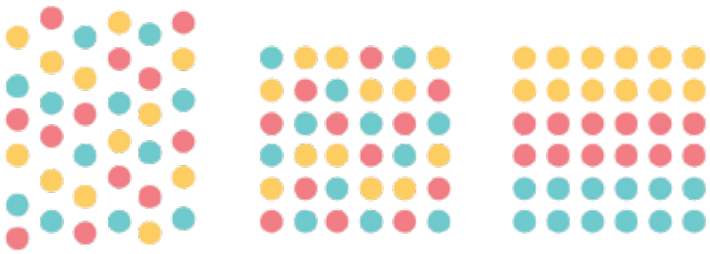
USDE - Notes

Unstructured Streaming and Data Engineering

Professors: Emanuele Della Valle - Marco Brambilla

Author: Simone Staffa



November 21, 2020

Contents

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

- **Hi**ghest **P**aid **P**erson **O**pinion (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

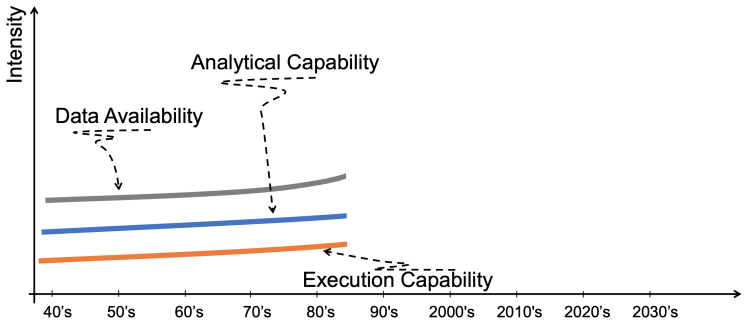


Figure 1: Up until '90s the data available was growing together with our analytical and execution capabilities.

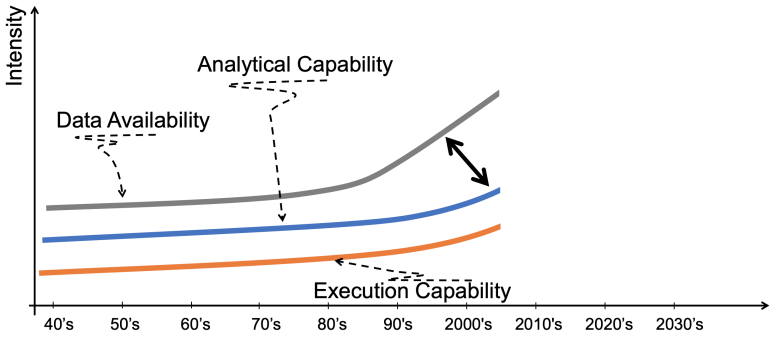


Figure 2: With the new millenium we have the appearence of Big Data. Data availability is growing fast and the digital revolution gap is growing.

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).

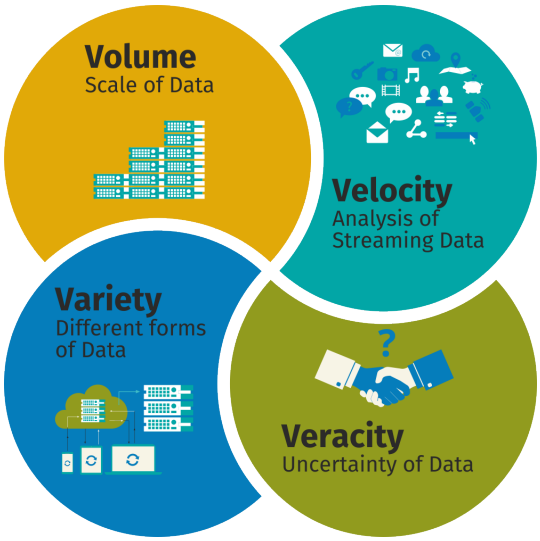


Figure 3: With the new millenium we have the appearence of Big Data. Data availability is growing fast and the digital revolution gap is growing.

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

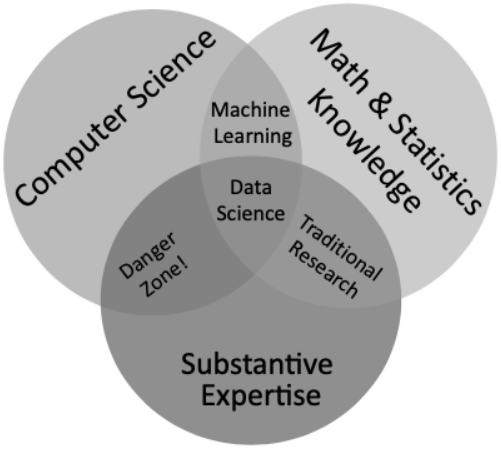
- Extract
- Transport in mega-tankers
- Ship through pipelines
- Store in massive silos

1.2.2 What's Data Science

The Science (and Art) of:

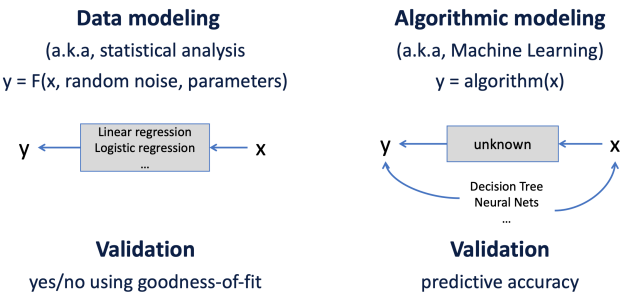
- **Discovering** what we don't know from data
- Obtaining **predictive, actionable insight** from data
- **Creating Data Products** that have business impact now
- **Communicating** relevant business stories from data
- **Building confidence** in decisions that drive business value

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.

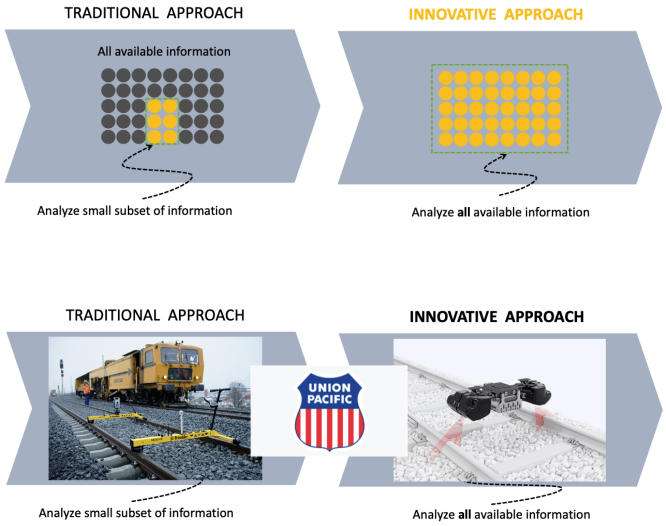


Figure 4: Instead of looking for the perfect exact data, measure everything and **leverage more of the data being captured.** With a large enough dataset at some point we reach the same result (Central Limit Theorem).

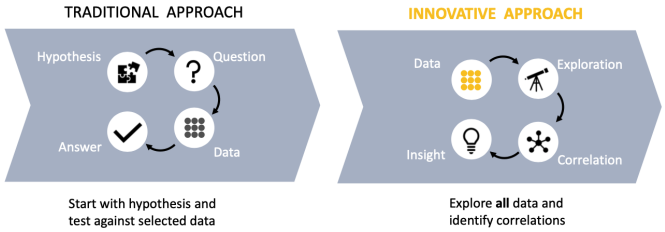


Figure 5: **Data-driven exploration looking for correlation.** For instance, your butcher sells both pure meat and semi-prepared dishes because he knows that if you see the variety of products that he prepares and sells, you will probably notice something that you like!

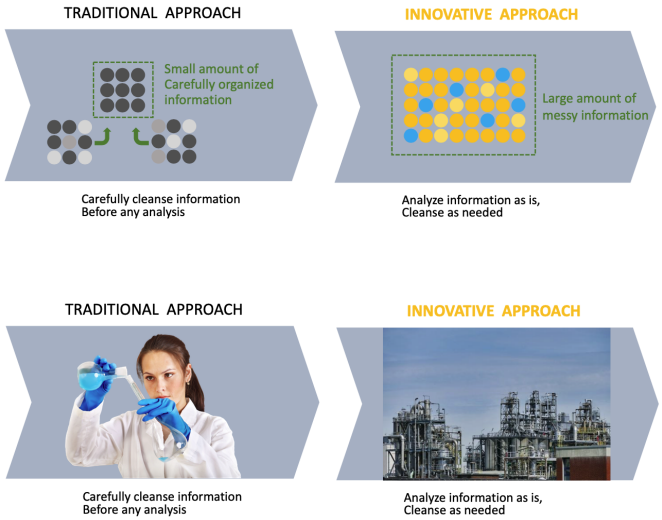


Figure 6: **Reduce effort required to leverage data.** If you can do it by hand it is not said that you can do it automatically. Is hard to do things at scale.

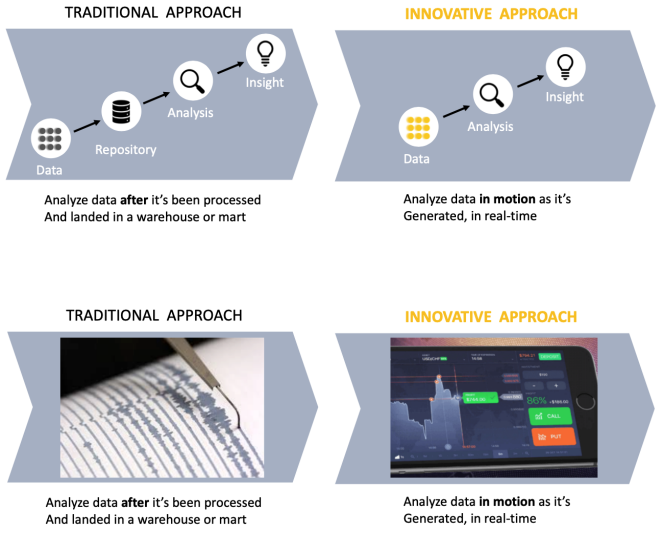
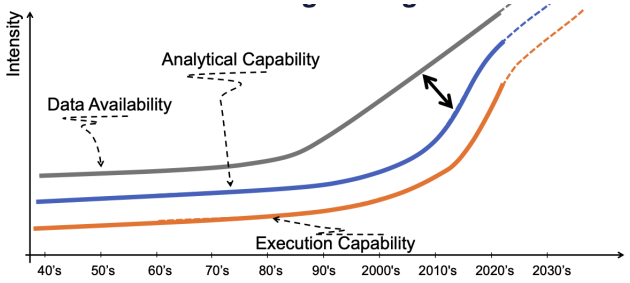


Figure 7: **Leverage data as it is captured.**

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

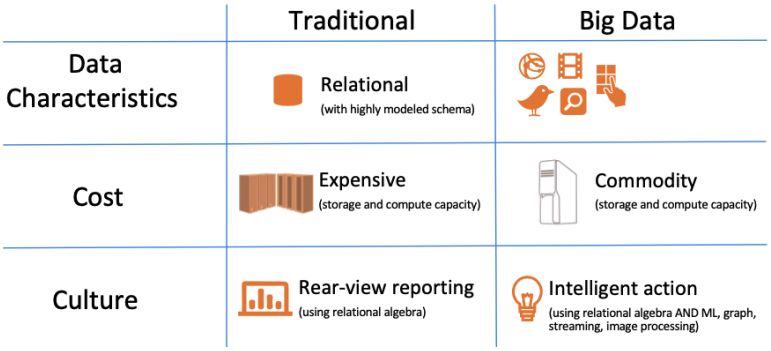


Figure 8: Big Data vs. Traditional Data

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**.

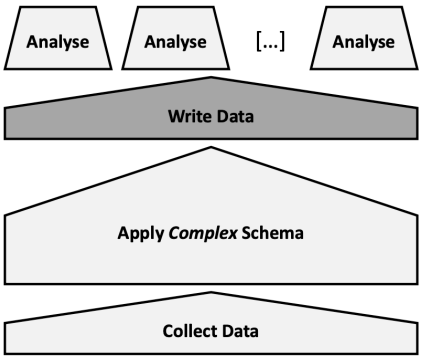


Figure 9: **Schema-on-write**: the rigid and traditional strategy (relational data) in which a complex schema is applied after a long lasting discussing. Here we collect the data from different sources, ensuring that it is compatible with our schema and then we make analysis on that.

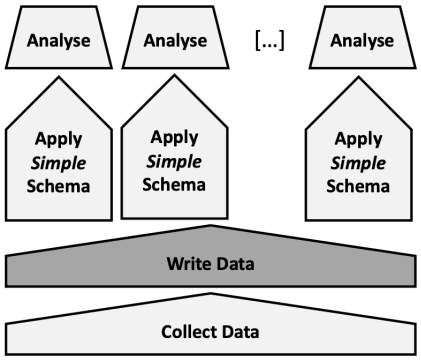


Figure 10: **Schema-on-read**: schema-less approach (document-based data) in which we collect and load data first and ask questions/queries later. All data are kept and the minimal schema for an analysis is applied only when needed. New analysis can then be introduced in any point in time.

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

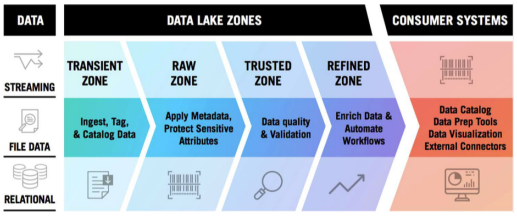


Figure 12: Data Lake in process

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.

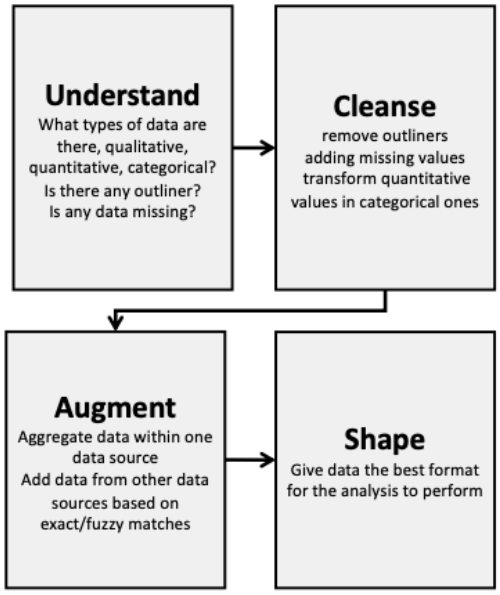


Figure 13: Data Wrangling

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.**

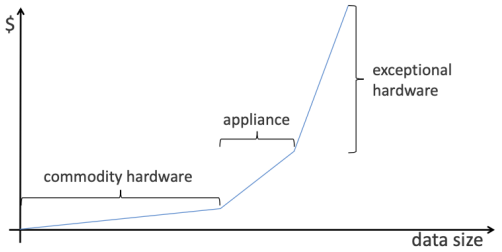


Figure 14: Vertical Scalability

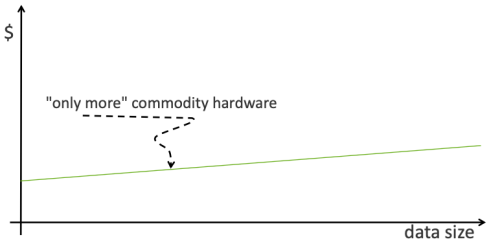


Figure 15: Horizontal Scalability

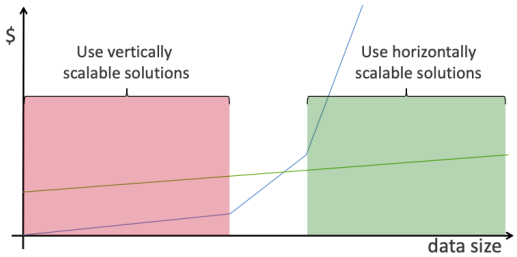


Figure 16: Vertical (Exponential) vs Horizontal (Linear) growth

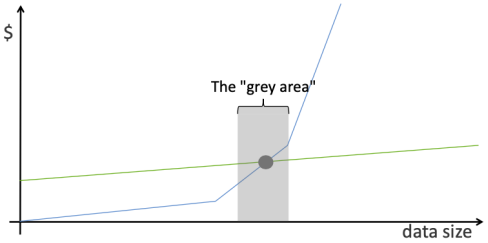


Figure 17: The space within vertical (blue) and horizontal (red) scalable solution. On the left we see an high price gap between the blue and the red line for which the preferred solution is the vertical scalability. On the right is the contrary: the price is growing faster on the blue line while the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

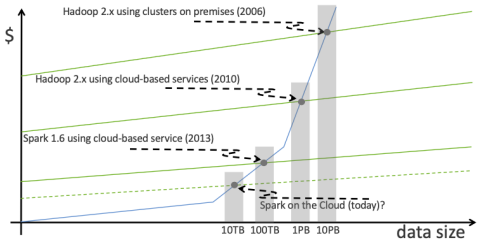


Figure 18: The grey area moves in time. As a result, we can see that horizontal scalability now makes sense when we have to deal with 10TB of data.

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.

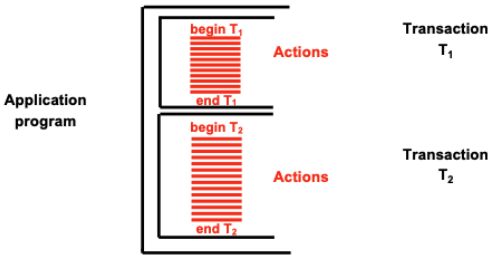


Figure 19: Application and Transactions

```
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

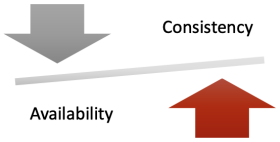
It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- **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)

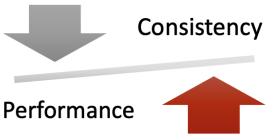
A distributed system can satisfy any two of these guarantees at the same time but not all three. In a distributed system, a network (of networks) is inevitable (by definition). We can't avoid to deal with partition tolerance, we need to cover that. Indeed. Failures can, and will, occur to a networked system. Then, the only option left is choosing between **C**onsistency and **A**vailability. 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.



(a) **Consistency vs Availability**



(b) **Consistency vs Performance**

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.

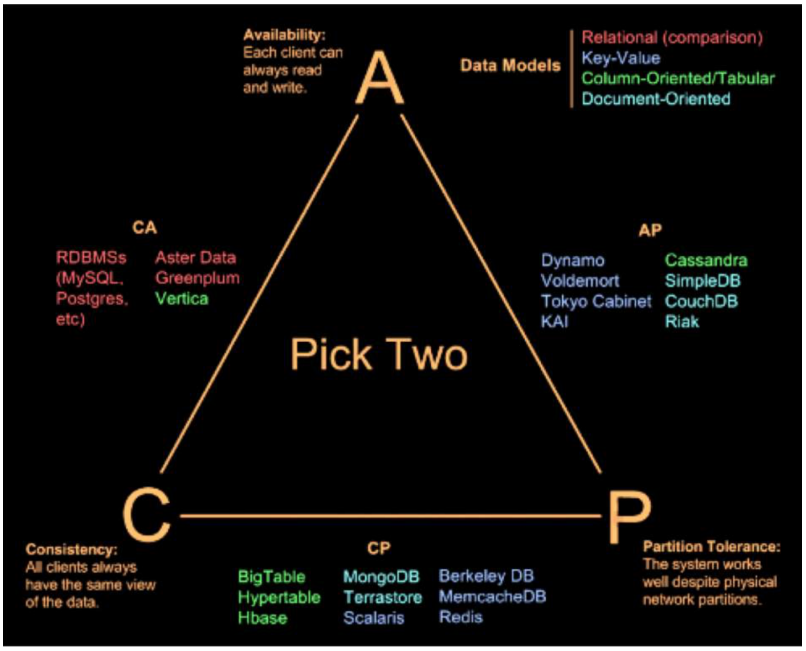


Figure 23: Visual Guide to CAP Theorem

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.

- **B**asic **A**vailability: fulfill request, even in partial consistency. Basic functionalities are always provided.
- **S**oft State: abandon the consistency requirements of the ACID model pretty much completely.
- **E**ventual 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

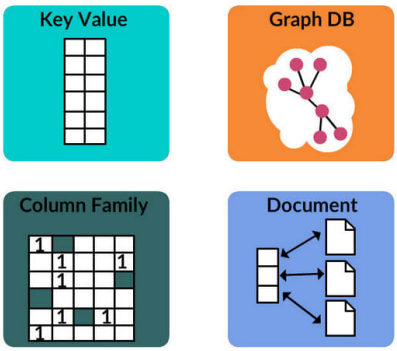


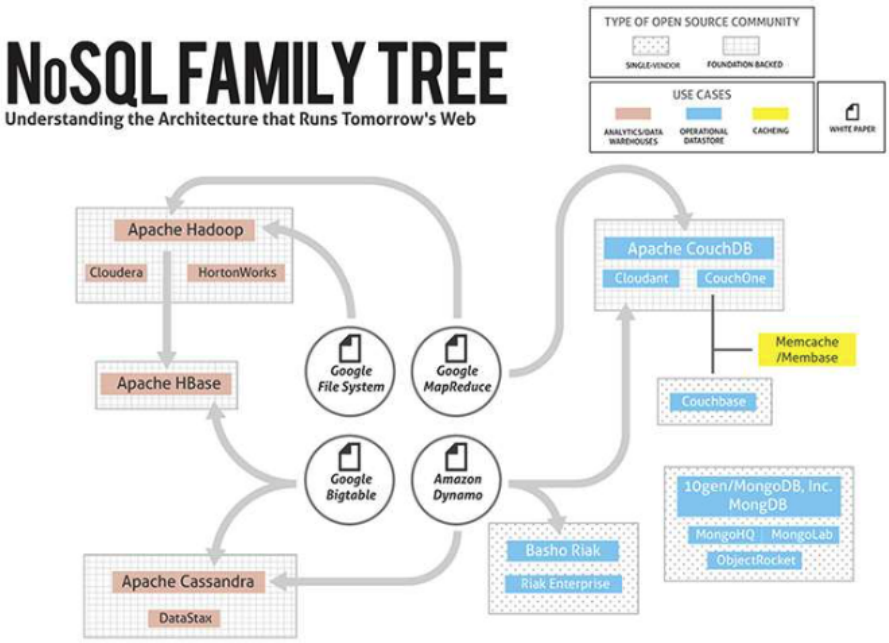
Figure 24: NoSQL types

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 ee 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.

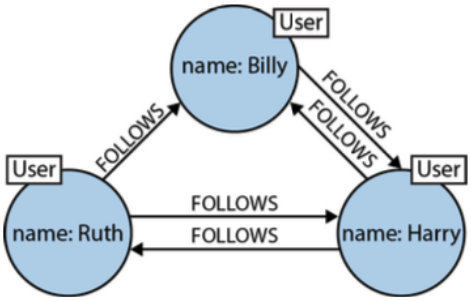


Figure 25: A small social graph

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.

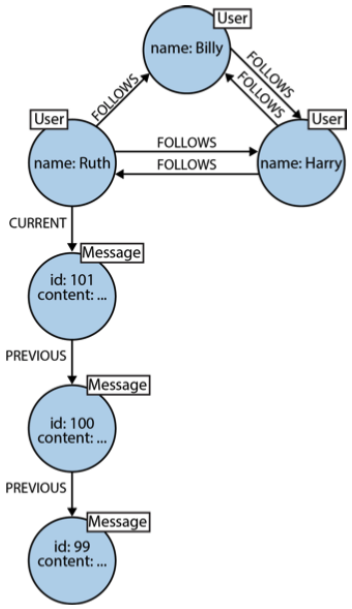


Figure 26: Publishing messages

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

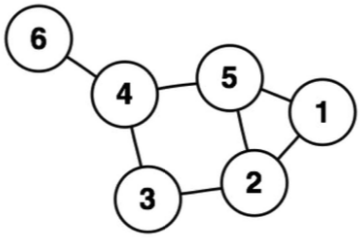


Figure 27: $V := \{1, 2, 3, 4, 5, 6\} - E : \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}$

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.

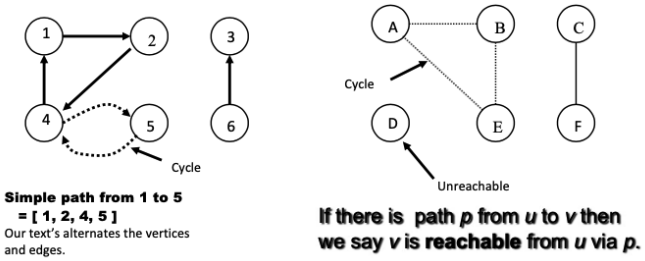


Figure 28: Graph path and reachability

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

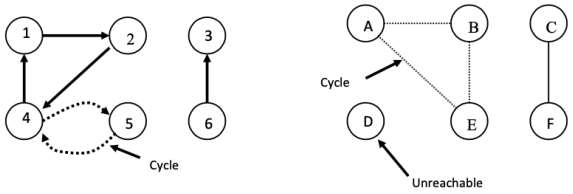


Figure 29: Graph cycle

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 $w : E \to R$.

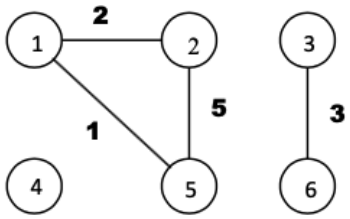


Figure 30: For example, in a GPS navigator we could use weight to specify duration, distance or traffic. The shortest path between two nodes is then calculated selecting the edges with the lowest weight.

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 $u \in V_2$

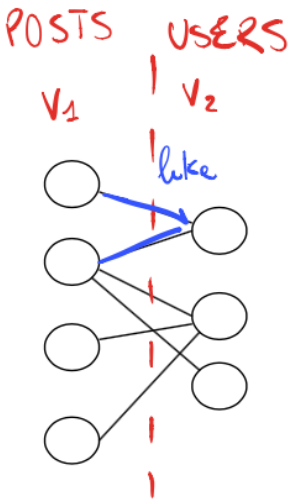


Figure 31: Nodes in V_1 connects only with nodes in V_2 (usually those are different categories of nodes e.g., Users and Posts)

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

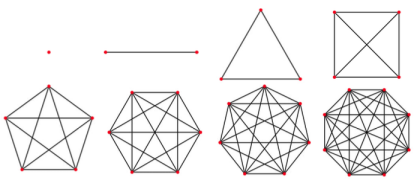


Figure 32: Exponential growth

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.

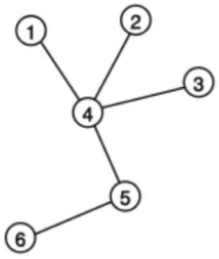


Figure 33: Example of Tree

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
- $Degree = indegree + outdegree$

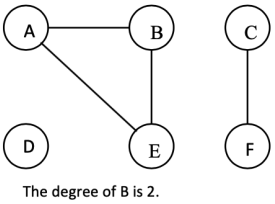


Figure 34: Node degree

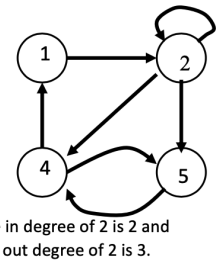


Figure 35: Node degree in directed graph

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.

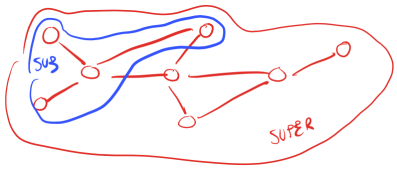


Figure 36: Subgraph and Supergraph

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

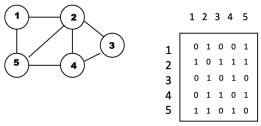


Figure 37: $|V|x|V|$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise.

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 transac- tional 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.

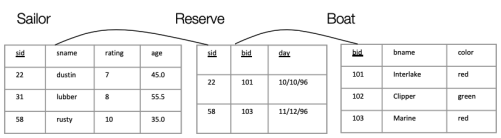


Figure 38: Relational DB with intermediate join table

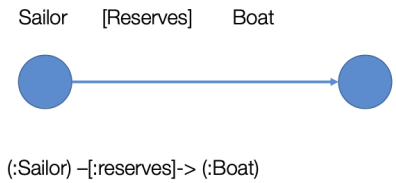


Figure 39: Graph DB model. Much simpler!

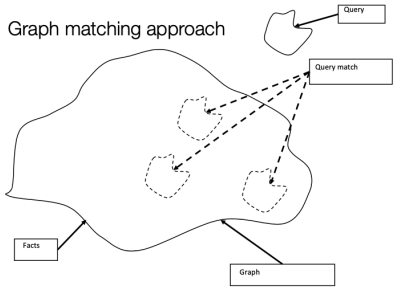


Figure 40: Querying a graph is similar to pattern matching. First, we define a pattern (shape of (sub)graph that we are looking for) and then we look in the graph for that shape.

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.)

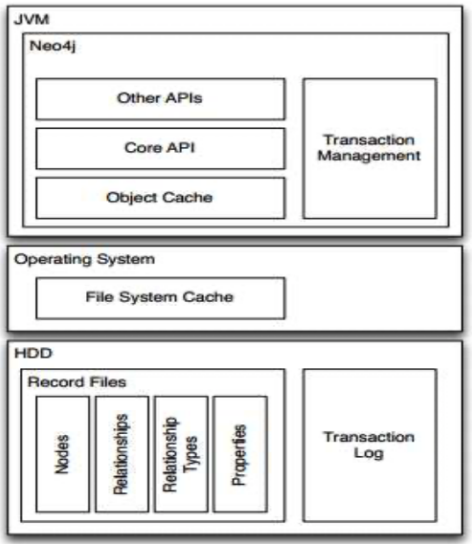


Figure 41: Neo4J software architecture. Each type of record (e.g., nodes, relationships) is stored in a separate and dedicated file. Traditional **C**onsistency and **Availability** support (no partitioning).

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 profes- sionals, 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.

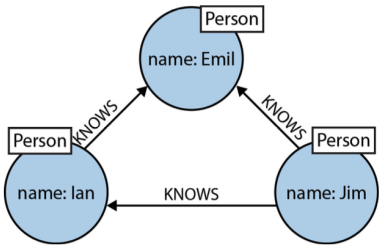


Figure 42: This pattern describes three mutual friends. Here's the equivalent ASCII art representation in Cypher: (emil)←[:KNOWS]−(jim)−[:KNOWS]→(ian)−[:KNOWS]→(emil)

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* identifer, 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 **DI**ctionary **S**erver (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.

Figure 48: Redis commands. Full command reference here

4.2.1 Scaling Redis

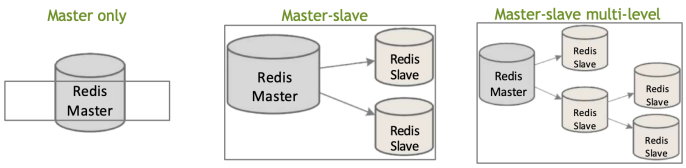
- **Persistence**: how can we be sure that we have some persistent storage of data (since Redis works in main memory)?
  For this reason, REDIS provides two mechanisms to deal with persistence (very basic options):
  - Redis Database Snapshots (RDB): save memory snapshot on disk (like a backup)
  - append-only 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)

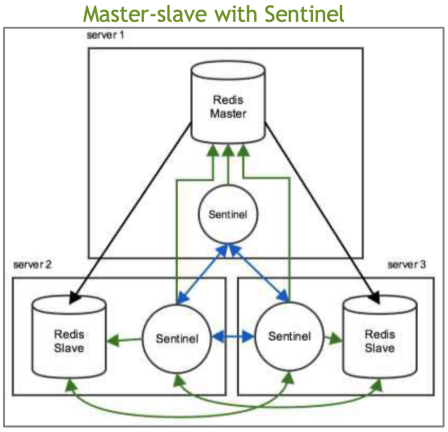
I - Standalone

- 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.



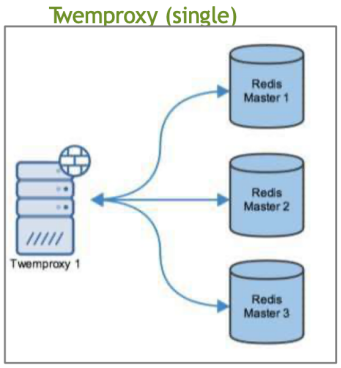
II - Sentinel

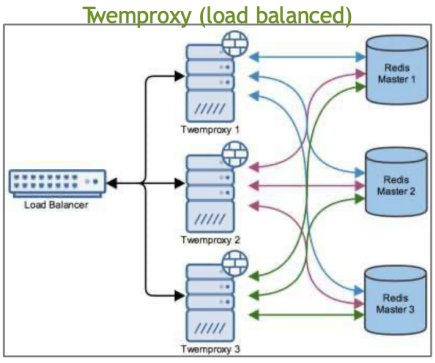
- Redis Sentinel provides a reliable **automatic failover** in a master/slave topology, automatically promoting a slave to master if the existing master fails.
- Every deployment (master or slave) have a sentinel component that is able to check the status of the other servers. If the master goes down,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



III - Twemproxy

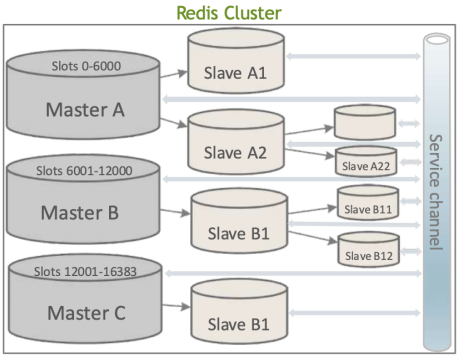
- Twemproxy (a project by twitter) works as a proxy between the clients and many Redis instances.
- Is able to **automatically distribute data** among different standalone Redis instances.
- Supports consistent hashing with different strategies and hashing functions
- Multi-key commands and transactions 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

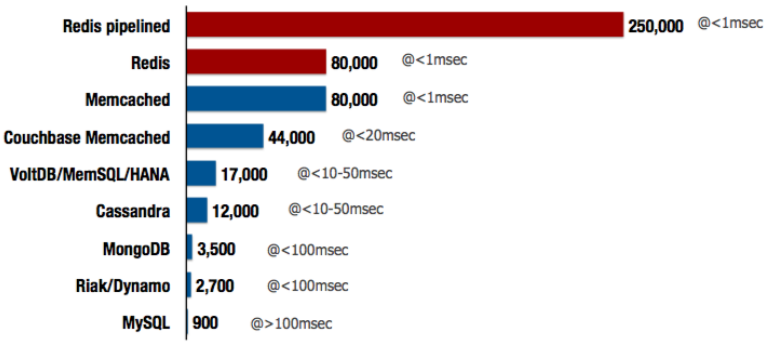


Figure 49: NoSQL & SQL response performance comparison.
Num of operations per unit of time - Average query time

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 data / 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 outweights the overhead of checking and keeping your cache up to data
- 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)

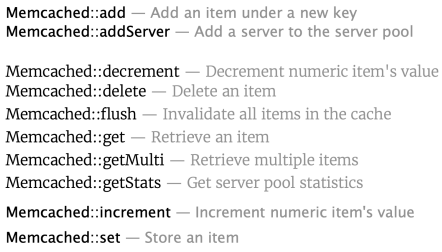


Figure 50: Memcached PHP Client functions.

```php
<?php

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 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.

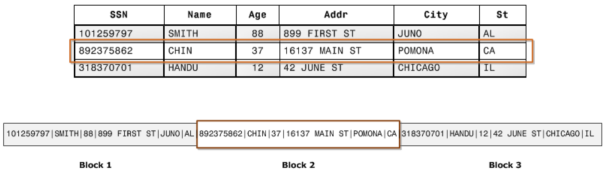


Figure 52: Row-wise

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.

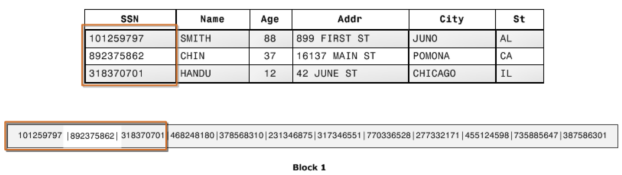


Figure 53: Column-wise

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 structured)
- 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.

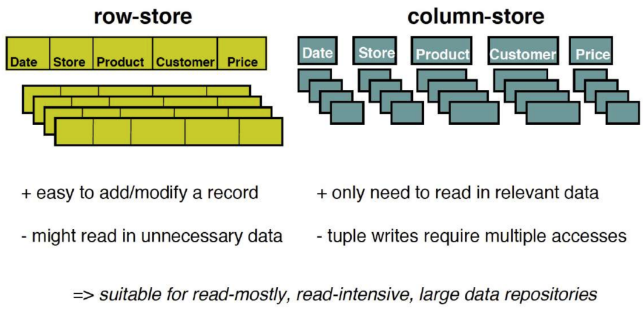


Figure 54: Column vs. Row data storage

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

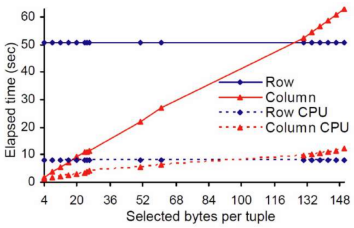


Figure 55: Row based always read the entire row (constant time). Column based instead are more efficient when few bytes have to be read. Then there is a breakeven point after which row based databases become more efficient to read data.

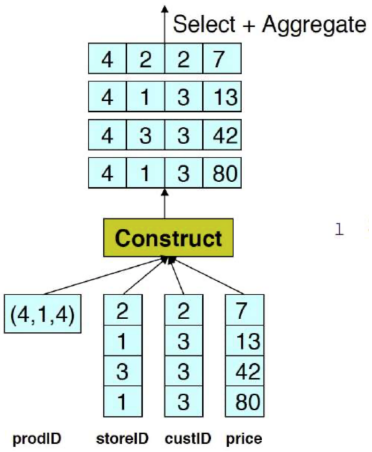


Figure 56: Tuple reconstruction.

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:

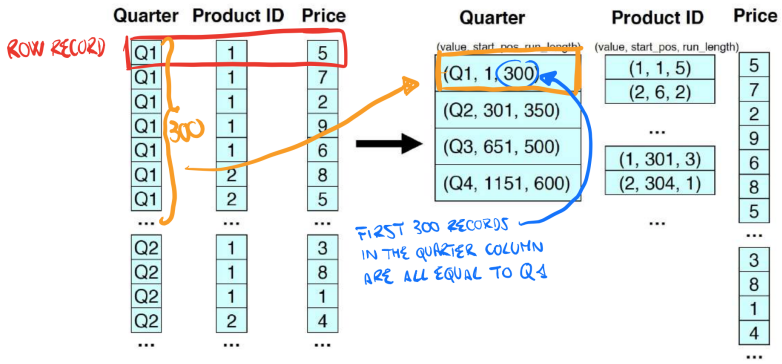


Figure 57: Column compression example using Run Length encoding

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.

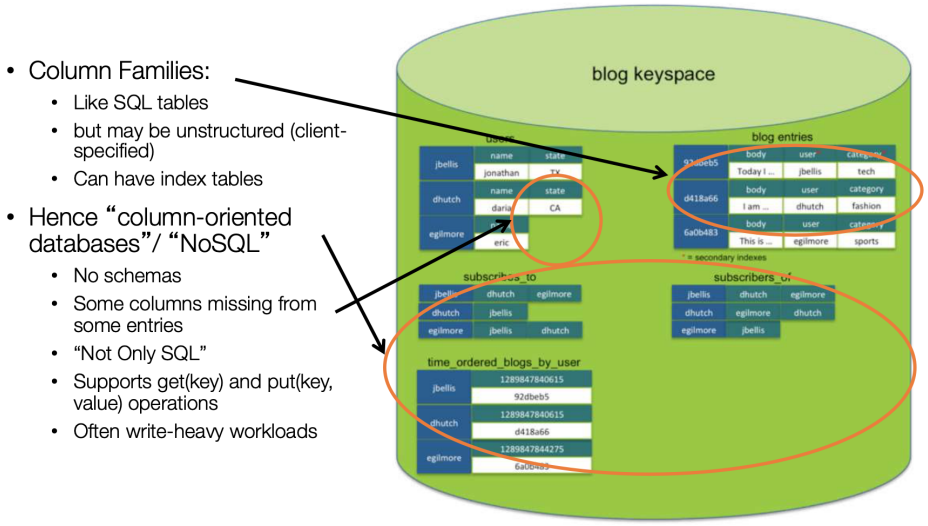


Figure 58: Cassandra Data Model

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**
- *tuneably* 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.

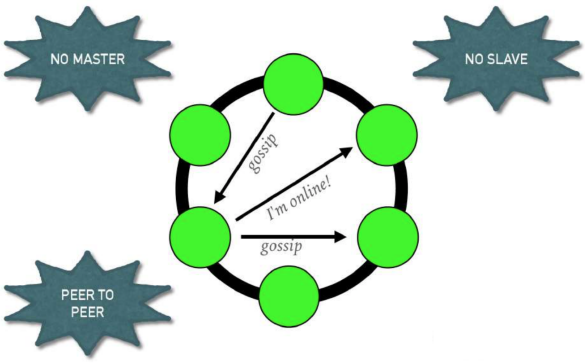


Figure 60: Cassandra Gossip Protocol

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.

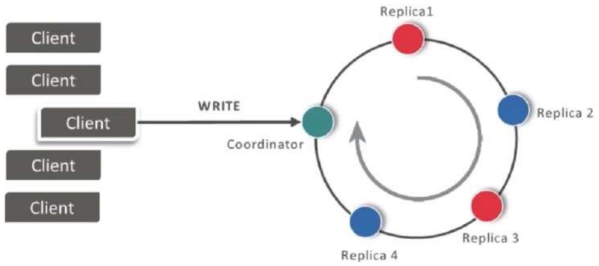
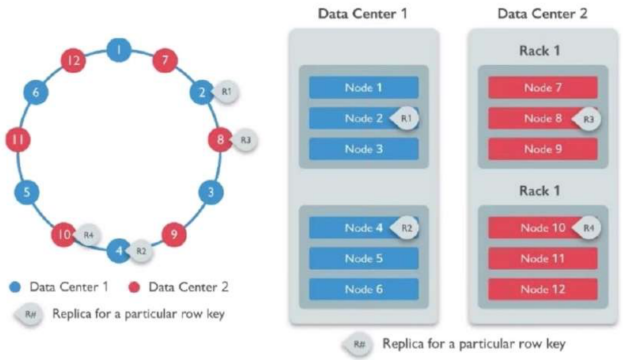


Figure 61: Cassandra's topology can be imagined as a ring, in which nodes are all equivalent (master-less). The process of replication is managed by a coordinator (who changes each time), who's in charge of handling the a write operation and then start propagating the new value to the others. The replicas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

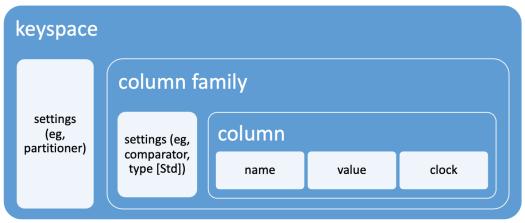


Figure 64: Cassandra data model

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.

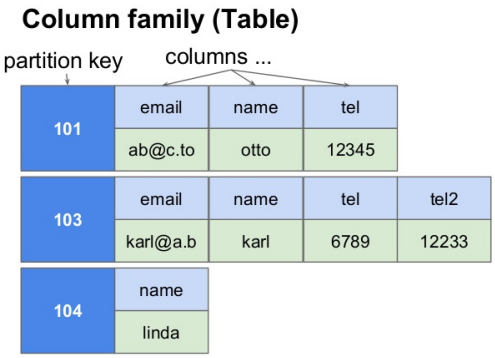


Figure 65: Cassandra column families are sparse tables. Some rows may contain all columns, while in other rows there could be only some of them. As in this case, row 104 has only one column while row 103 has 4 columns.

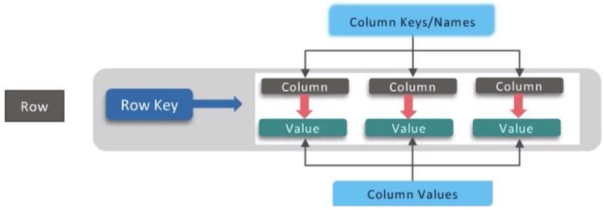


Figure 66: Cassandra: hybrid key-value based + column based database.

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.

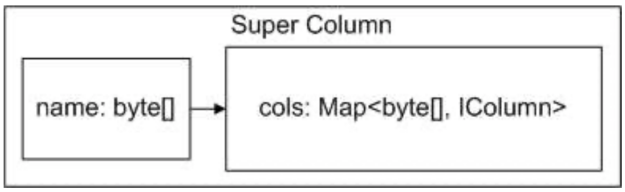


Figure 67: Cassandra supercolumn

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

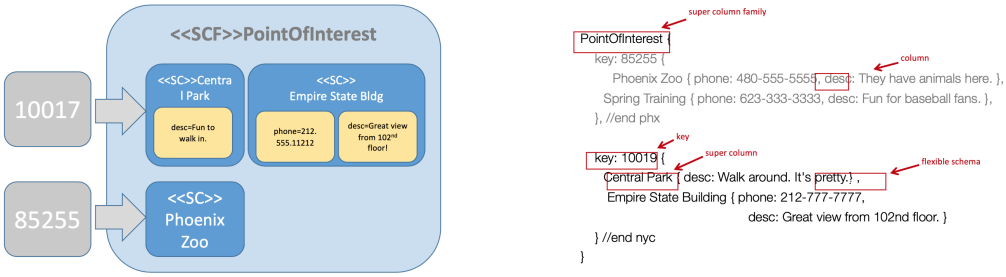


Figure 68: Supercolumn families

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.

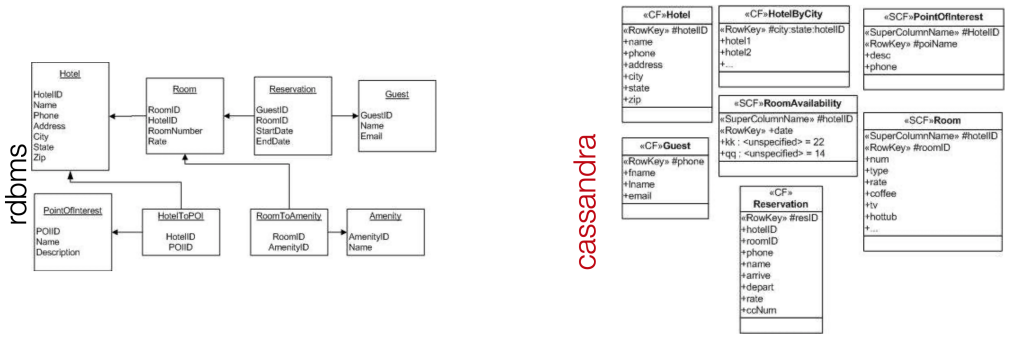


Figure 70: RDBMS schema are rigid schema defined when building up the database. Cassandra schema is more flexible because new column families are continuously added according to query needs.

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.

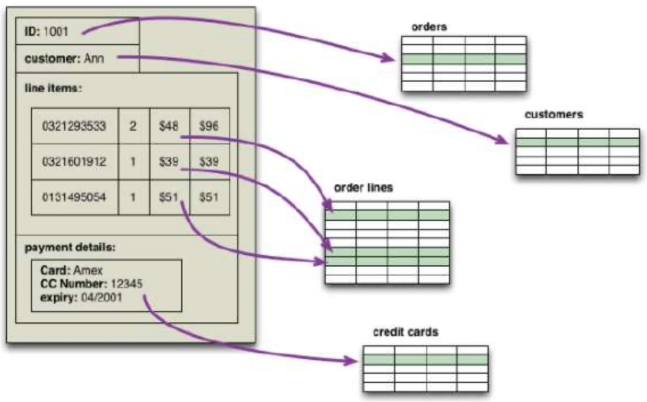


Figure 71: Example of document (invoice) stored in a document-based database.

The document in Figure 71 is made of nested data object 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.

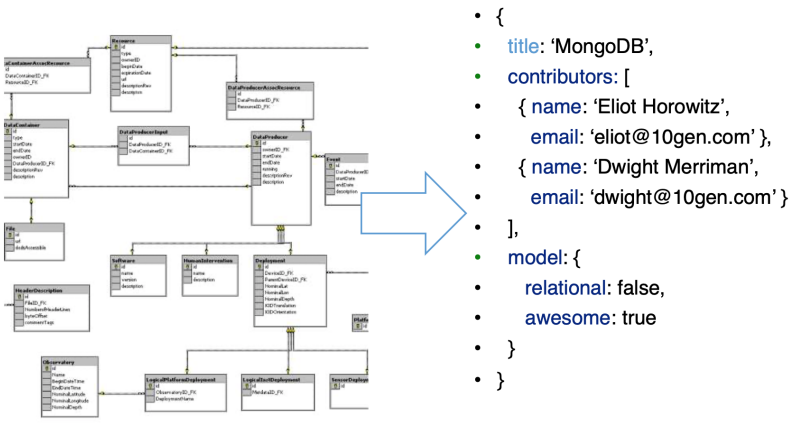


Figure 72: Example of JSON document

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

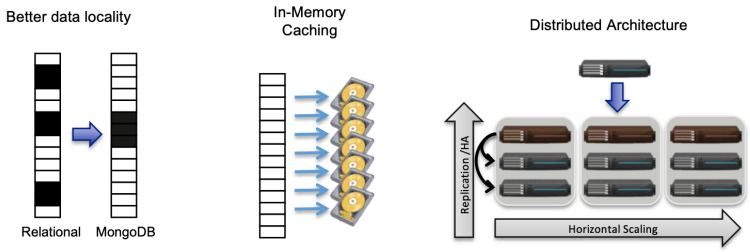


Figure 73: MongoDB features.

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

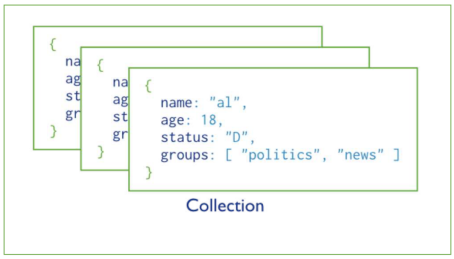


Figure 75: A collection includes a set of documents.

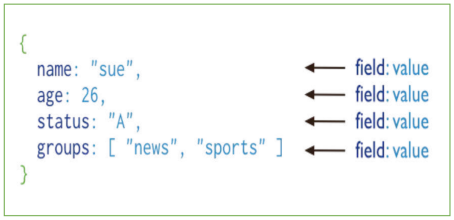


Figure 76: Structure of a JSON-document. The value of field could be one of: native data types, arrays, other documents. Rule: every document must have an _id.

Figure 77: Embdedded documents.

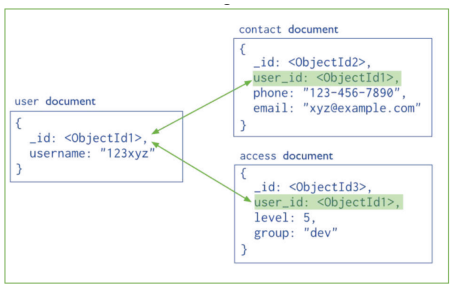


Figure 78: Reference documents or linking documents.

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 loginal 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

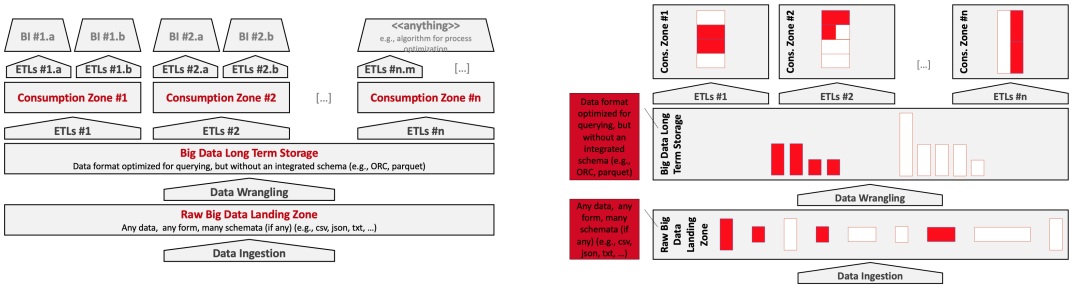
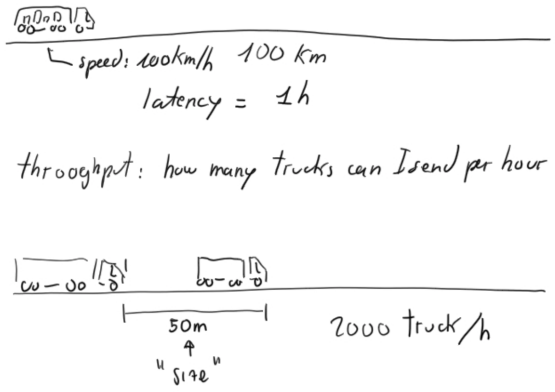


Figure 81: Logical architecture of a Big Data Platform.

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)

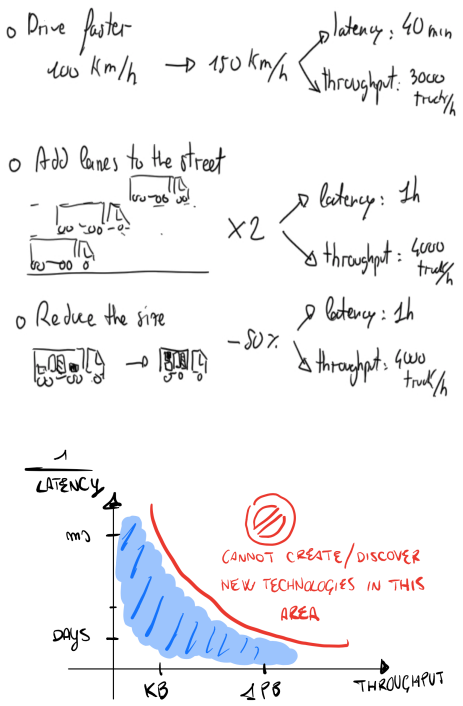


Figure 82: Latency-Throughput trade-off

7.1.2 Three Cases along a continuum

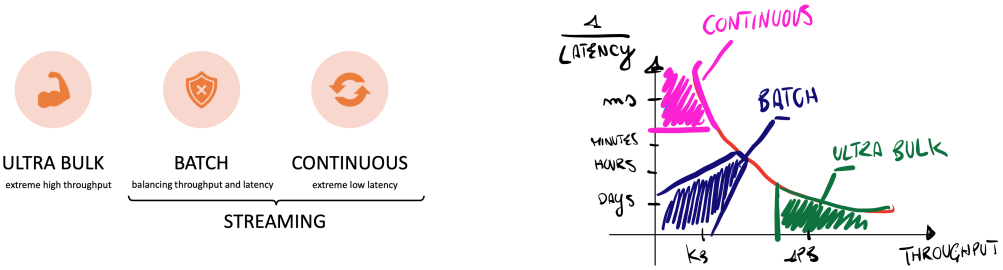


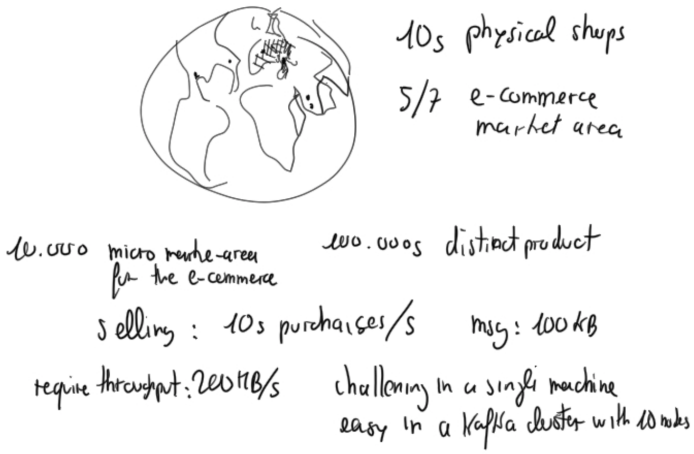
Figure 83: Three cases and their latency-throughput trade-off.

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.

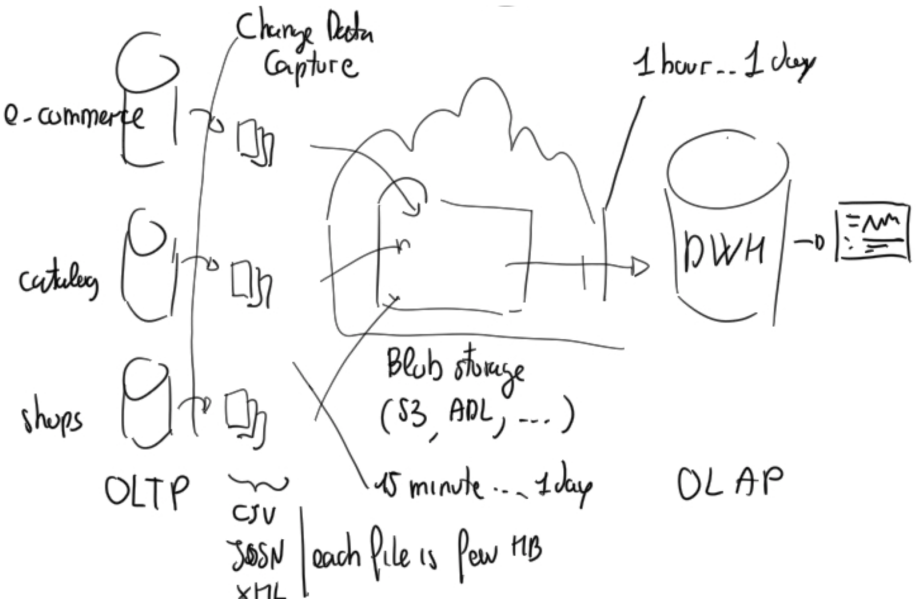
Continuous Case low throughput and low latency

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

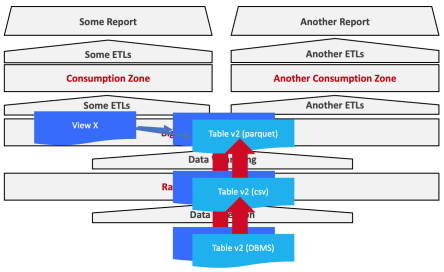


Figure 85: Overwrite option

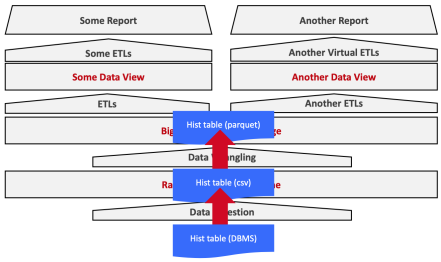
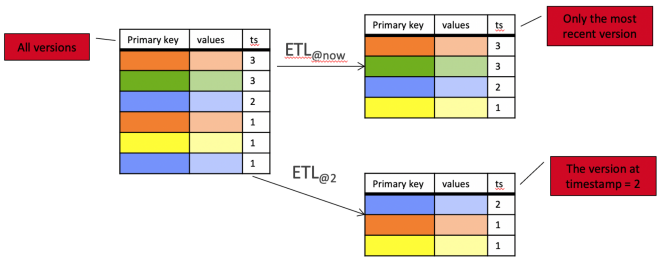


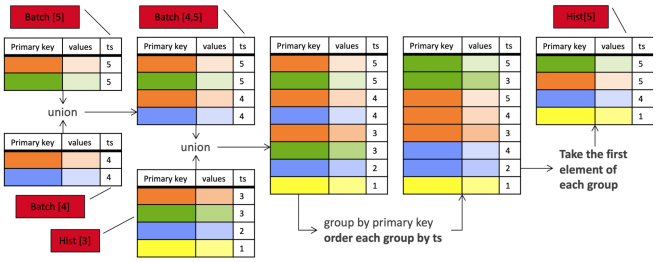
Figure 86: Batch option (1st batch case)

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	• Minimal ingestion & wrangling cost • The version of the data in any point in time can be rebuilt	• It requires more space than keeping only the most recent version • If multiple ETLs access the same data the same computation may be repeated • Avoidable by introducing a view that computes the shared data (e.g. ETL@now)
Compaction	• It requires less space than keeping all versions as in append only mode • Multiple ETLs can access the same data (@now) without additional computation	• It requires more computation that 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

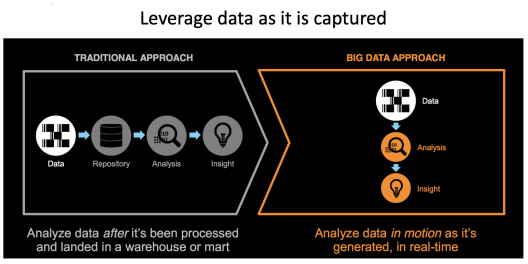


Figure 88: The continuous case is based on this paradigm shift.

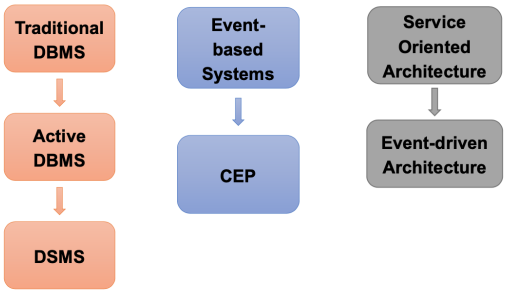


Figure 89: Many path to the same destination...

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

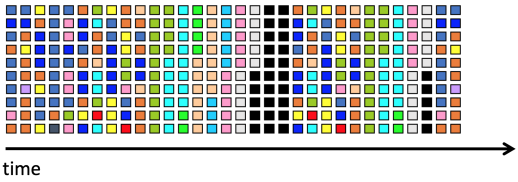


Figure 90: Unbounded window of data: up to a certain point is finite but it grows infinitely

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

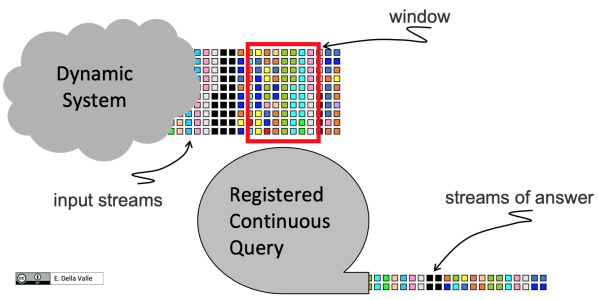


Figure 91: Continuous queries registered over strems that are observed through windows.

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

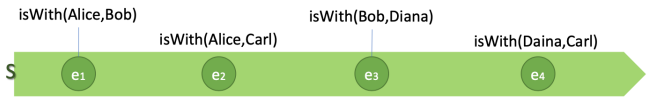


Figure 92: We don't know the distance between e1 and e2, we only know that e2 is after e1. The order can be exploited to perform queries like *Does Alice meet Bob before Carl?*

- **Absolute** - distance before events

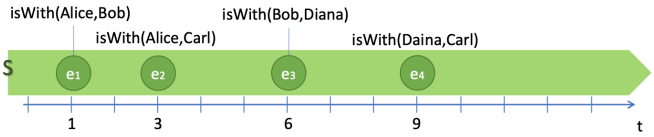


Figure 93: We can ask the queries presented in the causal model. WE can start to compose queries taking into account the time *How many people has Alice met in the last 5m* (window of 5mins opened in the past) or *Does Diana meet Bob and then Carl withing 5m?* (window opened in the future).

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

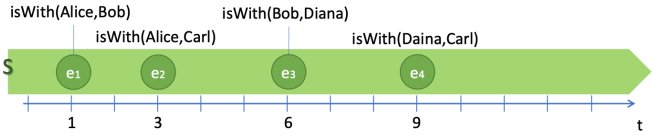


Figure 94: We can ask the queries presented in the previous cases. It is possible to write even more complex queries like *Which are the meetings that last less than 5m?* or *Which are the meetings with conflicts?*

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

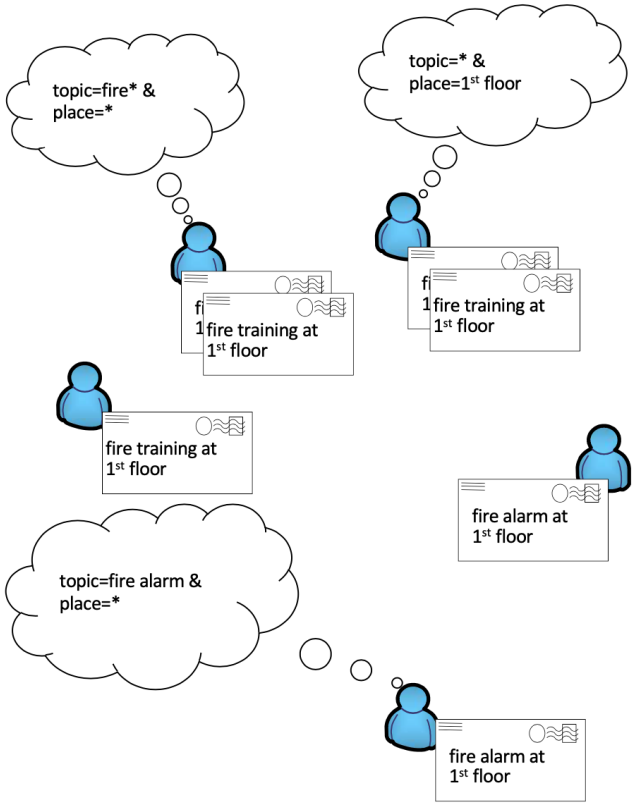


Figure 95: Event-based system. Some components are publishing events and some others are listening for specific events. In this example, the up left component is listening for events that include ”fire” happening in any place (* placeholder).

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.

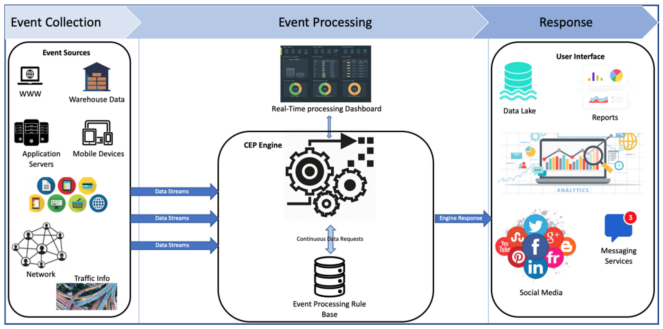


Figure 96: An idea of how a CEP system might look.

Relation	Illustration	Interpretation
$X < Y$ $Y > X$	X Y	X takes place before Y
X **m** Y Y **mi** X	X Y	X meets Y (*i* stands for ***inverse***)
X **o** Y Y **oi** X	X Y	X overlaps with Y
X **s** Y Y **si** X	X Y	X starts Y
X **d** Y Y **di** X	X Y	X during Y
X **f** Y Y **fi** X	X Y	X finishes Y
$X = Y$	X 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)

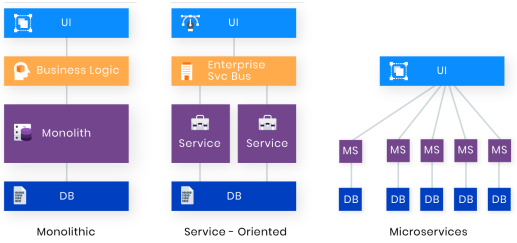


Figure 99: Evolution of software architectures.

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.

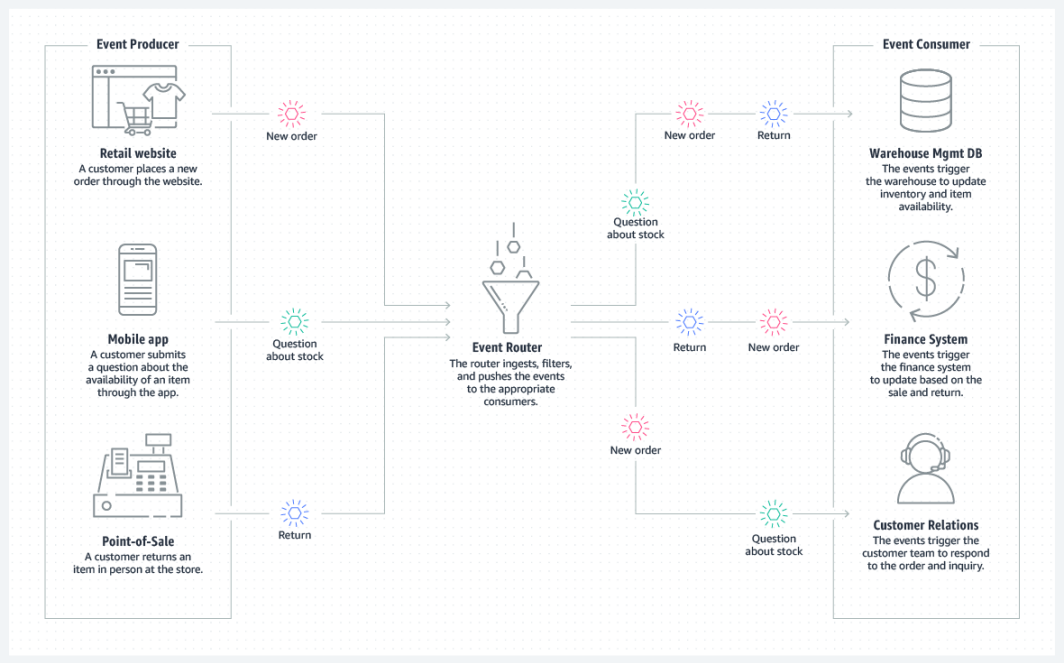


Figure 100: How it works an event-driven architecture

8 EPL

8.1 EPL and Esper

The **E**vent **P**rocessing **L**anguage (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.

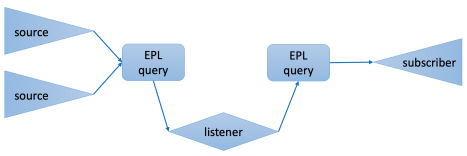


Figure 101: EPL processing model

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

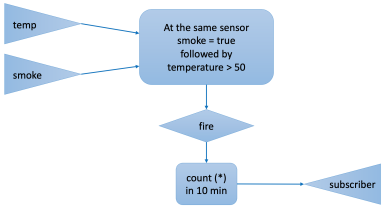


Figure 102: EPL processing model - Fire example

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
);
```

Figure 105: EPL - Declare event types example

```
select      *
from        TemperatureSensorEvent
where       temperature>50

select      avg(temperature)
from        TemperatureSensorEvent
```

Figure 106: EPL - Query syntax example

Type	Syntax	Description
Logical Sliding	win:time(*time_period*)	Sliding window that covers the specified time interval into the past
Logical Tumbling	win:time_batch(*time_period* [, *reference point*] [, *flow control*])	Tumbling window that batches events and releases them every specified time interval, with flow control options
Physical Sliding	win:length(*size*)	Sliding window that covers the specified number of elements into the past
Physical Tumbling	win:length_batch(*size*)	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.

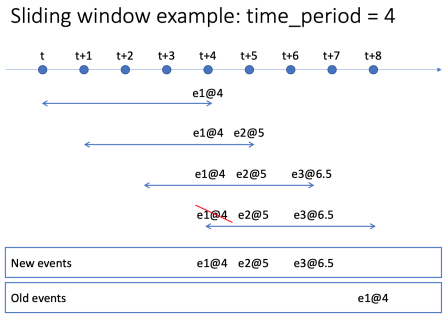


Figure 108: Window opened 4 seconds in the past. At t+8 e1@4 is discarded since is more than 4 seconds in the past.

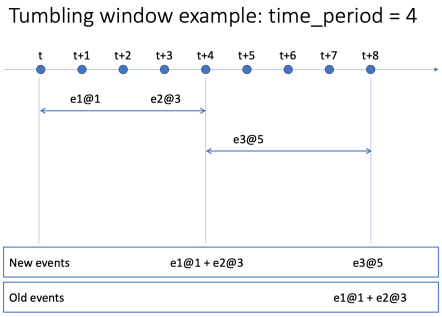


Figure 109: Each window is discarded after 4 seconds, and a new one is created.

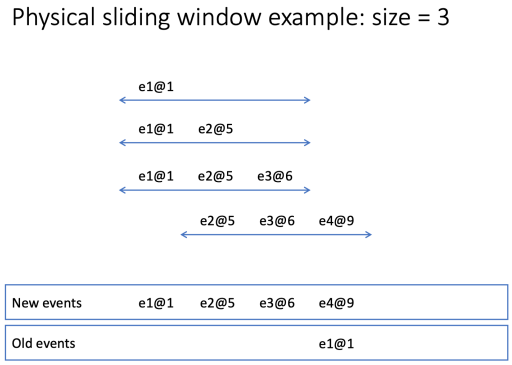


Figure 110: The window counts the number of elements seen so far, allowing to store only the last 3 of them. That's why e1@1 is discarded when e4@9 is seen.

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:

- $TempStream(sensor = "S0", val > 50)$

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

- $timer : interval(10seconds)$ fires after 10 seconds
- $timer : at(5, *, *, *, *)$ 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* $A \rightarrow B$ - The pattern fires for every A followed by a B event.
  Matches:
  $\{A1, B1\}, \{A2, B3\}, \{A3, B3\}, \{A4, B4\}$
- $A \rightarrow$ *every* B - The pattern fires for an A event followed by every B event.
  Matches:
  $\{A1, B1\}, \{A1, B2\}, \{A1, B3\}, \{A1, B4\}$
- *every* $A \rightarrow$ *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.

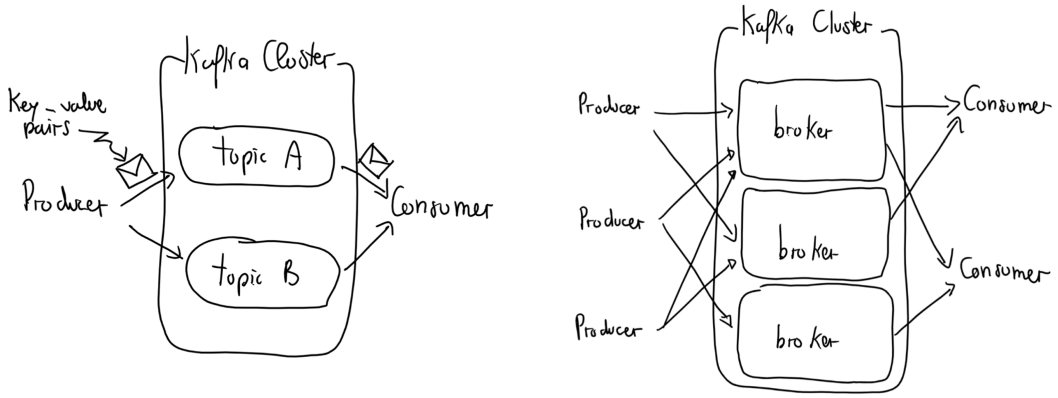


Figure 116: Kafka producers and consumers exchange messages through topics which are included within brokers.

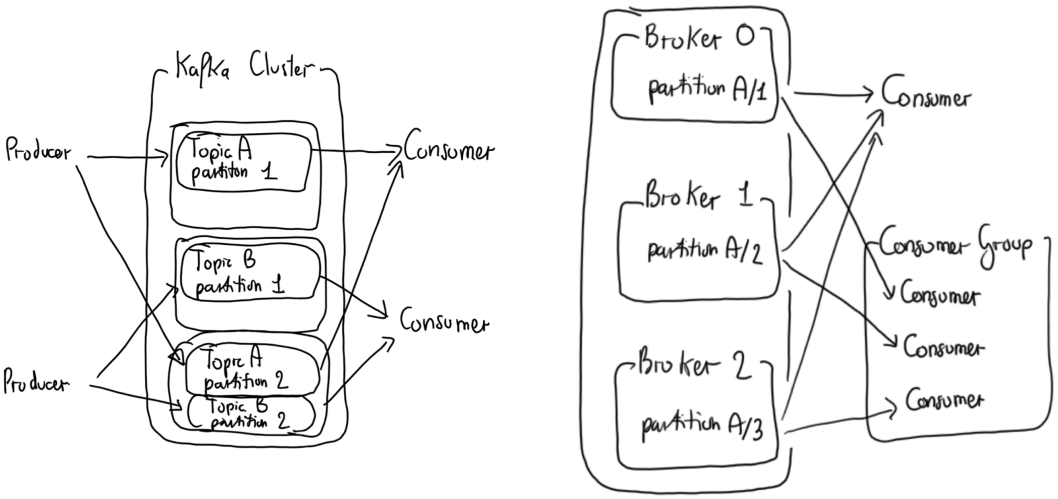


Figure 117: Using partitions, each broker can contain different topics and different partitions in order to improve availability (by replicating data) and scalability (by parallelizing message reading/writing). Moreover, consumer groups allow to scale the consumption process.

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

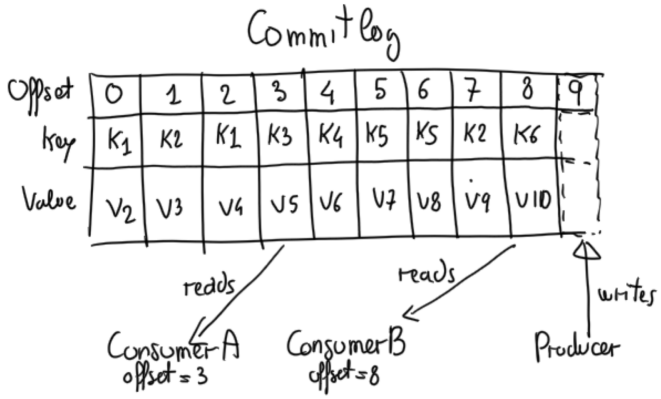


Figure 118: Consumer A after reading on offset 3, will read on offset 4. Consumer B can't read anything more, since 8 is the last offset that was written.

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**

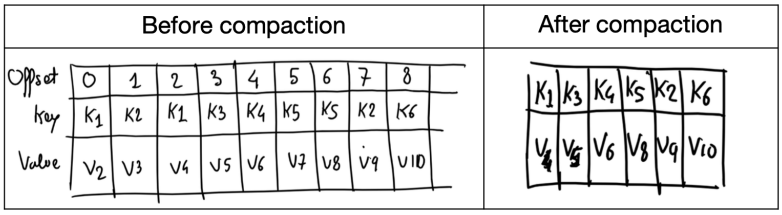


Figure 119: A compacted log retains at least the last known message value for each key within the partition.

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 sematics)
- **acks=all**: producer will wait until all all insync replicas have acknowledged receipt of the record

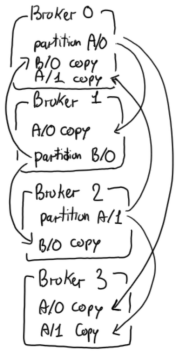


Figure 120: Fault tolerance via a Replicated Log

9.1.4 Zookeeper

Zookeeper is a top-level software developed by Apache that acts as a centralized service and is used to maintain naming and configuration data and to provide flexible and robust synchronization within distributed systems. Zookeeper keeps track of status of the Kafka cluster nodes and it also keeps track of Kafka topics, partitions etc.

How does it work:
The data within Zookeeper is divided across multiple collection of nodes and this is how it achieves its high availability and consistency. In case a node fails, Zookeeper can perform instant failover migration; e.g. if a master node fails, a new one is selected in real-time by polling within an ensemble. A client connecting to the server can query a different node if the first one fails to respond.

Why is Zookeeper necessary for Apache Kafka?

- **Controller election:** the controller is one of the most important broking entity in a Kafka ecosystem, and it also has the responsibility to maintain the leader-follower relationship across all the partitions. If a node by some reason is shutting down, it's the controller's responsibility to tell all the replicas to act as partition leaders in order to fulfill the duties of the partition leaders on the node that is about to fail. So, whenever a node shuts down, a new controller can be elected and it can also be made sure that at any given time, there is only one controller and all the follower nodes have agreed on that.
- **Configuration of Topics:** the configuration regarding all the topics including the list of existing topics, the number of partitions for each topic, the location of all the replicas, list of configuration overrides for all topics and which node is the preferred leader, etc.
- **Access control lists:** access control lists or ACLs for all the topics are also maintained within Zookeeper.

- **Membership of the cluster:** Zookeeper also maintains a list of all the brokers that are functioning at any given moment and are a part of the cluster.

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.

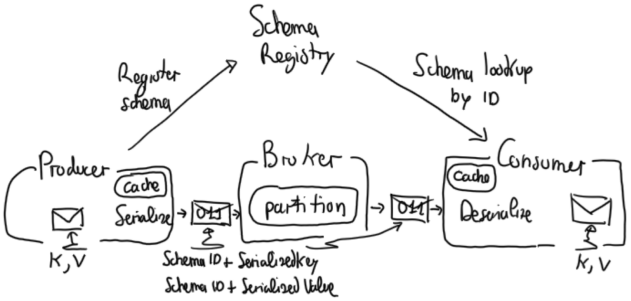


Figure 122: **Schema registry at work**. The producer, after having registered a schema, encodes messages containing a concatenation of the SchemaID and the content (key and value). Then, the consumer extracts the SchemaID, lookups the schema on the registry and decodes the message's content.

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.

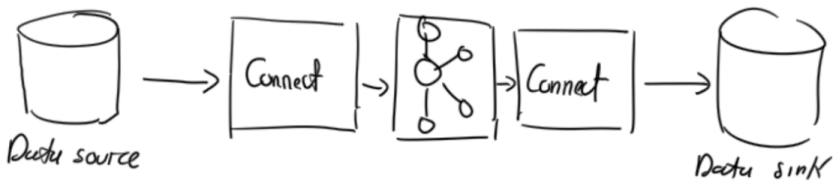


Figure 124: Kafka Connect: simpe, scalable and reliable.

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.

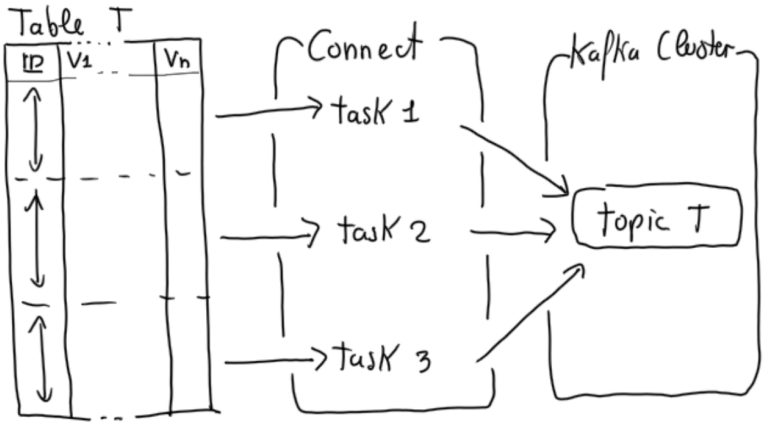


Figure 125: Parallelism and scalability is achieved splitting the workload into pieces run by a worker as different task, each in a separated thread.

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.

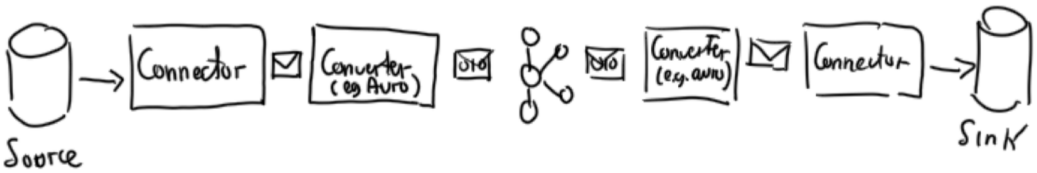


Figure 126: How converters are used when reading from a database using a Source Connector (e.g., MongoDB), writing to Kafka, and finally, writing on a Sink Connector (e.g., MySQL).

9.4 Kafka Stream Processing

Kafka Streams is a library for building streaming applications, specifically applications that transform input Kafka topics into output Kafka topics (or calls to external services, or updates to databases, or whatever). It lets you do this with concise code in a way that is distributed and fault-tolerant. Stream processing is a computer programming paradigm, equivalent to data-flow programming, event stream processing, and reactive programming, that allows some applications to more easily exploit a limited form of parallel processing.

9.4.1 Stream vs Table

In Kafka, events are captured by an event streaming platform into event streams. **An event stream records the history of what has happened in the world as a sequence of events.** An example stream is a sales ledger or the sequence of moves in a chess match. With Kafka, such a stream may record the history of your business for hundreds of years. This history is an ordered sequence or chain of events, so we know which event happened before another event to infer causality (e.g., "White moved the e2 pawn to e4, then Black moved the e7 pawn to e5"). A stream thus represents both the past and the present: as we go from today to tomorrow-or from one millisecond to the next-new events are constantly being appended to the history.

Compared to an event stream, **a table represents the state of the world at a particular point in time**, typically "now." An example table is total sales or the current state of the board in a chess match. A table is a view of an event stream, and this view is continuously being updated whenever a new event is captured.

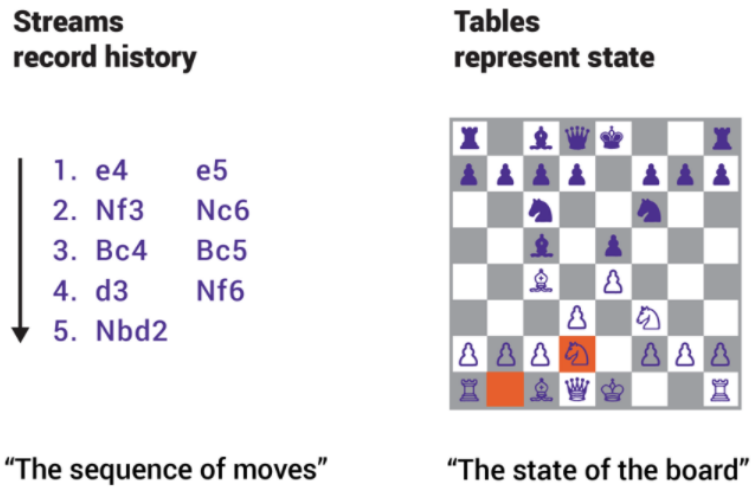


Figure 127: Kafka Stream vs Table

Streams and tables in Kafka differ in a few ways, notably with regard to whether their contents can be changed, i.e., whether they are *mutable*. (as for table we refer to what is called a KTable in Kafka Streams)

- A **stream** provides immutable data. It supports only inserting (appending) new events, whereas existing events cannot be changed. Streams are persistent, durable, and fault tolerant. Events in a stream can be keyed, and you can have many events for one key, like "all of Bob's payments." If you squint a bit, you could consider a stream to be like a table in a relational database (RDBMS) that has no unique key constraint and that is append only.
- A **table** provides mutable data. New events-rows can be inserted, and existing rows can be updated and deleted. Here, an event's key aka row key identifies which row is being mutated. Like streams, tables are persistent, durable, and fault tolerant. Today, a table behaves much like an RDBMS materialized view because it is being changed automatically as soon as any of its input streams or tables change, rather than letting you directly run insert, update, or delete operations against it.

	Stream	Table
First event with key `bob` arrives	Insert	Insert
Another event with key `bob` arrives	Insert	Update
Event with key `bob` and value `null` arrives	Insert	Delete
Event with key `null` arrives	Insert	<ignored>

Figure 128: Behavior of streams and tables

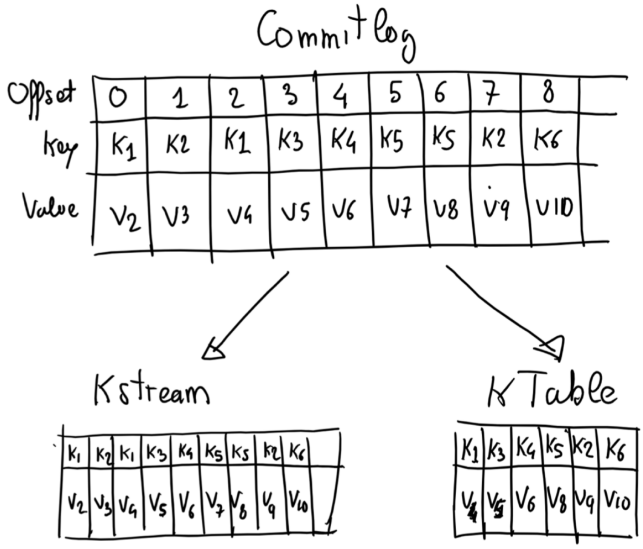
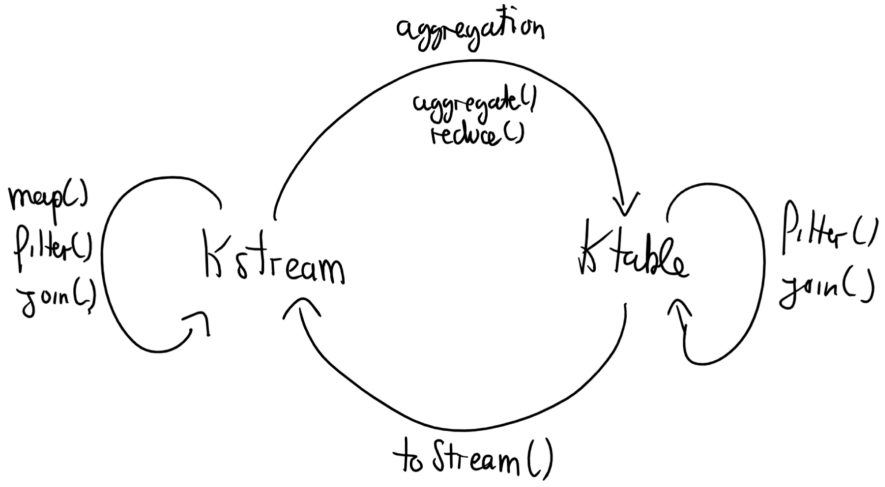


Figure 129: A **KStream** is an abstraction of a record stream. Each record in it represents a self-contained piece of data in the unbounded data set. A **KTable** is an abstraction of a changelog stream. Each record in it represents an update (it keeps only the most recent value for each key).

9.4.2 Stream-Table duality

Notwithstanding their differences, we can observe that there is a close relationship between a stream and a table. We call this the stream-table duality. What this means is:

- We can turn a stream into a table by aggregating the stream with operations such as $COUNT()$ or $SUM()$, for example. In our chess analogy, we could reconstruct the board's latest state (table) by replaying all recorded moves (stream).
- We can turn a table into a stream by capturing the changes made to the table—inserts, updates, and deletes—into a "change stream." This process is often called change data capture or CDC for short. In the chess analogy, we could achieve this by observing the last played move and recording it (into the stream) or, alternatively, by comparing the board's state (table) before and after the last move and then recording the difference of what changed (into the stream), though this is likely slower than the first option.



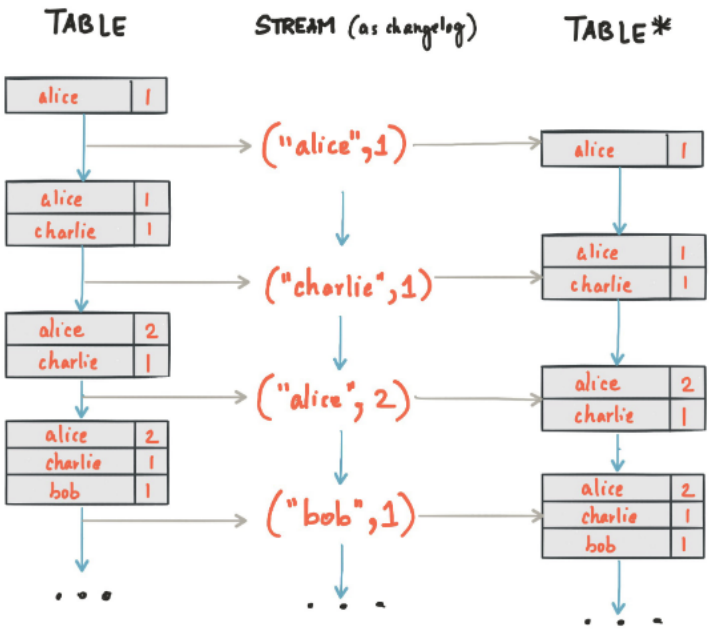


Figure 130: Kafka Stream-Table Duality

10 KSQL

10.1 Introducing KSQL: Streaming SQL for Apache Kafka

KSQL is a streaming SQL engine for Apache Kafka®. KSQL lowers the entry bar to the world of stream processing, providing a simple and completely interactive SQL interface for processing data in Kafka. KSQL is distributed, scalable, reliable, and real time. It supports a wide range of powerful stream processing operations including aggregations, joins, windowing, sessionization, and much more.

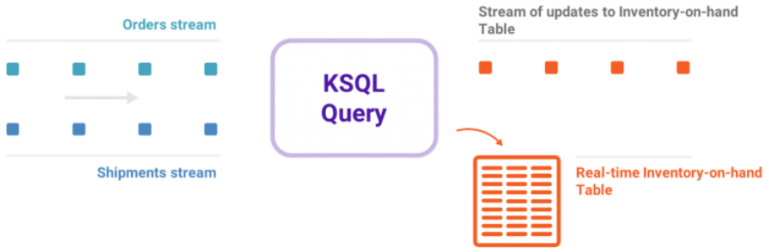


Figure 131: A simple example.

What does it even mean to query streaming data, and how does this compare to a SQL database?
Well, it's actually quite different to a SQL database. Most databases are used for doing on-demand lookups and modifications to stored data. KSQL doesn't do lookups (yet), what it does do is continuous transformations— that is, stream processing. For example, imagine that I have a stream of clicks from users and a table of account information about those users being continuously updated. KSQL allows me to model this stream of clicks, and table of users, and join the two together. Even though one of those two things is infinite.

So what KSQL runs are **continuous queries** — transformations that run continuously as new data passes through them — on streams of data in Kafka topics. In contrast, queries over a relational database are *one-time queries* — run once to completion over a data set—as in a SELECT statement on finite rows in a database.

10.1.1 What is KSQL good for?

1. **Real-time monitoring meets real-time analytics**

   ```
   CREATE TABLE error_counts AS
   SELECT error_code, count(*) FROM monitoring_stream
   WINDOW TUMBLING (SIZE 1 MINUTE)
   WHERE type = 'ERROR'
   ```

   One use of this is defining custom business-level metrics that are computed in real-time and that you can monitor and alert off of, just like you do your CPU load. KSQL allows defining custom metrics off of streams of raw events that applications generate, whether they are logging events, database updates, or any other kind.
   For example, a web app might need to check that every time a new customer signs up a welcome email is sent, a new user record is created, and their credit card is billed. These functions might be spread over different services or applications and you would want to monitor that each thing happened for each new customer within some SLA, like 30 secs.

2. **Security and anomaly detection**

   ```
   CREATE TABLE possible_fraud AS
   SELECT card_number, count(*)
   FROM authorization_attempts
   WINDOW TUMBLING (SIZE 5 SECONDS)
   ```

```
GROUP BY card_number
HAVING count(*) > 3;
```

KSQL queries can transform event streams into numerical time series aggregates that are pumped into Elastic using the Kafka-Elastic connector and visualized in a Grafana UI. Rather than monitoring application behavior or business behavior you're looking for patterns of fraud, abuse, spam, intrusion, or other bad behavior. KSQL gives a simple, sophisticated, and real-time way of defining these patterns and querying real-time streams.

3. **Online data integration**

```
CREATE STREAM vip_users AS
SELECT userid, page, action
FROM clickstream c
LEFT JOIN users u ON c.userid = u.user_id
WHERE u.level = 'Platinum';
```

Much of the data processing done in companies falls in the domain of data enrichment: take data coming out of several databases, transform it, join it together, and store it into a key-value store, search index, cache, or other data serving system. For a long time, ETL — Extract, Transform, and Load — for data integration was performed as periodic batch jobs. For example, dump the raw data in real time, and then transform it every few hours to enable efficient queries. For many use cases, this delay is unacceptable. KSQL, when used with Kafka connectors, enables a move from batch data integration to online data integration. You can enrich streams of data with metadata stored in tables using stream-table joins, or do simple filtering of PII (personally identifiable information) data before loading the stream into another system.

4. **Application development**
Many applications transform an input stream into an output stream. For example, a process responsible for reordering products that are running low in inventory for an online store might feed off a stream of sales and shipments to compute a stream of orders to place. For more complex applications written in Java, Kafka's native streams API may be just the thing. But for simple apps, or teams not interested in Java programming a simple SQL interface may be what they're looking for.

10.1.2 Core Abstractions in KSQL

KSQL uses Kafka's Streams API internally and they share the same core abstractions for stream processing on Kafka. There are two core abstractions in KSQL that map to the two core abstractions in Kafka Streams and allow you to manipulate Kafka topics:

- **Stream:** a stream is an unbounded sequence of structured data ("facts"). For example, we could have a stream of financial transactions such as "Alice sent $100 to Bob, then Charlie sent $50 to Bob". Facts in a stream are immutable, which means new facts can be inserted to a stream, but existing facts can never be updated or deleted. Streams can be created from a Kafka topic or derived from existing streams and tables.

```
CREATE STREAM pageviews (viewtime BIGINT, userid VARCHAR, pageid VARCHAR)
WITH (kafka_topic='pageviews', value_format='JSON');
```

- **Table:** a table is a view of a STREAM or another TABLE and represents a collection of evolving facts. For example, we could have a table that contains the latest financial information such as "Bob's current account balance is $150". It is the equivalent of a traditional database table but enriched by streaming semantics such as windowing. Facts in a table are mutable, which means new facts can be inserted to the table, and existing facts can be updated or deleted. Tables can be created from a Kafka topic or derived from existing streams and tables.

```
CREATE TABLE users (registertime BIGINT, gender VARCHAR, regionid VARCHAR, userid  VARCHAR)
WITH (kafka_topic='users', value_format='DELIMITED');
```

10.1.3 KSQL Internals

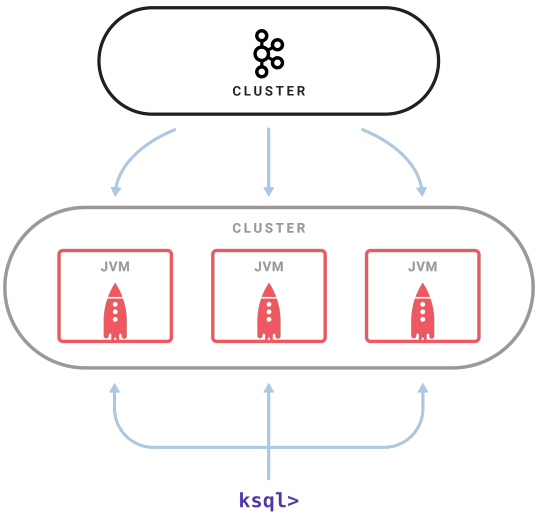
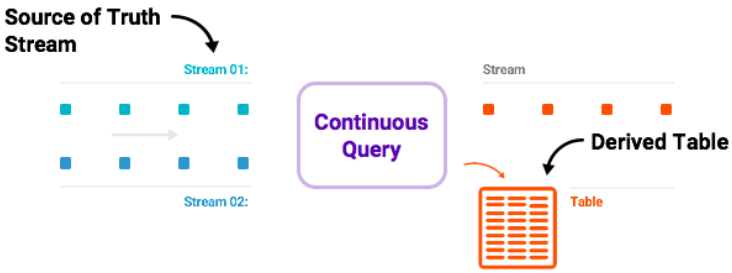


Figure 132: A look inside KSQL server

There is a KSQL server process which executes queries. A set of KSQL processes run as a cluster. You can dynamically add more processing capacity by starting more instances of the KSQL server. These instances are fault-tolerant: if one fails, the others will take over its work. Queries are launched using the interactive KSQL command line client which sends commands to the cluster over a REST API. The command line allows you to inspect the available streams and tables, issue new queries, check the status of and terminate running queries. Internally KSQL is built using Kafka's Streams API; it inherits its elastic scalability, advanced state management, and fault tolerance, and support for Kafka's recently introduced exactly-once processing semantics. The KSQL server embeds this and adds on top a distributed SQL engine (including some fancy stuff like automatic byte code generation for query performance) and a REST API for queries and control.

10.1.4 Kafka + KSQL turn the database inside out

In a relational database, the table is the core abstraction and the log is an implementation detail. In an event-centric world with the database is turned inside out, the core abstraction is not the table; it is the log. The tables are merely derived from the log and updated continuously as new data arrives in the log. The central log is Kafka and KSQL is the engine that allows you to create the desired materialized views and represent them as continuously updated tables. You can then run point-in-time queries against such streaming tables to get the latest value for every key in the log, in an ongoing fashion.



The Kafka log is the core storage abstraction for streaming data, allowing same data that went into your offline data warehouse is to now be available for stream processing. Everything else is a streaming materialized view over the log, be it various databases, search indexes, or other data serving systems in the company. All data enrichment and ETL needed to create these derived

views can now be done in a streaming fashion using KSQL. Monitoring, security, anomaly and threat detection, analytics, and response to failures can be done in real-time versus when it is too late. All this is available for just about anyone to use through a simple and familiar SQL interface to all your Kafka data: KSQL.

10.2 A DEMO of Kafka + KSQL + InfluxDB 2.0

This demo is an adaptation and extension to InfluxDB of the popular Workshop: Real-time SQL Stream Processing at Scale with Apache Kafka and KSQL (Mac/Linux).

Step list:

1. producing a topic
2. registering a stream
3. querying a stream
   (a) in real-time
   (b) in the past
4. creating a stream from a query
5. integrating a database via Kafka Connect
6. registering a tables
7. enriching a data stream joining it with a table
8. streaming aggregates
9. visual analytics in InfluxDB 2.0

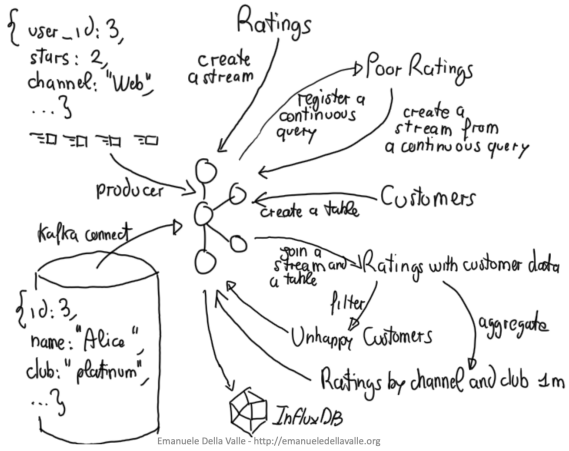
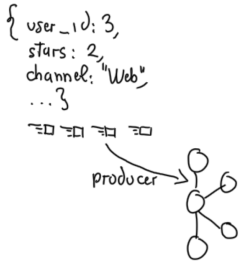


Figure 133: Demo scene (after having completed all the steps)

Before starting, make sure to have a KSQL server running on your machine. Check the link to the original demo provided at the beginning of this section, and follow the steps up to 3.

KSQL can be used via the command-line interface (CLI), a graphical UI built into Confluent Control Center, or the documented REST API. In this workshop, we will use the CLI, which if you have used Oracle's sql*plus, MySQL CLI, and so on will feel very familiar to you.

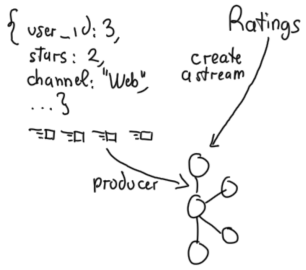
1. **Producing a topic**
   KSQL can be used to view the topic metadata on a Kafka cluster (SHOW TOPICS;), as well as inspect the messages in a topic (PRINT < *topic* >;).
   The event stream driving this example is a simulated stream of events purporting to show the ratings left by users on a website, with data elements including the device type that they used, the star rating, and a message associated with the rating. Using the PRINT command we can easily see column names and values within a topic's messages. Kafka messages consist of a timestamp, key, and message (payload), which are all shown in the PRINT output.




```
ksql> SHOW TOPICS;

 Kafka Topic                 | Registered | Partitions | Partition Replicas | Consumers | ConsumerGroups
---------------------------------------------------------------------------------------------------------
 _confluent-metrics          | false      | 12         | 1                  | 0         | 0
 _schemas                    | false      | 1          | 1                  | 0         | 0
 ratings                     | false      | 1          | 1                  | 0         | 0
[...]
---------------------------------------------------------------------------------------------------------
```

```
ksql> PRINT 'ratings';
Format:AVRO
22/02/18 12:55:04 GMT, 5312, {"rating_id": 5312, "user_id": 4, "stars": 4, "route_id": 2440, "rating_time": 1519304104965,
22/02/18 12:55:05 GMT, 5313, {"rating_id": 5313, "user_id": 3, "stars": 4, "route_id": 6975, "rating_time": 1519304105213,
```

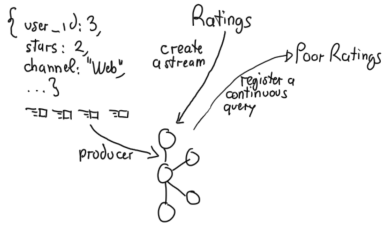
2. **From topics to streams**
   Having inspected the topics and contents of them, let's get into some SQL now. The first step in KSQL is to register the source topic with KSQL.
   By registering a topic with KSQL, we declare its schema and properties. The inbound event stream of ratings data is a STREAM. We just need a simple CREATE STREAM with the appropriate values in the WITH clause. We can notice that in the above CREATE STREAM statement we didn't specify any of the column names. That's because the data is in Avro format, and the Confluent Schema Registry supplies the actual schema details. We can use DESCRIBE to examine an object's columns.



```
ksql> CREATE STREAM ratings WITH (KAFKA_TOPIC='ratings', VALUE_FORMAT='AVRO');

 Message
----------------
 Stream created
----------------
```

```
ksql> DESCRIBE ratings;
Name                 : RATINGS
 Field       | Type
-----------------------------------------
 ROWTIME     | BIGINT           (system)
 ROWKEY      | VARCHAR(STRING)  (system)
 RATING_ID   | BIGINT
 USER_ID     | INTEGER
 STARS       | INTEGER
 ROUTE_ID    | INTEGER
 RATING_TIME | BIGINT
 CHANNEL     | VARCHAR(STRING)
 MESSAGE     | VARCHAR(STRING)
-----------------------------------------
For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Table>;
ksql>
```

3. **Querying streams** Let's run our first SQL. As anyone familar with SQL knows, SELECT * will return all columns from a given object. We'll notice that the data keeps on coming. That is because KSQL is fundamentally a streaming engine, and the queries that you run are continuous queries. Having previously set the offset to earliest KSQL is showing us the past (data from the beginning of the topic), the present (data now arriving in the topic), and the future (all new data that arrives in the topic from now on).



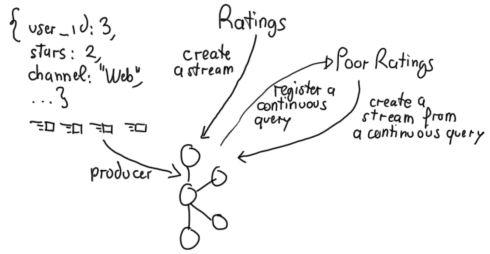

```
ksql> SELECT * FROM ratings;
1529501380124 | 6229 | 6229 | 17 | 2 | 3957 | 1529501380124 | iOS-test | why is it so difficult to keep the bathrooms clea
1529501380197 | 6230 | 6230 | 14 | 2 | 2638 | 1529501380197 | iOS | your team here rocks!
1529501380641 | 6231 | 6231 | 12 | 1 | 9870 | 1529501380641 | iOS-test | (expletive deleted)
[…]
```

Since KSQL is heavily based on SQL, you can do many of the standard SQL things we'd expect to be able to do, including predicates and selection of specific columns.

```
ksql> SELECT USER_ID, STARS, CHANNEL, MESSAGE FROM ratings WHERE STARS <3 AND CHANNEL='iOS' LIMIT 3;
3 | 2 | iOS | your team here rocks!
2 | 1 | iOS | worst. flight. ever. #neveragain
15 | 2 | iOS | worst. flight. ever. #neveragain
Limit Reached
Query terminated
ksql>
```

Since Apache Kafka persists data, it is possible to use KSQL to query and process data from the past, as well as new events that arrive on the topic. To tell KSQL to always process from beginning of topic run *SET 'auto.offset.reset' = 'earliest';*. Run this now, so that future processing includes all existing data.

4. **Creating a stream from a query**
   Let's take the poor ratings from people with iOS devices, and create a new stream from them!




```
ksql> CREATE STREAM POOR_RATINGS AS SELECT * FROM ratings WHERE STARS <3 AND CHANNEL='iOS';

 Message
----------------------------
 Stream created and running
----------------------------
```

What this does is set a KSQL continuous query running that processes messages on the source ratings topic to:

- applies the predicates (STARS < 3 AND CHANNEL='iOS')
- select just the specified columns (with * we are selecting all the columns)

Each processed message is written to a new Kafka topic. Remember, this is a continuous query, so every single source message—past, present, and future—will be processed with low-latency in this way. If we only want to process new messages and not existing ones, we would configure *SET 'auto.offset.reset' = 'latest';*.
Using DESCRIBE we can see that the new stream has the same columns as the source one.

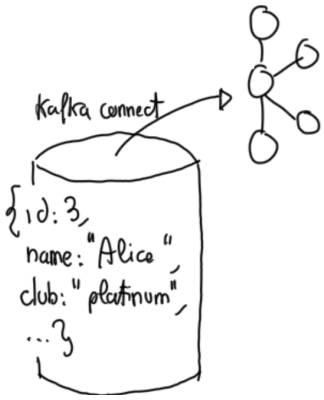
```
ksql> DESCRIBE POOR_RATINGS;
Name                 : POOR_RATINGS
 Field       | Type
-----------------------------------------
 ROWTIME     | BIGINT           (system)
 ROWKEY      | VARCHAR(STRING)  (system)
 RATING_ID   | BIGINT
 USER_ID     | INTEGER
 STARS       | INTEGER
 ROUTE_ID    | INTEGER
 RATING_TIME | BIGINT
 CHANNEL     | VARCHAR(STRING)
 MESSAGE     | VARCHAR(STRING)
-----------------------------------------
For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Table>;
ksql>
```

The query that we created above (CREATE STREAM POOR_RATINGS AS...) populates a Kafka topic, which we can also access as a KSQL stream (as in the previous step). Let's inspect this topic now, using KSQL.

```
ksql> SHOW TOPICS;

 Kafka Topic        | Registered | Partitions | Partition Replicas | Consumers | ConsumerGroups
-------------------------------------------------------------------------------------------------
[…]
 POOR_RATINGS       | true       | 4          | 1                  | 0         | 0
 ratings            | true       | 1          | 1                  | 1         | 1
-------------------------------------------------------------------------------------------------
ksql>
```

```
ksql> PRINT 'POOR_RATINGS';
Format:AVRO
6/20/18 11:01:03 AM UTC, 37, {"RATING_ID": 37, "USER_ID": 12, "STARS": 2, "ROUTE_ID": 8916, "RATING_TIME": 1529492463400,
6/20/18 11:01:07 AM UTC, 55, {"RATING_ID": 55, "USER_ID": 10, "STARS": 2, "ROUTE_ID": 5232, "RATING_TIME": 1529492467552,
```

5. **Integrating a database via Kafka Connect**



If you want to try a "complete" solution for integrating your database with Kafka, then log-based Change-Data-Capture (CDC) is the route to go. Done properly, CDC basically enables you to stream every single event from a database into Kafka. Broadly put, relational databases use a transaction log (also called a binlog or redo log depending on DB flavour), to which every event in the database is written. Update a row, insert a row, delete a row – it all goes to the database's transaction log. CDC tools generally work by utilising this transaction log to extract at very low latency and low impact the events that are occurring on the database (or a schema/table within it).

Many CDC tools exist, serving a broad range of sources. Some specialise in broad coverage of source systems, others in just specific ones. The common factor uniting most of them is close integration with Apache Kafka and Confluent Platform. Being able to stream your data from a database not only into Kafka, but with support for things such as the preservation of schemas through the Schema Registry, is a defining factor of these CDC tools. Some are built using the Kafka Connect framework itself (and tend to offer a richer degree of integration), whilst others use the Kafka Producer API in conjunction with support for the Schema Registry, etc.

CDC tools with support from the vendor and integration with Confluent Platform are (as of March 2018):

- Attunity Replicate
- Debezium
- IBM IIDR
- Oracle GoldenGate for Big Data
- SQ Data

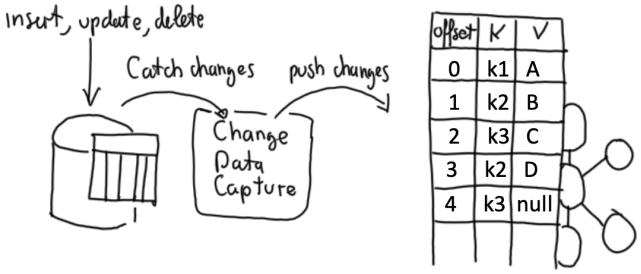


Figure 134: Log-based Change-Data-Capture integration.

Figure 134 shows how CDC works. In particular the data showed in the Kafka Commit Log (the table on the right) is the result of this flow of queries on the MySQL database:

(a) insert $K1 = A$, $K2 = B$, $K3 = C$

(b) update $K2 = D$

(c) delete $K3$

After having started MySQL and Debezium, we enter the MySQL command prompt and make some changes to the data.

```
docker-compose exec connect-debezium bash -c '/scripts/create-mysql-source.sh'
```

```
docker-compose exec mysql bash -c 'mysql -u $MYSQL_USER -p$MYSQL_PASSWORD demo'
```

```
INSERT INTO CUSTOMERS (ID,FIRST_NAME,LAST_NAME) VALUES (42,'Rick','Astley');
UPDATE CUSTOMERS SET FIRST_NAME = 'Thomas', LAST_NAME ='Smith' WHERE ID=2;
```

We should see each DML cause an almost-instantaneous update on the Kafka topic. For each change, inspect the output of the Kafka topic. Observe the difference between an INSERT and UPDATE.

Let's look at the customer data from the KSQL prompt. Here we use the FROM BEGINNING argument, which tells KSQL to go back to the beginning of the topic and show all data from there.

```
ksql> PRINT 'asgard.demo.CUSTOMERS' FROM BEGINNING;
Format:AVRO
3/4/19 5:50:42 PM UTC, Struct{id=1}, {"id": 1, "first_name": "Rica", "last_name": "Blaisdell", "email": "rblaisdell0@rambl
3/4/19 5:50:42 PM UTC, Struct{id=2}, {"id": 2, "first_name": "Ruthie", "last_name": "Brockherst", "email": "rbrockherst1@c
```

6. **From topics to tables**
   Since we're going to eventually join the customer data to the ratings, the customer Kafka messages must be keyed on the field on which we are performing the join. If this is not the case the join will fail and we'll get NULL values in the result. Our source customer messages are currently keyed using the Primary Key of the source table, but using a key serialisation that KSQL does not yet support—and thus in effect is not useful as a key in KSQL at all.

```
Format:AVRO
3/4/19 5:50:42 PM UTC, Struct{id=1}, {"id": 1, "first_name": "Rica", "last_name": "Blaisdell", "ema
3/4/19 5:50:42 PM UTC, Struct{id=2}, {"id": 2, "first_name": "Ruthie", "last_name": "Brockherst", "
```

```
Format:AVRO
22/02/18 12:55:04 GMT, 5312, {"rating_id": 5312, "user_id": 4, "stars": 4, "route_id": 2440, "rati
22/02/18 12:55:05 GMT, 5313, {"rating_id": 5313, "user_id": 3, "stars": 4, "route_id": 6975, "rati
```

To re-key a topic in Kafka we can use KSQL! First we will register the customer topic.

```
ksql> CREATE STREAM CUSTOMERS_SRC WITH (KAFKA_TOPIC='asgard.demo.CUSTOMERS', VALUE_FORMAT='AVRO');

 Message
----------------
 Stream created
----------------
ksql>
```

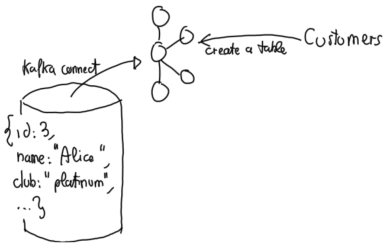
With the stream registered, we can now re-key the topic, using a KSQL CSAS (*Create Stream ... As*) and the PARTITION BY clause. Note that we're also changing the number of partitions from that of the source (4) to match that of the ratings topic (1).

```
SET 'auto.offset.reset' = 'earliest';

CREATE STREAM CUSTOMERS_SRC_REKEY
       WITH (PARTITIONS=1) AS
       SELECT * FROM CUSTOMERS_SRC PARTITION BY ID;
```

We can see that now the key is an integer (PRINT CUSTOMERS_SRC_REKEY FROM BEGINNING;).

```
Format:AVRO
3/4/19 5:50:42 PM UTC, 1, {"id": 1, "first_name": "Rica", "last_name": "Blaisdell", "ema
3/4/19 5:50:42 PM UTC, 2, {"id": 2, "first_name": "Ruthie", "last_name": "Brockherst", "
```

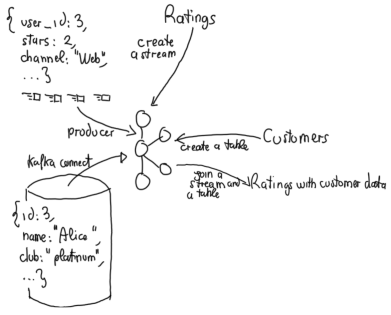
Finally, let's register a table over the new re-keyed topic. This because for each key (user id), we want to know its current value (name, status, etc).




```
ksql> CREATE TABLE CUSTOMERS WITH (KAFKA_TOPIC='CUSTOMERS_SRC_REKEY', VALUE_FORMAT ='AVRO', KEY='ID');

 Message
----------------
 Table created
----------------
ksql>
```

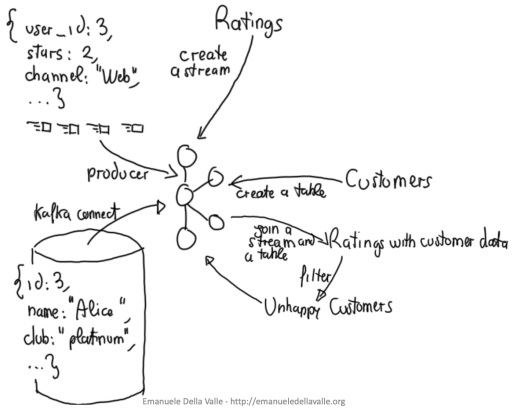
```
ksql> SELECT ID, FIRST_NAME, LAST_NAME, EMAIL, CLUB_STATUS FROM CUSTOMERS LIMIT 3;
1 | Rica | Blaisdell | rblaisdell0@rambler.ru | bronze
3 | Mariejeanne | Cocci | mcocci2@techcrunch.com | bronze
4 | Hashim | Rumke | hrumke3@sohu.com | platinum
Limit Reached
Query terminated
```

7. **Enriching each rating with customer data**
   Let's use the customer data (CUSTOMERS) and use it to enrich the inbound stream of ratings data (RATINGS) to show against each rating who the customer is, and their club status ('platinum','gold', etc).



```
CREATE STREAM RATINGS_WITH_CUSTOMER_DATA WITH (PARTITIONS=1) AS
SELECT R.RATING_ID, R.CHANNEL, R.STARS, R.MESSAGE,
       C.ID, C.CLUB_STATUS, C.EMAIL,
       C.FIRST_NAME, C.LAST_NAME
FROM RATINGS R
     INNER JOIN CUSTOMERS C
       ON R.USER_ID = C.ID ;
```

8. **Filtering an enriched stream**
   Having enriched the initial stream of ratings events with customer data, we can now persist a filtered version of that stream that includes a predicate to identify just those VIP customers who have left bad reviews.




```
CREATE STREAM UNHAPPY_PLATINUM_CUSTOMERS AS
SELECT CLUB_STATUS, EMAIL, STARS, MESSAGE
FROM   RATINGS_WITH_CUSTOMER_DATA
WHERE  STARS < 3
  AND  CLUB_STATUS = 'platinum';
```

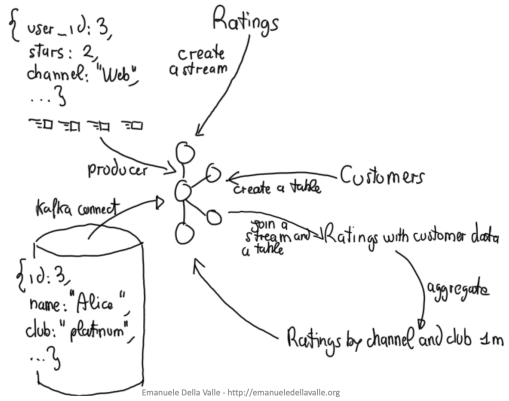
```
ksql> SELECT STARS, MESSAGE, EMAIL FROM UNHAPPY_PLATINUM_CUSTOMERS;
1 | is this as good as it gets? really ? | aarent0@cpanel.net
2 | airport refurb looks great, will fly outta here more! | aarent0@cpanel.net
2 | meh | aarent0@cpanel.net
```

9. **Streaming aggregates**
   KSQL can create aggregations of event data, either over all events to date (and continuing to update with new data), or based on a time window. The time window types supported are:

   - Tumbling (e.g. every 5 minutes : 00:00, 00:05, 00:10)
   - Hopping (e.g. every 5 minutes, advancing 1 minute: 00:00-00:05, 00:01-00:06)
   - Session (Sets a timeout for the given key, after which any new data is treated as a new session)

   To understand more about these time windows, you can read the related Kafka Streams documentation. Since KSQL is built on Kafka Streams, the concepts are the same. The KSQL-specific documentation is also useful. Aggregates can be persisted too. Instead of

CREATE STREAM as we did above, we're going to instead persist with a CREATE TABLE, since aggregates are always a table (key + value). Just as before though, a Kafka topic is continually populated with the results of the query.




```
CREATE TABLE RATINGS_BY_CLUB_STATUS AS
SELECT WindowStart() AS WINDOW_START_TS, CLUB_STATUS, COUNT(*) AS RATING_COUNT
FROM RATINGS_WITH_CUSTOMER_DATA
     WINDOW TUMBLING (SIZE 1 MINUTES)
GROUP BY CLUB_STATUS;
```

This table that we've created is just a first class object in KSQL, updated in real time with the results from the aggregate query. Because it's just another object in KSQL, we can query and filter it as any other.

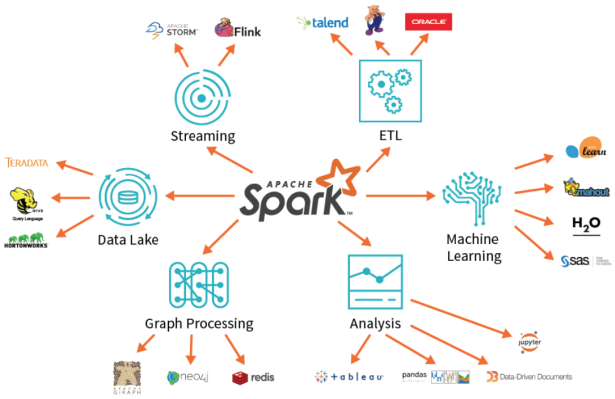
```
SELECT TIMESTAMPTOSTRING(WINDOW_START_TS, 'yyyy-MM-dd HH:mm:ss'),
       CLUB_STATUS, RATING_COUNT
FROM RATINGS_BY_CLUB_STATUS
WHERE CLUB_STATUS='bronze';
```

```
2019-03-04 17:42:00 | bronze | 2
2019-03-04 17:42:00 | bronze | 3
2019-03-04 17:42:00 | bronze | 5
2019-03-04 17:42:00 | bronze | 9
2019-03-04 17:42:00 | bronze | 10
```

If you let the SELECT output continue to run, you'll see all of the past time window aggregate values—but also the current one. Note that the *current* time window's aggregate value will continue to update, because new events are being continually processed and reflected in the value. If you were to send an event to the source ratings topic with a timestamp in the past, the corresponding time window's aggregate would be re-emitted.

11 Spark

11.1 Introduction



Apache Spark is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing.

Actually, today we have many free solutions for big data processing. Many companies also offer specialized enterprise features to complement the open-source platforms.

The trend started in 1999 with the development of Apache Lucene. The framework soon became open-source and led to the creation of Hadoop. Two of the most popular big data processing frameworks in use today are open source – **Apache Hadoop** and **Apache Spark**. Contents Introduction - Big Data Processing Today, we have many free solutions for big data processing. Many companies also offer specialized enterprise features to complement the open-source platforms.
The trend started in 1999 with the development of Apache Lucene. The framework soon became open-source and led to the creation of Hadoop. Two of the most popular big data processing frameworks in use today are open source – Apache Hadoop and Apache Spark.

There is always a question about which framework to use, Hadoop, or Spark.

Hadoop Apache Hadoop is a platform that handles large datasets in a distributed fashion. The framework uses MapReduce to split the data into blocks and assign the chunks to nodes across a cluster. MapReduce then processes the data in parallel on each node to produce a unique output.

Every machine in a cluster both stores and processes data. **Hadoop stores the data to disks using HDFS (disk-centrik)**. The software offers seamless scalability options. You can start with as low as one machine and then expand to thousands, adding any type of enterprise or commodity hardware.

The Hadoop ecosystem is highly fault-tolerant. Hadoop does not depend on hardware to achieve high availability. At its core, Hadoop is built to look for failures at the application layer. By replicating data across a cluster, when a piece of hardware fails, the framework can build the missing parts from another location.

HDFS: Hadoop Distributed File System.
This is the file system that manages the storage of large sets of data across a Hadoop cluster. HDFS can handle both structured and unstructured data. The storage hardware can range from any consumer-grade HDDs to enterprise drives.

Spark Apache Spark is an open-source tool. This framework can run in a standalone mode or on a cloud or cluster manager such as Apache Mesos, and other platforms. **It is designed for fast performance and uses RAM for caching and processing data**.

Spark performs different types of big data workloads. This includes MapReduce-like batch processing, as well as real-time stream processing, machine learning, graph computation, and interactive queries. With easy to use high-level APIs, Spark can integrate with many different libraries, including PyTorch and TensorFlow.

The Spark engine was created to improve the efficiency of MapReduce and keep its benefits. Even though Spark does not have its file system, it can access data on many different storage solutions. The data structure that Spark uses is called Resilient Distributed Dataset, or RDD.

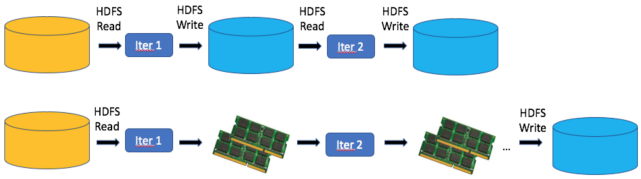


Figure 135: Spark (RAM-centric) vs Hadoop (disk-centric)

11.1.1 Spark APIs

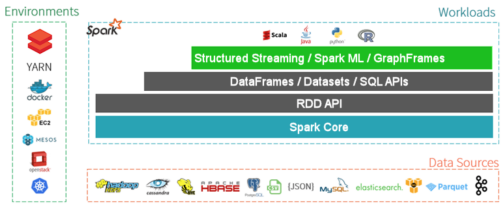


Figure 136: The architecture of Spark

Apache Spark has a well-defined layered architecture where all the spark components and layers are loosely coupled. This architecture is further integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions:

- **Resilient Distributed Dataset (RDD)**
- **Directed Acyclic Graph (DAG)**

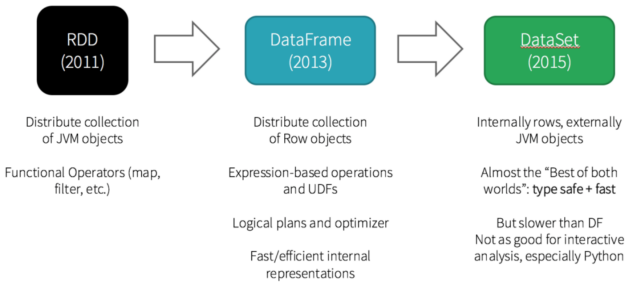


Figure 137: History of Spark APIs

RDD RDDs are the building blocks of any Spark application. RDD stands for:

- **Resilient**: Fault tolerant and is capable of rebuilding data on failure
- **Distributed:** Distributed data among the multiple nodes in a cluster
- **Dataset:** Collection of partitioned data with values

At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.

There are two basic operations that can be done on RDDs. They are **transformations** and **actions**.

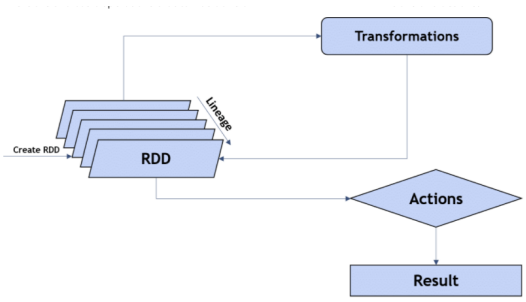


Figure 138: Spark RDD operations

Transformations: These are functions that accept the existing RDDs as input and outputs one or more RDDs. However, the data in the existing RDD in Spark does not change as it is immutable. Some of the transformation operations are provided in the table below:

Function	Description
map()	Returns a new RDD by applying the function on each data element
filter()	Returns a new RDD formed by selecting those elements of the source on which the function returns true
reduceByKey()	Aggregates the values of a key using a function
groupByKey()	Converts a (key, value) pair into a (key, <iterable value>) pair
union()	Returns a new RDD that contains all elements and arguments from the source RDD
intersection()	Returns a new RDD that contains an intersection of the elements in the datasets

These transformations are executed when they are invoked or called. Every time transformations are applied, a new RDD is created.

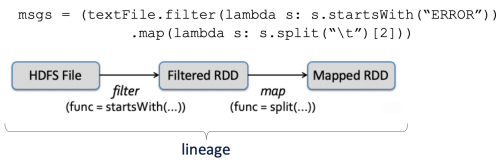


Figure 139: **Transformations - logical view.** Since RDD are immutable, transformations create new RDD.

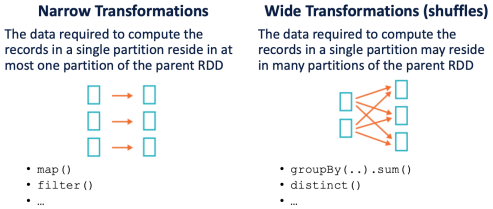


Figure 140: **Transformations - physical view.**

Actions: Actions in Spark are functions that return the end result of RDD computations. It uses a lineage graph to load data onto the RDD in a particular order. After all of the transformations are done, actions return the final result to the Spark Driver. Actions are operations that provide non-RDD values. Some of the common actions used in Spark are given below:

Function	Description
count()	Gets the number of data elements in an RDD
collect()	Gets all the data elements in an RDD as an array
reduce()	Aggregates data elements into an RDD by taking two arguments and returning one
take(n)	Fetches the first *n* elements of an RDD
foreach(operation)	Executes the **operation** for each data element in an RDD
first()	Retrieves the first data element of an RDD

Whenever an action is called, the driver delegates tasks to workers, who then in turn return the answer to the driver whenever their execution is completed.

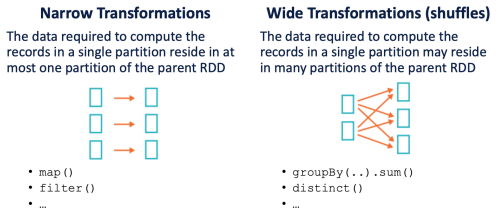


Figure 141: **Actions - system view.**

Fundamental to Apache Spark are the notions that:

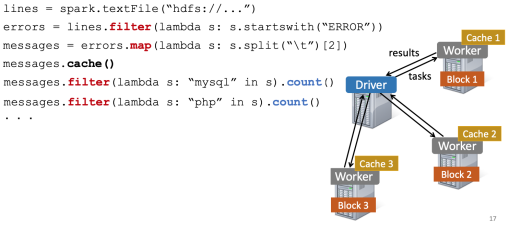
- Transformations are **LAZY**
- Actions are **EAGER**

This means that, we can run multiple transformations back-to-back, and no job is triggered until the first action is requested.

Lazyness is a common pattern in functional programming and Big Data languages (Scala core design, Java 8 Streams API, Hive and Pig). As said, it consists in the system collecting tasks and then do the best to optimize the performance of execution.
Benefits:

- Not forced to load all data at 1^{st} step: technically impossible with large datasets
- Easier to parallelize operations: N different transformations can be processed on a single data element, on a single thread, on a single machine
- Allows optimizations

With RDD we can also cache results and reuse them without recomputing the partial transformations.



DataFrame Like an RDD, a DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction; it provides a domain specific language API to manipulate your distributed data; and makes Spark accessible to a wider audience, beyond specialized data engineers.

```
# Read a CSV
userDF = spark.read.csv(" … /userData.csv")
# ... or Use DataFrame APIs and register a temp view
middleageSmokers = userDF.filter(col("smoker")=="N").filter(col("age")>40)
middleageSmokers.createOrReplaceTempView("middleageSmokers")
# ... read a CSV and register a temp view
spark.read.json(" … /part-00000.json.gz").createOrReplaceTempView("iot_stream")
# ... execute SQL query or ...
spark.sql("SELECT avg(calories_burnt) FROM iot_stream JOIN middleageSmokers ON …")
```

Figure 142: Spark DataFrame snippet

Because DataFrame API is declarative, a large number of optimizations are possible:

- Optimizing data types for storage
- Rewriting queries for performance
- Predicate push downs

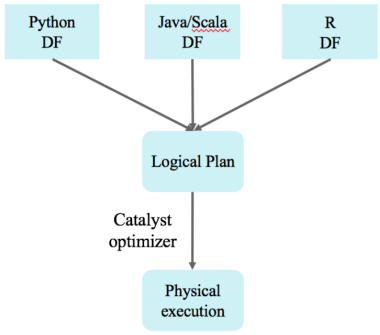


Figure 143: Spark DataFrame execution

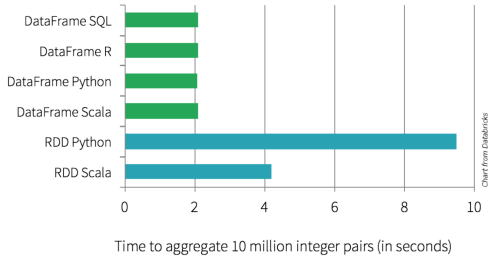


Figure 144: Performance - RDD vs. DataFrame

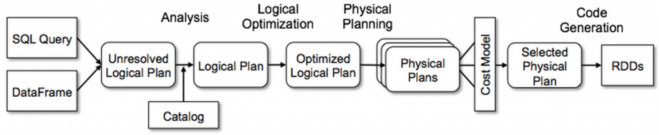


Figure 145: **Under the hood: Catalyst.** Generates optimal RDD code from the user request.

Dataset APIs In Spark 2.0, DataFrame APIs will merge with Datasets APIs, unifying data processing capabilities across libraries. Because of this unification, developers now have fewer concepts to learn or remember, and work with a single high-level and type-safe API called Dataset.

Indeed,sStarting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a *strongly-typed* API and an *untyped* API, as shown in the table below. Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic untyped JVM object. Dataset, by contrast, is a collection of strongly-typed JVM objects, dictated by a case class you define in Scala or a class in Java.

Language	Main Abstraction
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python*	DataFrame
R*	DataFrame

Figure 146: Since Python and R have no compile-time type-safety, we only have untyped APIs, namely DataFrames.

As a Spark developer, you benefit with the DataFrame and Dataset unified APIs in Spark 2.0 in a number of ways. The most relevant benefit is known as **Static-typing and runtime type-safety**.

Consider static-typing and runtime safety as a spectrum, with SQL least restrictive to Dataset most restrictive. For instance, in your Spark SQL string queries, you won't know a syntax error until runtime (which could be costly), whereas in DataFrames and Datasets you can catch errors at compile time (which saves developer-time and costs). That is, if you invoke a function in DataFrame that is not part of the API, the compiler will catch it. However, it won't detect a non-existing column name until runtime.

At the far end of the spectrum is Dataset, most restrictive. Since Dataset APIs are all expressed as lambda functions and JVM typed objects, any mismatch of typed-parameters will be detected at compile time. Also, your analysis error can be detected at compile time too, when using Datasets, hence saving developer-time and costs.

All this translates to is a spectrum of type-safety along syntax and analysis error in your Spark code, with Datasets as most restrictive yet productive for a developer.

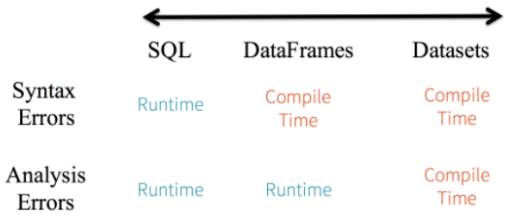


Figure 147: Error Discovery - DataFrame vs Dataset vs SQL

11.1.2 Spark at Work

DAG Scheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. It transforms a logical execution plan (i.e. RDD lineage of dependencies built using RDD transformations) to a physical one (a **Job)**) considering data locality and partitioning (to avoid shuffles). Each Job is a DAG of Stages which are broken down into Tasks.

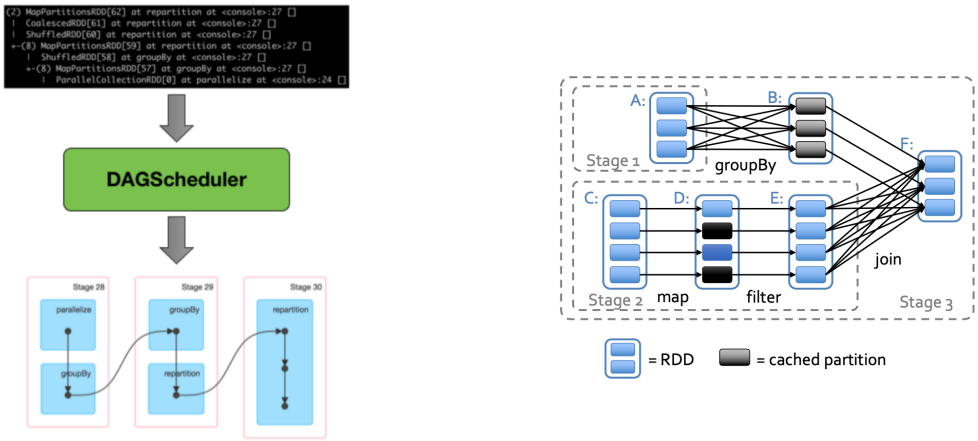
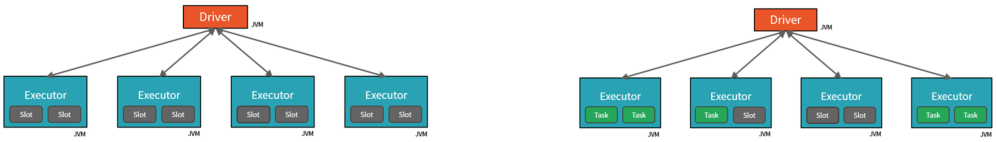


Figure 148: DAG Scheduler transforming RDD Lineage into stage DAG.

A Spark cluster includes:

- One **Driver** running the user application
- Multiple **Executors** (typically, one instance per node) with multiple **Slots** (typically, one per core/CPU)



The **Driver** assigns **Tasks** to Executors' **Slots** for parallel execution. The Driver also makes sure that all tasks of a Stage are executed before starting any task of the next stage

11.2 Spark Structured Streaming

11.2.1 Overview

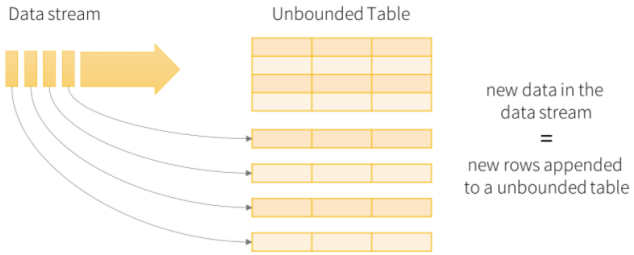
Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive. Developers can use the Dataset/DataFrame API in Scala, Java, Python or R to express streaming aggregations, event-time windows, stream-to-batch joins, etc. The computation is executed on the same optimized Spark SQL engine. Finally, the system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs. In short, Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.

Internally, by default, Structured Streaming queries are processed using a **micro-batch processing engine**, which processes data streams as a series of small batch jobs thereby achieving end-to-end latencies as low as 100 milliseconds and exactly-once fault-tolerance guarantees. However, since Spark 2.3, a new low-latency processing mode called **Continuous Processing** was introduced, which can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees. Without changing the Dataset/DataFrame operations in your queries, developers will be able to choose the mode based on your application requirements.

11.2.2 Programming Model

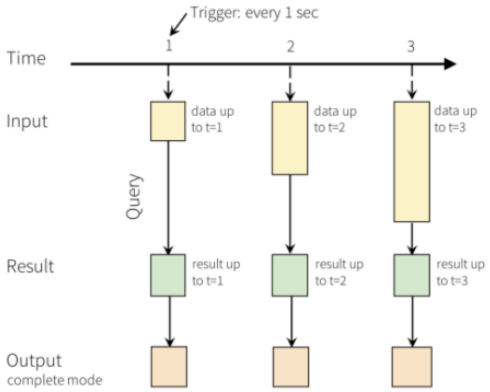
The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended. This leads to a new stream processing model that is very similar to a batch processing model. You will express your streaming computation as standard batch-like query as on a static table, and Spark runs it as an incremental query on the *unbounded input table*.

Basic Concepts Consider the input data stream as the "Input Table". Every data item that is arriving on the stream is like a new row being appended to the Input Table.



Data stream as an unbounded table

A query on the input will generate the "Result Table". Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table. Whenever the result table gets updated, we would want to write the changed result rows to an external sink.



Programming Model for Structured Streaming

The "Output" is defined as what gets written out to the external storage. The output can be defined in a different mode:

- **Complete Mode** - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.
- **Append Mode** - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.
- **Update Mode** - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage. Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger. If the query doesn't contain aggregations, it will be equivalent to Append mode.

Note that Structured Streaming does not materialize the entire table. It reads the latest available data from the streaming data source, processes it incrementally to update the

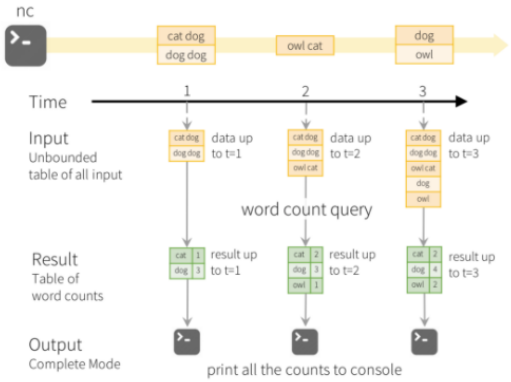


Figure 149: Example of Complete Mode.

result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result (e.g. intermediate counts in the earlier example).

This model is significantly different from many other stream processing engines. Many streaming systems require the user to maintain running aggregations themselves, thus having to reason about fault-tolerance, and data consistency (at-least-once, or at-most-once, or exactly-once). In this model, Spark is responsible for updating the Result Table when there is new data, thus relieving the users from reasoning about it. As an example, let's see how this model handles event-time based processing and late arriving data.

Handling Event-time and Late Data Event-time is the time embedded in the data itself. For many applications, you may want to operate on this event-time. For example, if you want to get the number of events generated by IoT devices every minute, then you probably want to use the time when the data was generated (that is, event-time in the data), rather than the time Spark receives them. This event-time is very naturally expressed in this model – each event from the devices is a row in the table, and event-time is a column value in the row. This allows window-based aggregations (e.g. number of events every minute) to be just a special type of grouping and aggregation on the event-time column – each time window is a group and each row can belong to multiple windows/groups. Therefore, such event-time-window-based aggregation queries can be defined consistently on both a static dataset (e.g. from collected device events logs) as well as on a data stream, making the life of the user much easier.

Furthermore, this model naturally handles data that has arrived later than expected based on its event-time. Since Spark is updating the Result Table, it has full control over updating old aggregates when there is late data, as well as cleaning up old aggregates to limit the size of intermediate state data. Since Spark 2.1, we have support for watermarking which allows the user to specify the threshold of late data, and allows the engine to accordingly clean up old state.

11.2.3 API using Datasets and DataFrames

Since Spark 2.0, DataFrames and Datasets can represent static, bounded data, as well as streaming, unbounded data. Similar to static Datasets/DataFrames, you can use the common entry point SparkSession (Scala/Java/Python/R docs) to create streaming DataFrames/Datasets from streaming sources, and apply the same operations on them as static DataFrames/Datasets.

Streaming DataFrames can be created through the *DataStreamReader* interface (Scala/Java/Python docs) returned by *SparkSession.readStream()*. Similar to the read interface for creating static DataFrame, you can specify the details of the source – data format, schema, options, etc.

Input Sources There are a few built-in sources:

- **File source** - Reads files written in a directory as a stream of data. Files will be processed in the order of file modification time. Supported file formats are text, CSV, JSON, ORC,

Parquet.

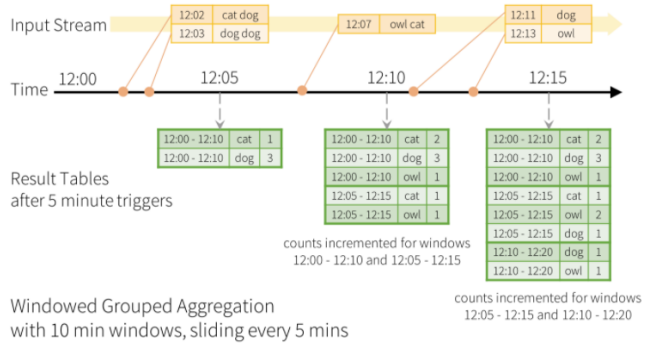
- **Kafka source** - Reads data from Kafka.
- **Socket source (for testing)** - Reads UTF8 text data from a socket connection. The listening server socket is at the driver. Note that this should be used only for testing as this does not provide end-to-end fault-tolerance guarantees.

```
{% highlight python %}
spark = SparkSession. ...

# Read text from socket
socketDF = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

socketDF.isStreaming()    # Returns True for DataFrames that have streaming sources

socketDF.printSchema()

# Read all the csv files written atomically in a directory
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark \
    .readStream \
    .option("sep", ";") \
    .schema(userSchema) \
    .csv("/path/to/directory")  # Equivalent to format("csv").load("/path/to/directory")



        df = ...

        # Select the devices which have signal more than 10
        df.select("device").where("signal > 10")

        # Running count of the number of updates for each device type
        df.groupBy("deviceType").count()
```

Figure 150: e.g., Python

Window Operations on Event Time Aggregations over a sliding event-time window are straightforward with Structured Streaming and are very similar to grouped aggregations. In a grouped aggregation, aggregate values (e.g. counts) are maintained for each unique value in the user-specified grouping column. In case of window-based aggregations, aggregate values are maintained for each window the event-time of a row falls into.

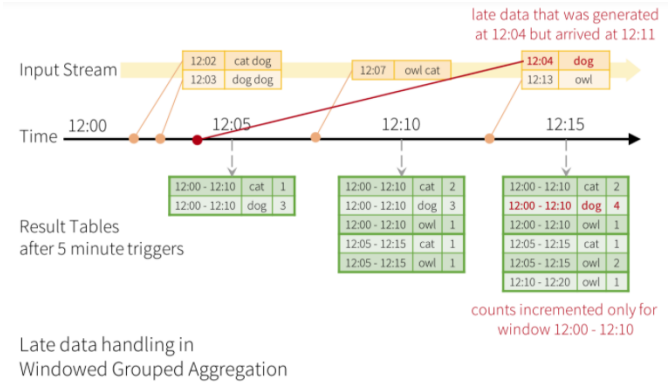
Imagine that the example of Figure 149 is modified and the stream now contains lines along with the time when the line was generated. Instead of running word counts, we want to count words within 10 minute windows, updating every 5 minutes. That is, word counts in words received between 10 minute windows 12:00 - 12:10, 12:05 - 12:15, 12:10 - 12:20, etc. Note that 12:00 - 12:10 means data that arrived after 12:00 but before 12:10. Now, consider a word that was received at 12:07. This word should increment the counts corresponding to two windows 12:00 - 12:10 and 12:05 - 12:15. So the counts will be indexed by both, the grouping key (i.e. the word) and the window (can be calculated from the event-time).




```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```

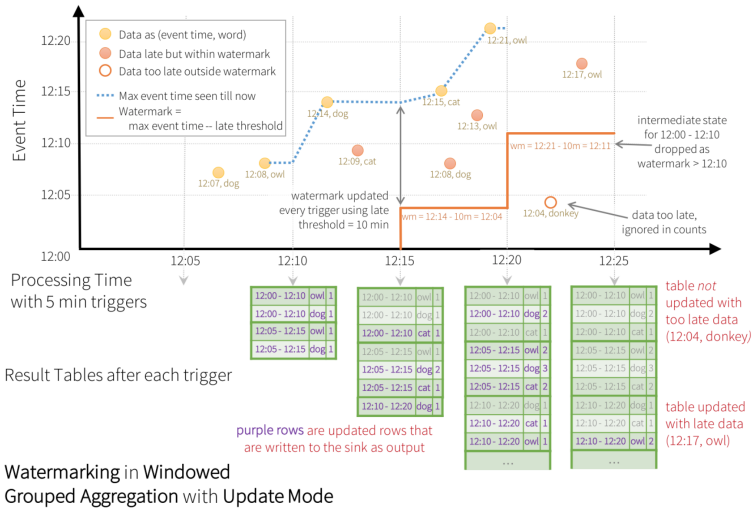
Figure 151: Spark Structured Streaming treats this type of windows as grouping clauses.

Handling Late Data and Watermarking Now consider what happens if one of the events arrives late to the application. For example, say, a word generated at 12:04 (i.e. event time) could be received by the application at 12:11. The application should use the time 12:04 instead of 12:11 to update the older counts for the window 12:00 - 12:10. This occurs naturally in our window-based grouping – Structured Streaming can maintain the intermediate state for partial aggregates for a long period of time such that late data can update aggregates of old windows correctly, as illustrated below.




```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) \
    .count()
```

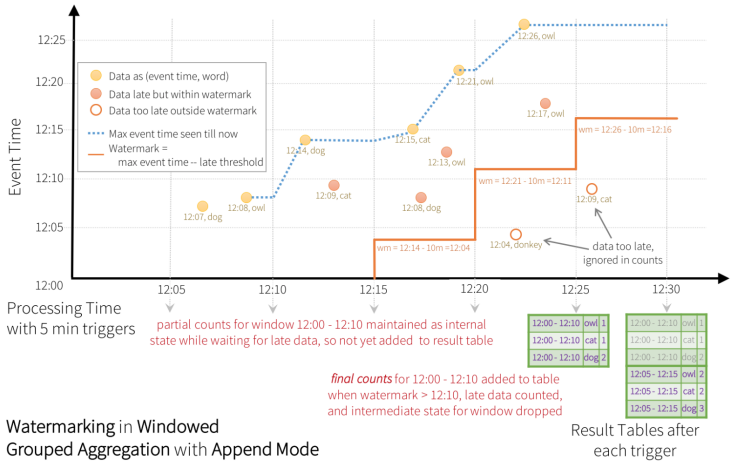
However, to run this query for days, it's necessary for the system to bound the amount of intermediate in-memory state it accumulates. This means the system needs to know when an old aggregate can be dropped from the in-memory state because the application is not going to receive late data for that aggregate any more. To enable this, in Spark 2.1, **watermarking** has been introduced, which lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly. You can define the watermark of a query by specifying the event time column and the threshold on how late the data is expected to be in terms of event time. For a specific window ending at time T, the engine will maintain state and allow late data to update the state until (*max event time seen by the engine - late threshold* $>$ *T*). In other words, late data within the threshold will be aggregated, but data later than the threshold will start getting dropped (see later in the section for the exact guarantees).



As shown in the illustration, the maximum event time tracked by the engine is the *blue dashed line*, and the watermark set as (max event time - '10 mins') at the beginning of every trigger is the *red line*. For example, when the engine observes the data (12:14, dog), it sets the watermark

for the next trigger as 12:04. This watermark lets the engine maintain intermediate state for additional 10 minutes to allow late data to be counted. For example, the data (12:09, cat) is out of order and late, and it falls in windows 12:00 - 12:10 and 12:05 - 12:15. Since, it is still ahead of the watermark 12:04 in the trigger, the engine still maintains the intermediate counts as state and correctly updates the counts of the related windows. However, when the watermark is updated to 12:11, the intermediate state for window (12:00 - 12:10) is cleared, and all subsequent data (e.g. (12:04, donkey)) is considered "too late" and therefore ignored. Note that after every trigger, the updated counts (i.e. purple rows) are written to sink as the trigger output, as dictated by the Update mode.

Some sinks (e.g. files) may not supported fine-grained updates that Update Mode requires. To work with them, we there is also the support for Append Mode, where only the final counts are written to sink. This is illustrated below.



Similar to the Update Mode earlier, the engine maintains intermediate counts for each window. However, the partial counts are not updated to the Result Table and not written to sink. The engine waits for "10 mins" for late date to be counted, then drops intermediate state of a $window < watermark$, and appends the final counts to the Result Table/sink. For example, the final counts of window 12:00 - 12:10 is appended to the Result Table only after the watermark is updated to 12:11.

Semantic Guarantees of Aggregation with Watermarking:

- A watermark delay (set with $withWatermark$) of "2 hours" guarantees that the engine will never drop any data that is less than 2 hours delayed. In other words, any data less than 2 hours behind (in terms of event-time) the latest data processed till then is guaranteed to be aggregated.
- However, the guarantee is strict only in one direction. Data delayed by more than 2 hours is not guaranteed to be dropped; it may or may not get aggregated. More delayed is the data, less likely is the engine going to process it.

Join Operations Structured Streaming supports joining a streaming Dataset/DataFrame with a static Dataset/DataFrame as well as another streaming Dataset/DataFrame. The result of the streaming join is generated incrementally, similar to the results of streaming aggregations in the previous section. In this section we will explore what type of joins (i.e. inner, outer, etc.) are supported in the above cases. Note that in all the supported join types, the result of the join with a streaming Dataset/DataFrame will be the exactly the same as if it was with a static Dataset/DataFrame containing the same data in the stream.

```
staticDf = spark.read. ...
streamingDf = spark.readStream. ...
streamingDf.join(staticDf, "type")
streamingDf.join(staticDf, "type", "left_outer")
```

Notes:

- INNER and LEFT-OUTER stream-static joins are not stateful
- OUTER and RIGHT-OUTER stream-static joins are not supported

In Spark 2.3, the support for **stream-stream joins** has been added, that is, you can join two streaming Datasets/DataFrames. The challenge of generating join results between two data streams is that, at any point of time, the view of the dataset is incomplete for both sides of the join making it much harder to find matches between inputs. Any row received from one input stream can match with any future, yet-to-be-received row from the other input stream. Hence, for both the input streams, we buffer past input as streaming state, so that we can match every future input with past input and accordingly generate joined results. Furthermore, similar to streaming aggregations, we automatically handle late, out-of-order data and can limit the state using watermarks.

The solution to this problem are **windows** (but not the kind you can express as a group by clause):

- buffer past input as streaming state
- handle late, out-of-order data using watermarks

Inner joins on any kind of columns along with any kind of join conditions are supported. However, as the stream runs, the size of streaming state will keep growing indefinitely as all past input must be saved as any new input can match with any input from the past. To avoid unbounded state, you have to define additional join conditions such that indefinitely old inputs cannot match with future inputs and therefore can be cleared from the state. In other words, you will have to do the following additional steps in the join.

- Define watermark delays on both inputs such that the engine knows how delayed the input can be (similar to streaming aggregations)
- Define a constraint on event-time across the two inputs such that the engine can figure out when old rows of one input is not going to be required (i.e. will not satisfy the time constraint) for matches with the other input. This constraint can be defined in one of the two ways.
  - Time range join conditions (e.g. *...JOIN ON leftTime BETWEEN rightTime AND rightTime + INTERVAL 1 HOUR*),
  - Join on event-time windows (e.g. *...JOIN ON leftTimeWindow = rightTimeWindow*).

Let's understand this with an example.

Let's say we want to join a stream of advertisement impressions (when an ad was shown) with another stream of user clicks on advertisements to correlate when impressions led to monetizable clicks. To allow the state cleanup in this stream-stream join, you will have to specify the watermarking delays and the time constraints as follows.

- Watermark delays: Say, the impressions and the corresponding clicks can be late/out-of-order in event-time by at most 2 and 3 hours, respectively.
- Event-time range condition: Say, a click can occur within a time range of 0 seconds to 1 hour after the corresponding impression.

```
from pyspark.sql.functions import expr

impressions = spark.readStream. ...
clicks = spark.readStream. ...

# Apply watermarks on event-time columns
impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")

# Join with event-time constraints
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """)
)
```

Figure 152: Inner Joins with optional Watermarking - Code example

Join Type	
Inner	Supported, optionally specify watermark on both sides + time constraints for state cleanup
Left Outer	Conditionally supported, must specify watermark on right + time constraints for correct results, optionally specify watermark on left for all state cleanup
Right Outer	Conditionally supported, must specify watermark on left + time constraints for correct results, optionally specify watermark on right for all state cleanup
Full Outer	Not supported

Figure 153: Support table for joins in streaming queries

Starting Streaming Queries Once you have defined the final result DataFrame/Dataset, all that is left is for you to start the streaming computation. To do that, you have to use the *DataStreamWriter* (Scala/Java/Python docs) returned through *Dataset.writeStream()*. You will have to specify one or more of the following in this interface.

- **Details of the output sink**: Data format, location, etc.
- **Output mode**: Specify what gets written to the output sink.
- **Query name**: Optionally, specify a unique name of the query for identification.
- **Trigger interval**: Optionally, specify the trigger interval.
- **Checkpoint location**: For some output sinks where the end-to-end fault-tolerance can be guaranteed, specify the location where the system will write all the checkpoint information. This should be a directory in an HDFS-compatible fault-tolerant file system.

Output sinks

- **File sink** - a directory
- **Kafka sink** - one or more topics in Kafka
- **Foreach sink** - Runs arbitrary computation
- **Console sink (for debugging)**
- **Memory sink (for debugging)**

```
# get the query object

query = df.writeStream.format("console").start()

# get the unique identifier of the running query
# that persists across restarts from checkpoint data

query.id()

# get the unique id of this run of the query, which
# will be generated at every start/restart

query.runId()

# get the name of the auto-generated or user-specified name

query.name()

# print detailed explanations of the query

query.explain()


# stop the query

query.stop()

# block until query is terminated, with stop() or with error
query.awaitTermination()

# the exception if the query has been terminated with error

query.exception()

# an array of the most recent progress updates for this query

query.recentProgress()

# the most recent progress update of this streaming query

query.lastProgress()
```

Figure 154: Managing Streaming Queries

12 Data Acquisition

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
- Kafka Basics
- Kafka and Zookeeper
- Avro and Schema Registry
- Kafka Stream and Tables
- Kafka Streams Concepts
- KSQL
- Hadoop vs. Spark
- Spark Architecture
- Spark APIs
- Spark RDDs