Add lanes to the street
 -
 - latency: 1h
 - throughput: 4000 truck/h
- o Reduce the size
 -
 - latency: 1h
 - throughput: 4000 truck/h

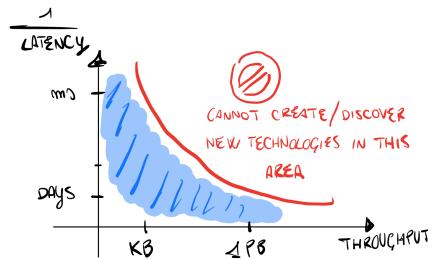


Figure 82: Latency-Throughput trade-off

7.1.2 Three Cases along a continuum

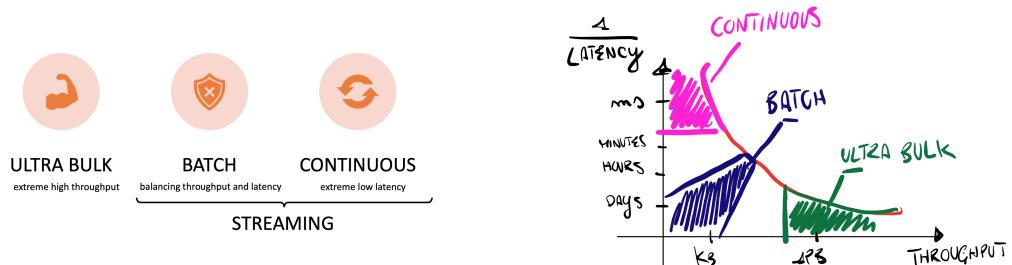


Figure 83: Three cases and their latency-throughput trade-off.

Ultra Bulk Case: high throughput and high latency

Transfer Appliance is a secure, rackable high capacity storage server that you set up in your datacenter. You fill it with data and ship it to an ingest location where the data is uploaded to Google Cloud Storage. Your data is encrypted automatically, and remains safe until you decrypt it.

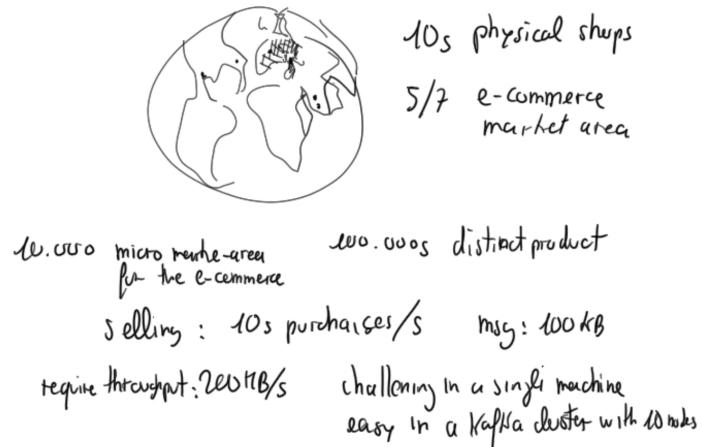


Figure 84: With [Google Cloud Transfer Appliance](#), you need only 2 days to upload 1PB to cloud.

Continuous Case low throughput and low latency

Example: Real Time Inventory

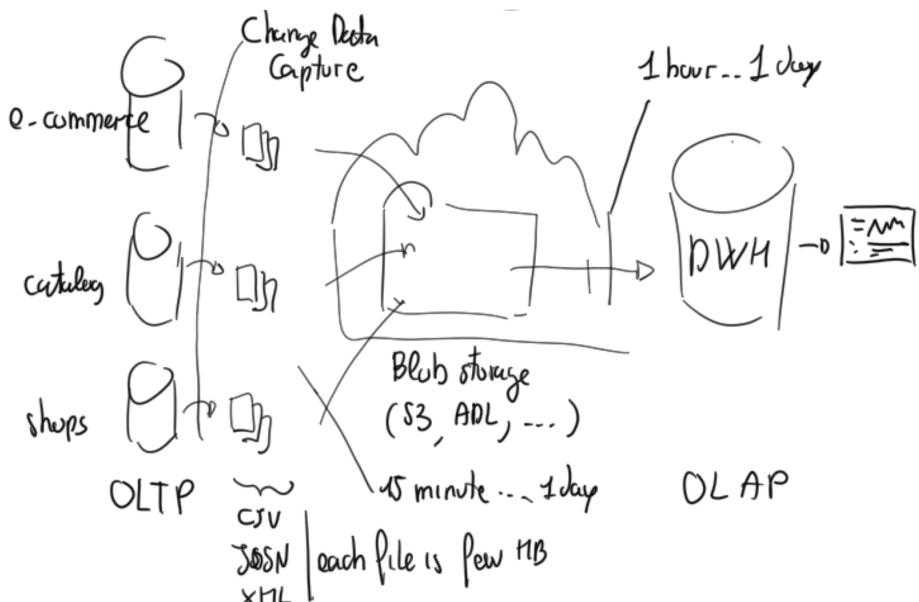
A company has 10k shops distributed all over the planet and an e-commerce site divided in 5/7 areas, with a total of 100k distinct products. To optimize the shipping time they further transform each shop in a shipping point for their products, thus we now have 10k micro market area for the e-commerce (one per each shop). They receive around 10k purchase/s and each message has a size of 100KB. The required throughput for the system is 200MB/s, which is easy to achieve with a Kafka cluster with 10 nodes.



Batch Case balanced throughput-latency trade-off

Example: customer 360° journey

Here a customer accessing to an e-commerce is sending request to three different subsystems which are handling the different section of the size. Each modification performed on one of the subsystems is captured and stored in a blob storage service and then asynchronously propagated to a data warehouse for analytics purposes. The latency of the process can vary from minutes to days.



7.2 The Batch Case

In a distributed system dealing with Big Data (where complex failure patterns can happen) we usually apply **Write Once, Read Many principles**

- allows reliable and parallel writing
- simplifies data coherency issues
- enables high throughput data access.

As a consequence:

- All data are kept in **immutable files**
- **Append-only operations** are allowed
- **Deletion of entire files** is also possible

Then we have two options:

- The **OVERWRITE** option: overwriting ingested table each time and using a view to map all the ingested table to a common structure
- The **BATCH** option: the process differs based on
 - **1st batch** – the first time a file is ingested & wrangled
 - **Next batch** – every time a new part of an existing file is ingested & wrangled

Big Data projects oriented to Data Warehouse enhancement call

- **history table** (or, shortly, hist table) the 1st batch of a given type ingested & wrangled
- **batch** the new parts of an existing file ingested & wrangled in the next iterations

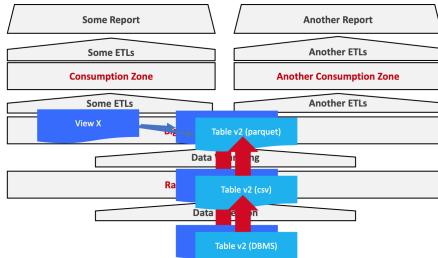


Figure 85: Overwrite option

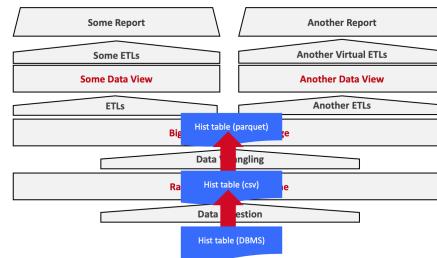
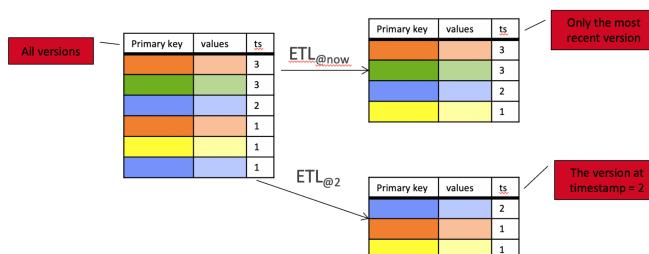


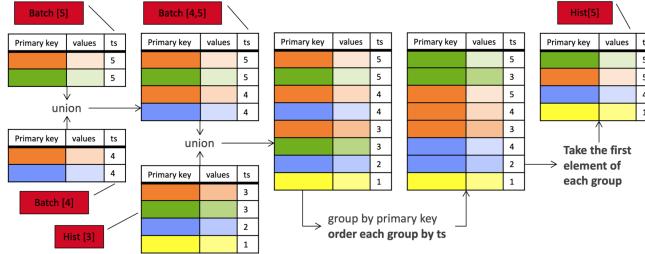
Figure 86: Batch option (1st batch case)

In the case of *Next batch* we have two alternatives in ingesting and wrangling batches:

- **Append mode**: the approach most often used in Big Data consists in **appending batches at the end of hist table**, i.e. changes are treated as inserts at a given timestamp.
Building a consistent view at a given point in time is left to ETLs that apply the schema on read.



- **Compaction mode** (maintain only the most recent version for each key): Big Data projects oriented to Data Warehouse enhancement often uses the compaction mode (keeping only the most recent version) inherited from Data Warehouse practice, i.e. **changes are treated as updates**. This is challenging given that files are immutable. ETLs, which apply the schema on read, always access the most recent consistent version of each table (old versions are discarded).



	pros	cons
Append	<ul style="list-style-type: none"> Minimal ingestion & wrangling cost The version of the data in any point in time can be rebuilt 	<ul style="list-style-type: none"> It requires more space than keeping only the most recent version If multiple ETLs access the same data the same computation may be repeated <ul style="list-style-type: none"> Avoidable by introducing a view that computes the shared data (e.g. ETL@now)
Compaction	<ul style="list-style-type: none"> It requires less space than keeping all versions as in append only mode Multiple ETLs can access the same data (@now) without additional computation 	<ul style="list-style-type: none"> It requires more computation than ingestion and wrangling in append only mode Only the most recent version is available, build version in arbitrary points in time is impossible

Figure 87: Append vs. Compaction mode

7.3 The Continuous Case

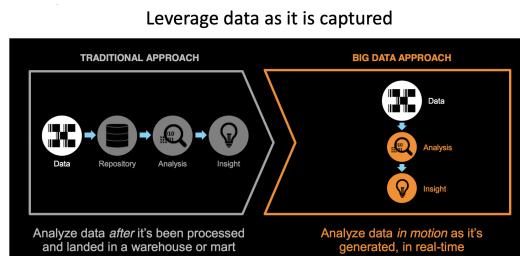


Figure 88: The continuous case is based on this paradigm shift.

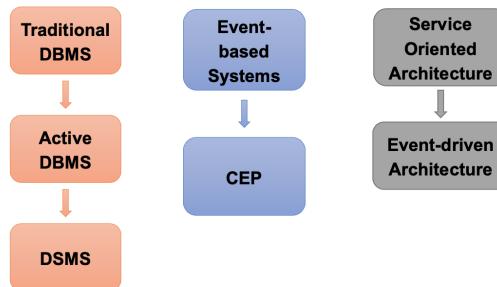


Figure 89: Many path to the same destination...

7.3.1 From Passive to Active DBMS and DSMS

- Standard DBMSs
 - Purely passive: *Human-active, database-passive (HADP)*
 - Execution happens only when asked by clients (through queries)
- Active DBMSs
 - The reactive behavior moves (in part) from the application to the DB layer ...
 - ...which executes Event Condition Action (ECA) rules (similar to triggers)

Active DBMSs

- As a DBMS extension
 - Rules may only refer to the internal state of the DB
- Closed DB applications
 - Rules may support the semantics of the application, but external sources events are not allowed
 - But events may come from external sources
- Open DB applications
 - Events may come from external sources

Data Stream Management Systems (DSMS) Data streams are (*unbounded*) sequences of time-varying data elements. They represent:

- an (almost) "continuous" flow of information (no silence)
- with the recent information being more relevant as it describes the current state of a dynamic system

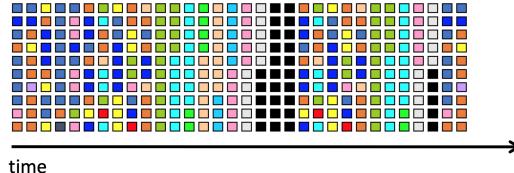


Figure 90: Unbounded window of data: up to a certain point is finite but it grows infinitely

The nature of streams requires a paradigmatic change **from persistent data** (one time semantics) **to transient data** (continuous data flow, with static queries).

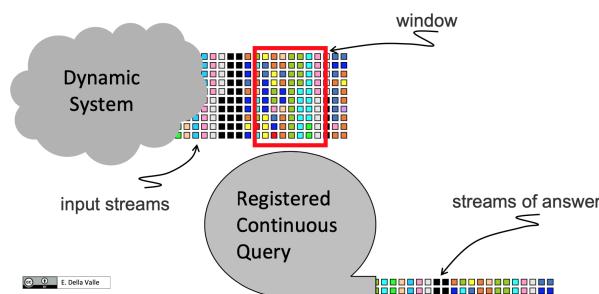


Figure 91: Continuous queries registered over streams that are observed through windows.

Time Model: relationship between information items and passing of time

Ability of an Information Flow Processing (IFP) system to associate some kind of *happened-before* (ordering) relationship to information items. There are 3 classes:

- **Causal** - this happened before than that



Figure 92: We don't know the distance between e1 and e2, we only know that e2 is after e1. The order can be exploited to perform queries like *Does Alice meet Bob before Carl?*

- **Absolute** - distance before events

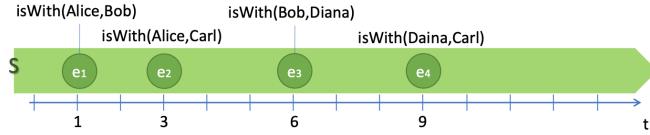


Figure 93: We can ask the queries presented in the causal model. WE can start to compose queries taking into account the time *How many people has Alice met in the last 5m* (window of 5mins opened in the past) or *Does Diana meet Bob and then Carl withing 5m?* (window opened in the future).

- **Interval** - event that last for a given amount of time

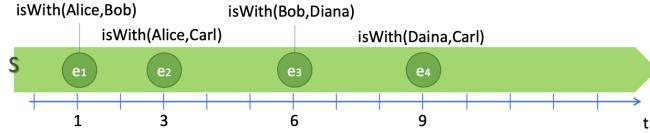


Figure 94: We can ask the queries presented in the previous cases. It is possible to write even more complex queries like *Which are the meetings that last less than 5m?* or *Which are the meetings with conflicts?*

7.3.2 Event-based systems

An event is something happened that our application needs to react to. Changing the customer address, making a purchase, or calculating the customer bill, are all events. These events might come from the external world or triggered internally such as having a scheduled job that is being executed every some time.

Components collaborate by exchanging information about occurrent events. In particular

- Components *publish* notifications about the events they observe, or
- they *subscribe* to the events that they are interested to be notified about.

And the essence here is to capture these events and then process them to cause changes to the application in addition to storing them as an audit log.

Communication is:

- Purely message based
- Asynchronous
- Multicast
- Implicit
- Anonymous

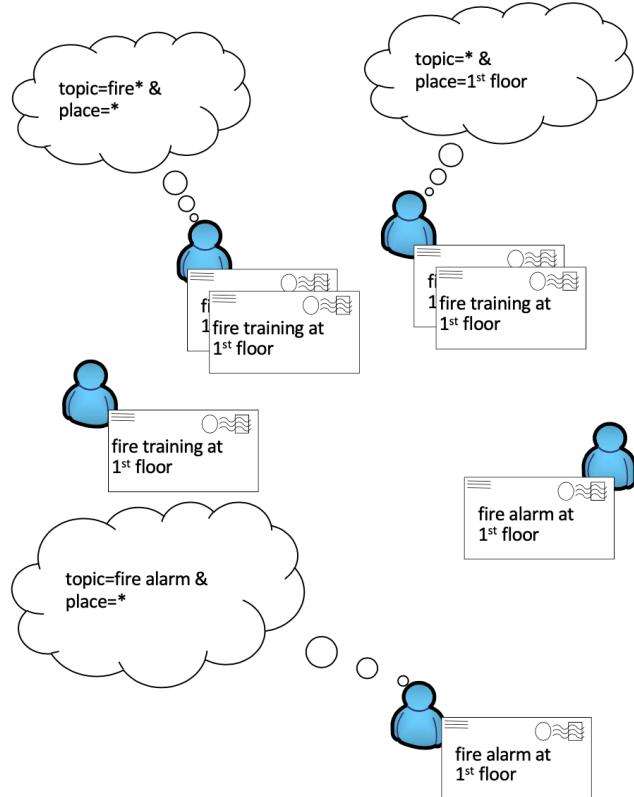


Figure 95: Event-based system. Some components are publishing events and some others are listening for specific events. In this example, the up left component is listening for events that include "fire" happening in any place (* placeholder).

Complex Event Processing (CEP) Also known as event, stream or event stream processing is the use of technology for querying data before storing it within a database or, in some cases, without it ever being stored. Complex event processing is an organizational tool that helps to aggregate a lot of different information and that identifies and analyzes cause-and-effect relationships among events in real time. CEP matches continuously incoming events against a pattern and provides insight into what is happening and allows you to proactively take effective actions.

CEP processes incoming events based on an existing pattern, in a real-time fashion (typical CEP rules search for *sequences of events*). In comparison to Simple Event Processing, CEP systems execute data manipulation on via an algorithm that is pre-stored. The process achieves speed by discarding any irrelevant data in the beginning. As soon as the incoming events are compared to all the stored patterns, the result/response is sent out straight away, giving the process real-time capabilities. CEP is used for highly demanding, continuous-intelligence applications that enhance situational awareness and support real-time decisions. In addition to this speed, CEP systems are also highly scalable and performance-oriented. This allows them to create an insightful response in real-time.

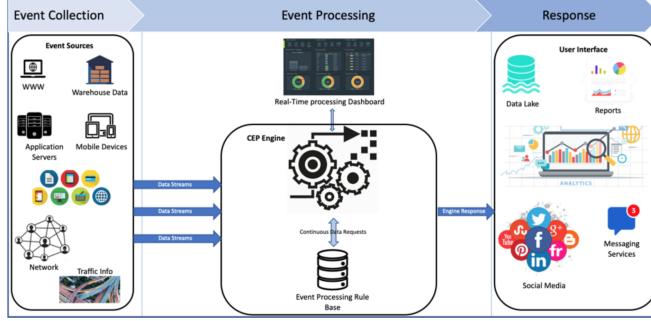


Figure 96: An idea of how a CEP system might look.

Relation	Illustration	Interpretation
$X < Y$	\underline{X}	X takes place before Y
$Y > X$	\underline{Y}	
$X \text{ m } Y$	$\underline{X} \quad \underline{Y}$	X meets Y (i stands for <i>inverse</i>)
$X \text{ o } Y$	$\underline{X} \quad \underline{Y}$	X overlaps with Y
$Y \text{ oi } X$	$\underline{Y} \quad \underline{X}$	
$X \text{ s } Y$	$\underline{X} \quad \underline{Y}$	X starts Y
$Y \text{ si } X$	$\underline{Y} \quad \underline{X}$	
$X \text{ d } Y$	$\underline{X} \quad \underline{Y}$	X during Y
$Y \text{ di } X$	$\underline{Y} \quad \underline{X}$	
$X \text{ f } Y$	$\underline{Y} \quad \underline{X}$	X finishes Y
$Y \text{ fi } X$	$\underline{Y} \quad \underline{X}$	
$X = Y$	$\underline{X} \quad \underline{Y}$	X is equal to Y

Figure 97: CEP semantics, a subset of Allen's semantics

```

TESLA / T-Rex           ACTION
Define Fire(area: string, measuredTemp: double)
From   Smoke(area=$a) and last
        Temp(area=$a and value>45)
        within 5 min. from Smoke
Where  area=Smoke.area and
        measuredTemp=Temp.value

```

CONDITION (PATTERN)

Figure 98: Example of CEP detecting languages by *Cugola, G. and Margara, A., 2010, July. TESLA: a formally defined event specification language.*

7.3.3 Service Oriented Architecture (SOA)

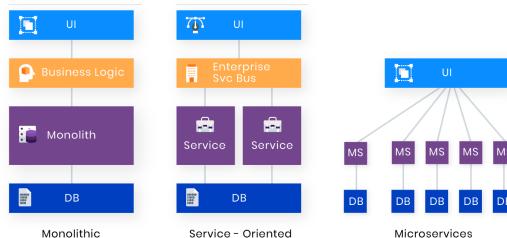


Figure 99: Evolution of software architectures.

Monolithic Architecture Monolith is an ancient word referring to a huge single block of stone. Though this term is used broadly today, the image remains the same across fields. In software engineering, a monolithic pattern refers to a single indivisible unit. The concept of monolithic software lies in different components of an application being combined into a single program on a single platform. Usually, a monolithic app consists of a database, client-side user

interface, and server-side application. All the software's parts are unified and all its functions are managed in one place.

A monolithic architecture is comfortable for small teams to work with, which is why many startups choose this approach when building an app. Components of monolithic software are interconnected and interdependent, which helps the software be self-contained. This architecture is a traditional solution for building applications, but some developers find it outdated. However, we believe that a monolithic architecture is a perfect solution in some circumstances.

Pros:

- **Simpler development and deployment:** there are lots of tools you can integrate to facilitate development. In addition, all actions are performed with one directory, which provides for easier deployment. With a monolithic core, developers don't need to deploy changes or updates separately, as they can do it at once and save lots of time.
- **Fewer cross-cutting concerns:** most applications are reliant on a great deal of cross-cutting concerns, such as audit trails, logging, rate limiting, etc. Monolithic apps incorporate these concerns much easier due to their single code base. It's easier to hook up components to these concerns when everything runs in the same app.
- **Better performance:** if built properly, monolithic apps are usually more performant than microservice-based apps. An app with a microservices architecture might need to make 40 API calls to 40 different microservices to load each screen, for example, which obviously results in slower performance. Monolithic apps, in turn, allow faster communication between software components due to shared code and memory.

Cons:

- **Codebase gets cumbersome over time:** in the course of time, most products develop and increase in scope, and their structure becomes blurred. The code base starts to look really massive and becomes difficult to understand and modify, especially for new developers. It also gets harder to find side effects and dependencies
- **Difficult to adopt new technologies:** if there's a need to add some new technology to your app, developers may face barriers to adoption. Adding new technology means rewriting the whole application, which is costly and time-consuming.
- **Limited agility:** in monolithic apps, every small update requires a full redeployment. Thus, all developers have to wait until it's done. When several teams are working on the same project, agility can be reduced greatly.

Service Oriented Architecture A service-oriented architecture (SOA) is a software architecture style that refers to an application composed of discrete and loosely coupled software agents that perform a required function. SOA has two main roles: a service provider and a service consumer. Both of these roles can be played by a software agent. The concept of SOA lies in the following: an application can be designed and built in a way that its modules are integrated seamlessly and can be easily reused.

Pros:

- **Reusability of services** Due to the self-contained and loosely coupled nature of functional components in service-oriented applications, these components can be reused in multiple applications without influencing other services.
- **Better maintainability** Since each software service is an independent unit, it's easy to update and maintain it without hurting other services. For example, large enterprise apps can be managed easier when broken into services.
- **Higher reliability** Services are easier to debug and test than are huge chunks of code like in the monolithic approach. This, in turn, makes SOA-based products more reliable.
- **Parallel development** As a service-oriented architecture consists of layers, it advocates parallelism in the development process. Independent services can be developed in parallel and completed at the same time. Below, you can see how SOA app development is executed by several developers in parallel:

Cons:

- **Complex management** The main drawback of a service-oriented architecture is its complexity. Each service has to ensure that messages are delivered in time. The number of these messages can be over a million at a time, making it a big challenge to manage all services.
- **High investment costs** SOA development requires a great upfront investment of human resources, technology, and development.
- **Extra overload** In SOA, all inputs are validated before one service interacts with another service. When using multiple services, this increases response time and decreases overall performance.

Microservice Architecture Microservice is a type of service-oriented software architecture that focuses on building a series of autonomous components that make up an app. Unlike monolithic apps built as a single indivisible unit, microservice apps consist of multiple independent components that are glued together with APIs.

The microservices approach focuses mainly on business priorities and capabilities, whereas the monolithic approach is organized around technology layers, UIs, and databases. The microservices approach has become a trend in recent years as more and more enterprises become agile and move toward DevOps.

There are lots of examples of companies that have evolved from a monolithic approach to microservices. Among the most prominent are Netflix, Amazon, Twitter, eBay, and PayPal. In order to determine whether microservices are suitable for your project, let's define the pros and cons of this approach. **Pros:**

- **Easy to develop, test, and deploy** The biggest advantage of microservices over other architectures is that small single services can be built, tested, and deployed independently. Since a deployment unit is small, it facilitates and speeds up development and release. Besides, the release of one unit isn't limited by the release of another unit that isn't finished. And the last plus here is that the risks of deployment are reduced as developers deploy parts of the software, not the whole app.
- **Increased agility** With microservices, several teams can work on their services independently and quickly. Each individual part of an application can be built independently due to the decoupling of microservice components. For example, you may have a team of 100 people working on the whole app (like in the monolithic approach), or you can have 10 teams of 10 people developing different services for the app. Increased agility allows developers to update system components without bringing down the application. Moreover, agility provides a safer deployment process and improved uptime. New features can be added as needed without waiting for the entire app to launch.
- **Ability to scale horizontally** Vertical scaling (running the same software but on bigger machines) can be limited by the capacity of each service. But horizontal scaling (creating more services in the same pool) isn't limited and can run dynamically with microservices. Furthermore, horizontal scaling can be completely automated.

Cons:

- **Complexity** The biggest disadvantage of microservices lies in their complexity. Splitting an application into independent microservices entails more artifacts to manage. This type of architecture requires careful planning, enormous effort, team resources, and skills. The reasons for high complexity are the following:
 - Increased demand for automation, as every service should be tested and monitored
 - Available tools don't work with service dependencies
 - Data consistency and transaction management becomes harder as each service has a database

- **Security concerns** In a microservices application, each functionality that communicates externally via an API increases the chance of attacks. These attacks can happen only if proper security measurements aren't implemented when building an app.
- **Different programming languages** The ability to choose different programming languages is two sides of the same coin. Using different languages make deployment more difficult. In addition, it's harder to switch programmers between development phases when each service is written in a different language.

Event-Driven Architecture (EDA) An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website. Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

Event-driven architectures have three key components: event producers, event routers, and event consumers. A producer publishes an event to the router, which filters and pushes the events to consumers. Producer services and consumer services are decoupled, which allows them to be scaled, updated, and deployed independently. **Pros:**

- **Scale and fail independently** By decoupling your services, they are only aware of the event router, not each other. This means that your services are interoperable, but if one service has a failure, the rest will keep running. The event router acts as an elastic buffer that will accommodate surges in workloads.
- **Audit with ease** An event router acts as a centralized location to audit your application and define policies. These policies can restrict who can publish and subscribe to a router and control which users and resources have permission to access your data. You can also encrypt your events both in transit and at rest.
- **Develop with agility** You no longer need to write custom code to poll, filter, and route events; the event router will automatically filter and push events to consumers. The router also removes the need for heavy coordination between producer and consumer services, speeding up your development process.
- **Cut costs** Event-driven architectures are push-based, so everything happens on-demand as the event presents itself in the router. This way, you're not paying for continuous polling to check for an event. This means less network bandwidth consumption, less CPU utilization, less idle fleet capacity, and less SSL/TLS handshakes.

Cons:

- **Over-engineering of processes** Sometimes a simple call from one service to another is enough. If a process uses event driven architecture, it usually requires much more infrastructure to support it, which will add costs (as it will need a queueing system)
- **Inconsistencies** Because processes now rely on eventual consistency, it is not typical to support ACID (atomicity, consistency, isolation, durability) transactions, so handling of duplications, or out of sequence events can make service code more complicated, and harder to test and debug all situations.



Figure 100: How it works an event-driven architecture

8 EPL

8.1 EPL and Esper

The **Event Processing Language (EPL)** is a declarative language for dealing with high frequency time-based event data. It is grounded on DSMS approach:

- Windowing
- Relational select, join, aggregate
- Relation-to-stream operators to produce output
- Sub-queries

It implements and extends the SQL-standard and enables rich expressions over events and time. Furthermore, it also include complex event recognition abstraction, in particular for pattern detection.

It is offered by [Esper](#).

The Esper compiler compiles EPL into byte code that can be saved in jar package file format for distribution and execution. The Esper runtime loads and executes byte code produced by the Esper compiler. The runtime provides a **highly scalable, memory-efficient, in-memory computing, minimal latency, real-time streaming-capable** processing engine for online and real-time arriving data and high-variety data, as well as for historical event analysis. It has several adapters for input/output: CSV, Java Message System in/out, API, DB, Socket and HTTP. In conclusion, Esper focuses on High Availability, ensuring that the state is recoverable in the case of failure.

8.2 Processing Model

It is built on four abstractions:

- **Sources**
 - Produce data items from sensors, trace files, etc.
- **Registered EPL queries**
 - Continuously executed against the data items produced by the sources
- **Listeners**
 - Receive data items from queries
 - Push data items to other queries
- **Subscribers**
 - Receive processed data tuples

The EPL processing model is continuous: Listeners to statements receive updated data as soon as the engine processes events for that statement, according to the statement's choice of event streams, retain clause restrictions, filters and output rates.

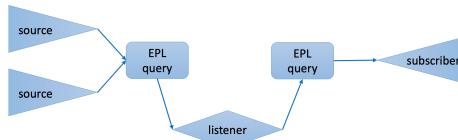


Figure 101: EPL processing model

Running example: Count the number of fires detected using a set of smoke and temperature sensors in the last 10 minutes.

Events:

- Smoke event: String sensor, boolean state
- Temperature event: String sensor, double temperature
- Fire event: String sensor, boolean smoke, double temperature

Condition:

- Fire: at the same sensor *smoke* followed by *temperature > 50*

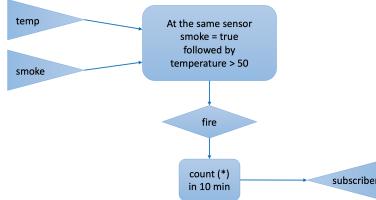


Figure 102: EPL processing model - Fire example

8.3 Event types and Query syntax

Two ways to declare events:

- EPL *create schema clause*
- Runtime configuration API *addEventType*

```

create schema
schema_name [as]
(property_name property_type
[,property_name property_type [,....])
[inherits inherited_event_type
[, inherited_event_type] [,....])

```

Figure 103: EPL - Declare event types

```

[insert into insert_into_def]
select select_list
from stream_def [as name]
[, stream_def [as name]] [,....]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]

```

Figure 104: EPL - Query syntax

EPL is similar to SQL... it uses *selects*, *where*, ...

Event streams and views instead of tables:

- Views define the data available for the query
- Views can represent windows over streams
- Views can also sort events, derive statistics from event attributes, group events, ...

```

create schema SmokeSensorEvent(
    sensor string,
    smoke boolean
);

create schema TemperatureSensorEvent(
    sensor string,
    temperature double
);

create schema FireComplexEvent(
    sensor string,
    smoke boolean,
    temperature double
);

```

<pre> select * from TemperatureSensorEvent where temperature>50 </pre>	<pre> select avg(temperature) from TemperatureSensorEvent </pre>
---	---

Figure 106: EPL - Query syntax example

Figure 105: EPL - Declare event types example

Type	Syntax	Description
Logical Sliding	win:time(<i>time_period</i>)	Sliding window that covers the specified time interval into the past
Logical Tumbling	win:time_batch(<i>time_period</i> [, <i>reference point</i>] [, <i>flow control</i>])	Tumbling window that batches events and releases them every specified time interval, with flow control options
Physical Sliding	win:length(<i>size</i>)	Sliding window that covers the specified number of elements into the past
Physical Tumbling	win:length_batch(<i>size</i>)	Tumbling window that batches events and releases them when a given minimum number of events has been collected

Figure 107: EPL offers 4 types of windows that can be used to perform more customized queries.

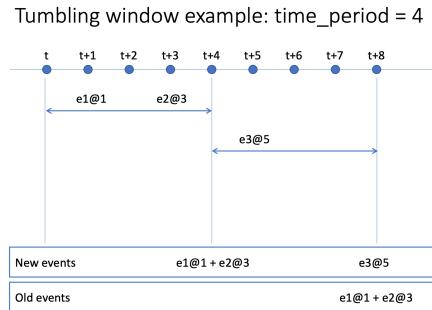
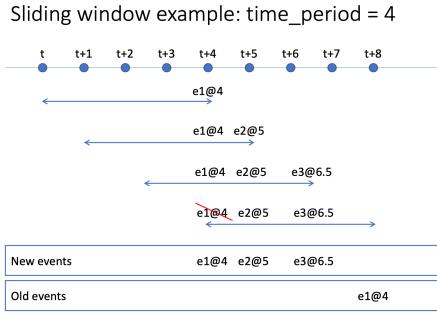


Figure 108: Window opened 4 seconds in the past. At t+8 e1@4 is discarded since it is more than 4 seconds in the past.

Figure 109: Each window is discarded after 4 seconds, and a new one is created.

Physical sliding window example: size = 3

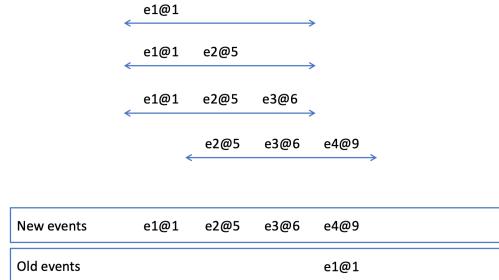


Figure 110: The window counts the number of elements seen so far, allowing to store only the last 3 of them. That's why e1@1 is discarded when e4@9 is seen.

Output clause The output clause is optional in Esper. It is used to control the output rate and suppress output events.

```

output  [all | first | last | snapshot]
every   output_rate [seconds | events]           select      avg(temperature)
                                                    from        TemperatureSensorEvent.win:length(4)
                                                    output      snapshot every 2 events

                                                    select      avg(temperature)
                                                    from        TemperatureSensorEvent.win:time(4 sec)
                                                    output      snapshot every 2 sec

```

Figure 111: Output can be used to control the advancement of sliding windows.

8.4 Pattern Matching

An event pattern emits when one or more event occurrences match the pattern definition, which can include:

- Constraints on the content of events
- Constraints on the time of occurrence
- Conditions for pattern creation

Content-based event selection:

- $\text{TempStream}(\text{sensor} = "S0", \text{val} > 50)$

Time-based event observers specify time intervals or time schedules:

- $\text{timer : interval}(10\text{seconds})$ fires after 10 seconds
- $\text{timer : at}(5, *, *, *, *)$ every 5 minutes (ubuntu cron syntax)

Pattern matching operators

- Logical operators: *and, or, not*
- Temporal operators that operate on event order: \rightarrow (*followed-by*)
- Creation/termination control: *every, every-distinct, [num] and until*
- Guards filter out events and cause termination: *timer:within, timer:withinmax and while-expression*

```
select a.sensor from pattern
[every (
    a = SmokeEvent(smoke=true)
    ->
    TempEvent(val>50, sensor=a.sensor)
    where timer:within(2 sec)
)]
```

Figure 112: EPL - Pattern Matching example. The result returns the name of sensors that match the specified pattern: whenever a sensor sees some smoke and after that the temperature becomes greater than 50, within an interval of at most 2 seconds.

Every expression

- When *expr* evaluates true or false the pattern matching should-restart.
- Without the **every** operator the pattern matching process does not re-start.

Examples:

- *A* - This pattern fires when encountering an A event and then stops
- *every A* - This pattern keeps firing when encountering A events, and does not stop

A1 B1 B2 A2 A3 B3 A4 B4

Figure 113: Example sequence of events.

- *every* ($A \rightarrow B$) - Detect an event A followed by an event B: at the time when B occurs, the pattern matches and restarts looking for the next A event.

Matches:

{A1, B1}, {A2, B3}, {A4, B4}

- *every A → B* - The pattern fires for every A followed by a B event.

Matches:

{A1, B1}, {A2, B3}, {A3, B3}, {A4, B4}

- *A → every B* - The pattern fires for an A event followed by every B event.

Matches:

{A1, B1}, {A1, B2}, {A1, B3}, {A1, B4}

- *every A → every B* - The pattern fires for every A event followed by every B event.

Matches:

{A1, B1}, {A1, B2}, {A1, B3}, {A1, B4}, {A2, B3}, {A2, B4}, {A3, B3}, {A3, B4}, {A4, B4}

With the *every* operator multiple (partial) instances of the same pattern can be active at the same time. Each instance can consume some resources when events enter the engine. For this reason is good practice to end pending instance whenever possible with:

- the *timer:within* construct
- the *and not* construct

	A1	A2	B1
Pattern	Results		
every A → B	{A1, B1}, {A2, B1}		
every A → (B and not A)	{A2, B1}		

Figure 114: The **and not** operator causes the sub-expression looking for A1, B? to end when A2 arrives.

	A1@1	A2@3	B1@4
Pattern	Results		
every A → B	{A1, B1}, {A2, B1}		
every A → (B where timer:within(2 sec))	{A2, B1}		

Figure 115: The **timer:within** operator causes the sub-expression looking for A1, B? to end after 2 seconds

9 Kafka

9.1 Kafka Basics

9.1.1 Kafka in a nutshell

Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

- **Servers:** Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.
- **Clients:** they allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala including the higher-level Kafka Streams library, for Go, Python, C/C++, and many other programming languages as well as REST APIs.

9.1.2 Main Concepts and Terminology

In Kafka, an **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of \$200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Producers are those client applications that publish (write) events to Kafka, and **Consumers** are those that subscribe to (read and process) these events. Multiple consumers can be combined into a Consumer Group, which provide scaling capabilities. In a Consumer Group each consumer is assigned a subset of partitions for consumption. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

Events are organized and durably stored in **topics**. A Topic is a category/feed name to which records are stored and published. An example topic name could be "payments". Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Topics are partitioned, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. Indeed, partitions allow topics to be parallelized by splitting the data into a particular topic across multiple brokers. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

There are two policies that can be used when assigning partitions:

- **Sticky-Assignment ON:** when adding a new consumer to the group, nothing happens unless the system understand that is useful to assign one of the partitions to that consumer (e.g., if we have 4 partitions and 3 consumers, when adding a 4th consumer, the system assigns a partition to it, so that there are exactly one partition per consumer)
- **Sticky-Assignment OFF:** when adding a new consumer to the group, it is assigned with a partition, even if it means detaching one partition from one of the other consumers (some consumers could remain without any partitions assigned)

Note that partitions can not be shared by consumers in the same consumer group.

A producer must know which partition to write to, this is not up to the broker. The simplest way is to use a Round-Robin strategy, in which the producer writes on each partition at turn.

Another option is to use an hashing function over the event key to calculate the partition in which the record should go to. In this way, all records with the same key will arrive at the same partition. Before a producer can send any records, it has to request metadata about the cluster from the broker. The metadata contains information on which broker is the leader for each partition and a producer always writes to the partition leader.

A common error when publishing records is setting the same key or null key for all records, which results in all records ending up in the same partition and you get an unbalanced topic.

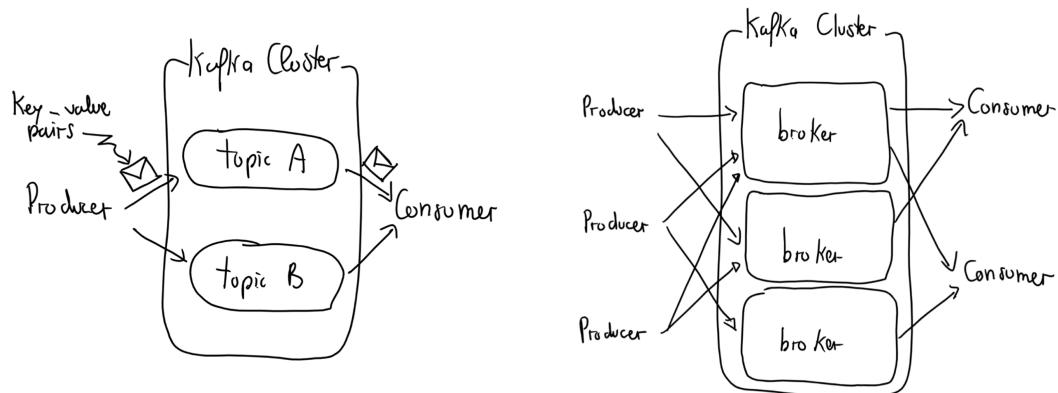


Figure 116: Kafka producers and consumers exchange messages through topics which are included within brokers.

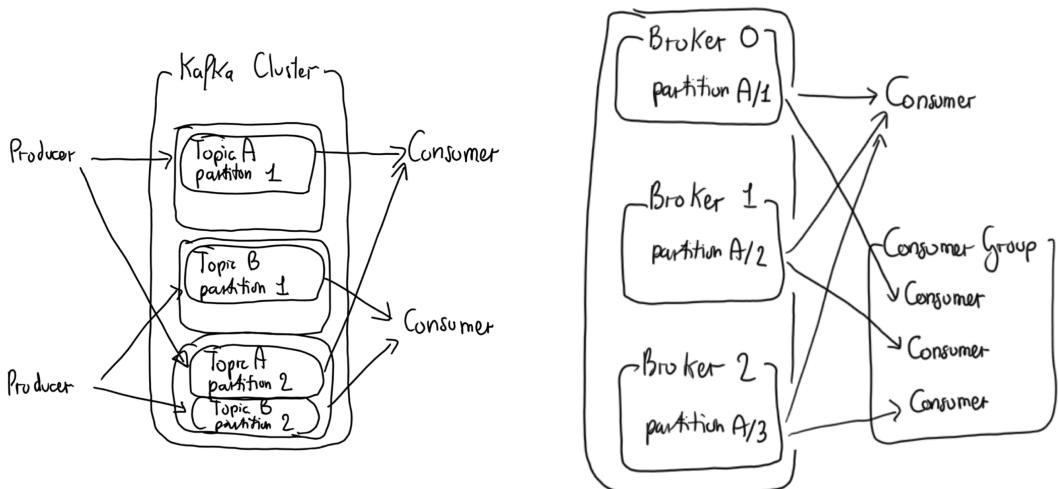


Figure 117: Using partitions, each broker can contain different topics and different partitions in order to improve availability (by replicating data) and scalability (by parallelizing message reading/writing). Moreover, consumer groups allow to scale the consumption process.

9.1.3 Kafka Internals

In Kafka, each partition is stored on the Broker's disk as one or more log files. Each message in the log is identified by its offset number. The write operations on the log files are performed in append mode, in this way the performance are not influenced by the file size (which may grow a lot!).

Relevant notes:

- Consumers can consume from different offset
- Brokers are single threaded to guarantee consistency

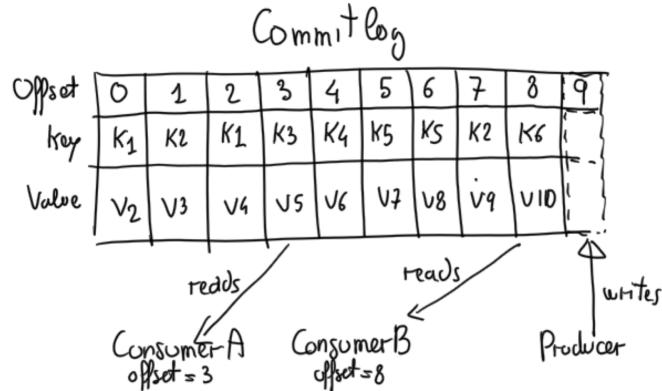


Figure 118: Consumer A after reading on offset 3, will read on offset 4. Consumer B can't read anything more, since 8 is the last offset that was written.

By default Kafka messages will be retained for seven days. By the way, the log retention is configurable per Broker by setting:

- a time period
- a size limit

When cleaning up a log there are two policies:

- the **default policy** consists in deleting the oldest messages
- an alternate policy is **log compaction**

Before compaction									After compaction						
Offset	0	1	2	3	4	5	6	7	8	K ₁	K ₃	K ₄	K ₅	K ₂	K ₆
key	K ₁	K ₂	K ₁	K ₃	K ₄	K ₅	K ₅	K ₂	K ₆						
Value	V ₂	V ₃	V ₄	V ₅	V ₆	V ₇	V ₈	V ₉	V ₁₀	V ₄	V ₅	V ₆	V ₈	V ₉	V ₁₀

Figure 119: A compacted log retains at least the last known message value for each key within the partition.

In Kafka, **replication is implemented at the partition level**. The redundant unit of a topic partition is called a replica. Each partition usually has one or more replicas meaning that partitions contain messages that are replicated over a few Kafka brokers in the cluster.

Every partition (replica) has one server acting as a leader and the rest of them as followers. The leader replica handles all read-write requests for the specific partition and the followers replicate the leader. If the lead server fails, one of the follower servers becomes the leader by default. You should strive to have a good balance of leaders so each broker is a leader of an equal amount of partitions to distribute the load.

When a producer publishes a record to a topic, it is published to its leader. The leader appends the record to its commit log and increments its record offset. Kafka only exposes a record to a consumer after it has been committed and each piece of data that comes in will be stacked on the cluster.

Producers can control durability by requiring the leader a number of acknowledgments before considering the request complete:

- **acks=0**: producer will not wait for any acknowledgment from the broker
- **acks=1**: producer will wait until the leader has written the record to its local log (equal to eventual consistency - at least once semantics)
- **acks=all**: producer will wait until all all insync replicas have acknowledged receipt of the record

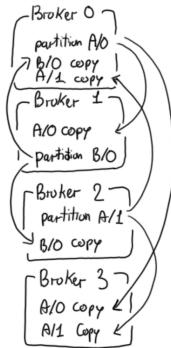


Figure 120: Fault tolerance via a Replicated Log

9.2 Avro and Schema Registry

[Apache Avro](#) is a binary serialization format. It relies on schemas (defined in JSON format) that define what fields are present and their type. Nested fields are supported as well as arrays.

In Avro, data is defined with a self-describing schema allowing for:

- code generation for serializers and de-serializers in multiple languages
- type checking at write time

```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "favorite_number", "type": ["int", "null"]},
    {"name": "favorite_color", "type": ["string", "null"]}
  ]
}
```

Figure 121: Avro schema example

Avro supports schema evolutivity: you can have multiple versions of your schema, by adding or removing fields. A little care needs to be taken to indicate fields as optional to ensure backward or forward compatibility. Since Avro converts data into arrays of bytes, and that Kafka messages also contain binary data, we can ship Avro messages with Kafka. The real question is: where to store the schema?

The [Schema Registry](#) is the answer to this problem: it is a server that runs in your infrastructure (close to your Kafka brokers) and that stores your schemas (including all their versions). When you send Avro messages to Kafka, the messages contain an identifier of a schema stored in the Schema Registry.

A library allows you to serialize and deserialize Avro messages, and to interact transparently with the Schema Registry:

- When sending a message, the serializer will make sure the schema is registered, get its ID, or register a new version of the schema for you.
- When reading a message, the deserializer will find the ID of the schema in the message, and fetch the schema from the Schema Registry to deserialize the Avro data.

Both the Schema Registry and the library are under the Confluent umbrella: open source but not part of the Apache project. This means you will want to use the Confluent distribution to use the Schema Registry, not the Apache distribution.

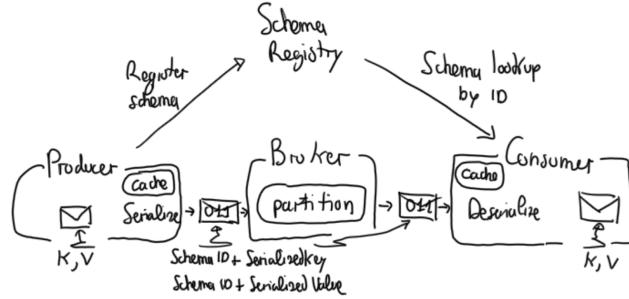


Figure 122: **Schema registry at work.** The producer, after having registered a schema, encodes messages containing a concatenation of the SchemaID and the content (key and value). Then, the consumer extracts the SchemaID, lookups the schema on the registry and decodes the message's content.

9.2.1 Schema Evolution

As said, Avro supports schema evolutivity. In particular we define two types of schema compatibility:

- **Backward compatibility:**
 - Code with a new version of the schema can read data written in the old schema
 - Code that reads data written with the schema will assume default values if fields are not provided
- **Forward compatibility:**
 - Code with previous versions of the schema can read data written in the new schema
 - Code that reads data written with the schema ignores new fields

Thus, we have a **full compatibility** (forward and backward).

```

Consider a schema written with the following fields
{
  "name": "suit", "type": "string",
  "name": "card", "type": "string"
}
Backward compatibility: Consumer is expecting the following schema
and assumes default for omitted size field
{
  "name": "suit", "type": "string",
  "name": "card", "type": "string",
  "name": "size", "type": "string", "default": ""
}
Forward compatibility: Consumer is expecting the following schema
and ignores additional card field
{
  "name": "suit", "type": "string"
}

```

Figure 123: Compatibility examples

9.3 Connect for Data Movement

9.3.1 Kafka Connect

Kafka Connect, an open source component of Apache Kafka®, is a framework for connecting Kafka with external systems such as databases, key-value stores, search indexes, and file systems.

Kafka Connect is focused on streaming data to and from Kafka, making it simpler for you to write high quality, reliable, and high performance connector plugins. It also enables the framework to make guarantees that are difficult to achieve using other frameworks. Kafka Connect is an integral component of an ETL pipeline, when combined with Kafka and a stream processing framework.

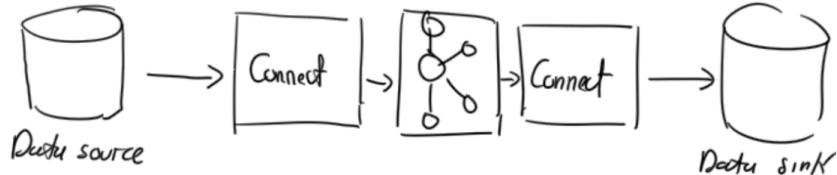


Figure 124: Kafka Connect: simple, scalable and reliable.

The main actors in Kafka Connect are:

- **Source Connector:** basically a Kafka Producer client that reads data from an external data system into Kafka. It ingest entire databases and streams table updates to Kafka topics. It can also collect metrics from all of your application servers and store these in Kafka topics, making the data available for stream processing with low latency.
- **Sink Connectors:** a Kafka Consumer client that writes data to an external data system. It delivers data from Kafka topics into secondary indexes such as Elasticsearch, or batch systems such as Hadoop for offline analysis.

In order to efficiently discuss the inner workings of Kafka Connect, it is helpful to establish a few [major concepts](#).

- **Connectors** – the high level abstraction that coordinates data streaming by managing tasks
- **Tasks** – the implementation of how data is copied to or from Kafka
- **Workers** – the running processes that execute connectors and tasks
- **Converters** – the code used to translate data between Connect and the system sending or receiving data
- **Transforms** – simple logic to alter each message produced by or sent to a connector
- **Dead Letter Queue** – how Connect handles connector errors

Connectors:

Connectors in Kafka Connect define where data should be copied to and from. A connector instance is a logical job that is responsible for managing the copying of data between Kafka and another system. All of the classes that implement or are used by a connector are defined in a connector plugin.

There are many [existing connectors](#) that fits several popular technologies. However, it is possible to write a new connector plugin from scratch.

Tasks:

Tasks are the main actor in the data model for Connect. Each connector instance coordinates a set of tasks that actually copy the data. By allowing the connector to break a single job into many tasks, Kafka Connect provides built-in support for parallelism and scalable data copying

with very little configuration. These tasks have no state stored within them. As such, tasks may be started, stopped, or restarted at any time in order to provide a resilient, scalable data pipeline.

When a connector is first submitted to the cluster, the workers rebalance the full set of connectors in the cluster and their tasks so that each worker has approximately the same amount of work. This same rebalancing procedure is also used when connectors increase or decrease the number of tasks they require, or when a connector's configuration is changed. When a worker fails, tasks are rebalanced across the active workers. When a task fails, no rebalance is triggered as a task failure is considered an exceptional case. As such, failed tasks are not automatically restarted by the framework and should be restarted via the REST API.

Workers:

Connectors and tasks are logical units of work and must be scheduled to execute in a process. Kafka Connect calls these processes workers and has two types of workers: standalone and distributed.

- **Standalone Workers:** the simplest mode, where a single process is responsible for executing all connectors and tasks. Since it is a single process, it requires minimal configuration. Standalone mode is convenient for getting started, during development, and in certain situations where only one process makes sense, such as collecting logs from a host. However, because there is only a single process, it also has more limited functionality: scalability is limited to the single process and there is no fault tolerance beyond any monitoring you add to the single process.
- **Distributed Workers:** this mode provides scalability and automatic fault tolerance for Kafka Connect. In distributed mode, you start many worker processes using the same `group.id` and they automatically coordinate to schedule execution of connectors and tasks across all available workers. If you add a worker, shut down a worker, or a worker fails unexpectedly, the rest of the workers detect this and automatically coordinate to redistribute connectors and tasks across the updated set of available workers. Note the similarity to consumer group rebalance. Under the covers, connect workers are using consumer groups to coordinate and rebalance.

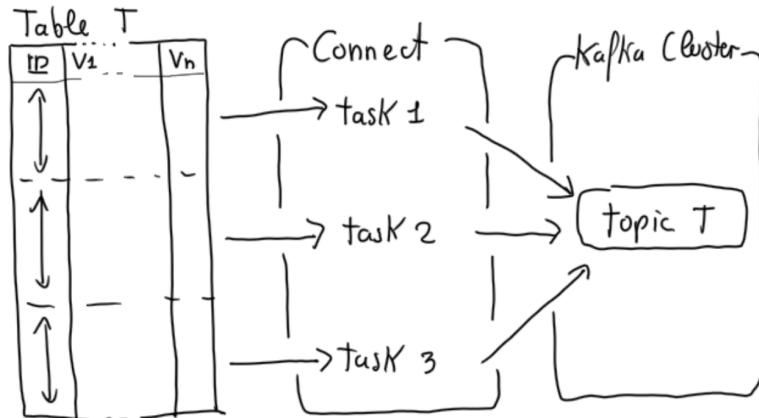


Figure 125: Parallelism and scalability is achieved splitting the workload into pieces run by a worker as different task, each in a separated thread.

Converters:

Converters are necessary to have a Kafka Connect deployment support a particular data format when writing to or reading from Kafka. Tasks use converters to change the format of data from bytes to a Connect internal data format and vice versa.

Converters are decoupled from connectors themselves to allow for reuse of converters between connectors naturally. For example, using the same Avro converter, the JDBC Source Connector can write Avro data to Kafka and the HDFS Sink Connector can read Avro data from Kafka.

This means the same converter can be used even though, for example, the JDBC source returns a ResultSet that is eventually written to HDFS as a parquet file.

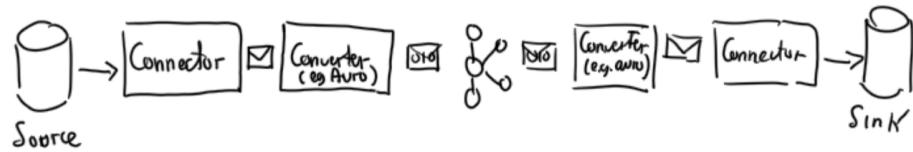


Figure 126: How converters are used when reading from a database using a Source Connector (e.g., MongoDB), writing to Kafka, and finally, writing on a Sink Connector (e.g., MySQL).

9.4 Kafka Stream Processing

References

- Emanuele Della Valle and Marco Brambilla's course slides
- Neo4J - Graph Databases, free ebook
- MongoDB Key-Value database article
- Understanding Key-Value Databases
- Redis command reference
- Columnar databases
- Row-based vs Columnar databases
- Columnar storage documentation on Amazon
- Cassandra gossip protocol
- Cassandra data replication
- Why document databases?
- MongoDB and CAP Theorem
- Big Data Architecture
- Event Driven Systems
- Complex Event Processing (CEP)
- Complex Event Processing (CEP) [2]
- Monolith, Service Oriented and Microservices architecture
- Event-Driven Architecture
- <https://kafka.apache.org/intro>
- <https://aseigneurin.github.io/2018/08/02/kafka-tutorial-4-avro-and-schema-registry.html>