POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Computer Science and Engineering

Master Degree's Thesis

Towards a Unifying Modeling Framework for Data-Intensive Tools

Supervisor
Chiar.mo Prof. Alessandro Margara

Co-supervisor
Chiar.mo Prof. Gianpaolo Cugola

Candidate
Stefano Cilloni
880924

Academic Year 2018–2019

Abstract

In the past two decades, the volume of data collected and processed by computer systems has grown exponentially. The databases, employed initially to store and query data, are now inappropriate to address big data-related issues. They are indeed limited in horizontal scalability due to architectures that were conceived for single machines. The new big data requirements such as the large volume, the high velocity, and heterogeneity of formats have led to the development of a new tools' family defined as data-intensive.

With the advent of these technologies, new computational paradigms were introduced. MapReduce was one of the first and brought the idea to send the processing towards the data kept stored on multiple nodes. Later, another technique proposed the concept of stream of data by keeping the computation statically allocated and making the information flow through the system. With this approach, it is no longer necessary to store the data before it is processed.

Today's big data requirements are to transform raw data inputs into valuable knowledge, but also they ask for database-like capabilities such as state management and transactional guarantees. In this scenario, the design patterns of centralized databases from the '80s would impose a very limiting trade-off between strong guarantees and scalability.

The NoSQL approach is the first attempt made to address the issue. This category of systems is able to scale, but it provides neither strong guarantees nor a standardized SQL interface. Subsequently, the strong demand to exploit transactional guarantees again and to reintroduce the SQL language at scale motivated the origin of NewSQLs: databases with the legacy relational model but able to provide guarantees and also largely scale horizontally.

In this work, we studied data-intensive tools to compare functioning paradigms and new approaches to strong guarantees in distributed scenarios. We analyzed the differences and common aspects that are emerging and could drive future design patterns. We developed a modeling framework to conduct our analysis systematically. The proposed models could be exploited in next-generation systems' design to leverage established and emerging techniques in today's data-intensive tools to store, query, and analyze data.

Sommario

Negli ultimi due decenni, il volume di dati raccolti ed elaborati dai sistemi informatici è cresciuto esponenzialmente. I database, originariamente impiegati per archiviare e interrogare banche dati, risultano ora inappropriati nel risolvere problemi relativi ai big data. Essi sono infatti limitati nella scalabilità orizzontale a causa di architetture concepite per una singola macchina. I nuovi requisiti relativi ai big data come il grande volume, l'alta velocità e l'eterogeneità dei formati hanno portato allo sviluppo di una nuova famiglia di strumenti definita ad alta intensità di dati.

Con l'avvento di queste tecnologie sono stati introdotti nuovi paradigmi computazionali. Uno tra i primi, Map-Reduce, ha introdotto l'idea di inviare l'elaborazione verso i dati salvati su più nodi. Successivamente, un'altra tecnica ha proposto il concetto di flusso di dati mantenendo il calcolo allocato staticamente e facendo fluire l'infromazione attraverso il sistema. Tramite quest'ultimo approccio non è più necessario salvare i dati prima della loro elaborazione.

I requisiti odierni rispetto alla processazione di big data sono sia di trasformare dati grezzi in preziose conoscenze ma anche di poter utilizzare funzionalità simili ai database come la gestione di uno stato e il supporto a garanzie transazionali. In questo scenario, i modelli di progettazione dei database centralizzati degli anni '80 imporrebbero un compromesso fortemente limitativo tra garanzie e scalabilità.

L'approccio NoSQL è stato il primo tentativo fatto nel risolvere il problema. Questa categoria di sistemi ha la capacità di scalare ma non fornisce né garanzie transazionali né un'interfaccia SQL standard. Successivamente, la forte richiesta di sfruttare nuovamente le garanzie forti e di reintrodurre il linguaggio SQL su larga scala ha motivato l'origine dei NewSQL: database con il modello relazionale classico ma in grado di fornire garanzie e anche scalare orizzontalmente.

In questo lavoro di tesi abbiamo studiato gli strumenti ad alta intensità di dati per confrontare paradigmi di funzionamento e nuovi approcci introdotti a supporto di garanzie "forti" in scenari distribuiti. Abbiamo analizzato le differenze e gli aspetti comuni che stanno emergendo e che potrebbero guidare i modelli di progettazione futuri. Abbiamo sviluppato un framework di modellazione per condurre sistematicamente la nostra analisi. I modelli proposti potrebbero essere impiegati nella progettazione dei sistemi di prossima generazione, sfruttando le tecniche affermatesi ed emerse negli strumenti ad alta intesità di dati odierni per salvare, eseguire queries e analizzare dati.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database management systems are software that offer data storage and query features. They were created with the objective to save structured data and simplify information management to applications. Their birth in the 80s was driven by the necessity to provide logical abstractions and guarantees in saving and reading data. One of their major limitations is the assumption that was made for their design, a centralized architecture hence limited to the resources of a single computer. Over the past two decades, the amount of data handled and collected by computer systems has grown exponentially, and databases have quickly become inappropriate tools for storing these giant volumes of information. Alongside this massive data generation, it also arose the need to extract valuable information so to exploit them to answer analytical questions. By this necessity, a new class of data processing tools born and broke away from the one of pure data storage. A milestone in the dawn of data processing techniques was the introduction in 2008 of the Map-Reduce programming model from Google [DG04]. Its core idea is to keep data stored on cluster nodes and perform the computation by sending identical copies of processing, the *map* functions, towards them. In this way, the transformations can be effectively performed in parallel and, later, the partial results can be collected and aggregated through *reduce* operations. This computational paradigm influenced the design of analytical engines in the following years and it was a breakthrough in processing decentralization.

A further evolution to this initial concept led to conceive another computational paradigm called stream processing. In this opposite approach data continuously move through the system and by undergoing continuous conversions it is finally transformed into results. The transformations, that in Map-Reduce were sent towards the data, in this processing technique are stationary. They are “statically” kept on the nodes and continuously process new data as soon as there is any available.

Both these two types of data-intensive tools are used to balance and distribute the workload across all the available processing units. However, the concept of data stream is also extended to batch. The difference with respect to streams is that in the latter transformations happen over each datum. A single item can be potentially processed from the beginning to the end without waiting time. Instead, in the batch case, more elements are processed in parallel and at the same time. The concurrent processing of multiple items come from the willing to waste less computational and memory resources in favor of a larger amount of data processed despite a bit higher latency times. Both the streaming and batch processing outlined a new class of analytics systems characterized by the shift to distributed scenarios with the compromise to do not provide transactional guarantees at all; a development path opposed to the databases’ one.

Databases’ design patterns from the ’80s imposed rigid trade-off between strong guarantees and high performance due to employed protocols and coordination strategies. However, even this class of system evolved over time in the attempt to provide horizontal scalability. The NoSQL, a new databases' class, derived by the need to be able to scale-out data storage systems employing distributed approaches. They do not support transactions and also have weaker isolation levels but provide basically unlimited storage capabilities by leveraging all the available resources. This new conception however had the strong limitation of not supporting the standardized SQL interface for querying and storing the data. Moreover, software architectures employing these databases became more complex to cope with the weaker guarantees. They use graphical-interfaces strategies to mask propagation times that could cause the view of inconsistent states. All this complexity moved to the application architecture has spurred the birth of another database model, the NewSQL. This class of relational database management systems has the primary objective to provide the scalability of NoSQL but alongside with all the features of classical databases such as the support to online transaction processing (OLTP) and the enforcement of ACID guarantees. The NewSQL paradigm conception has proved how, through new paradigms, was possible to go beyond the original misleading trade-off of classical locking-based approaches.

The proposals in the literature of new definitions for transactional isolation levels [ALO00] and anomaly phenomena [BBG$^+$95] in concurrent executions enable the functioning of NewSQL databases and also are guiding stream processors towards transactional guarantees' support in the big data processing.

The new streaming paradigms, combined with the demand to provide strong guarantees in big data scenarios, are critical factors to the development of next-generation platforms. The modern developing models are focusing on merging the best characteristics of both stream processors and databases' worlds. Future systems conceived on these ideas would probably be able to enforce strong transactional guarantees and at the same time provide high execution throughput thanks to coordination-free isolation mechanisms.

The study conducted in this thesis has the objective to understand and highlight what the similarities and differences of existing data processing and storage systems are. For this purpose, we design a modeling framework to guide our study with a systematic and precise approach. The analysis will take into account systems' computational paradigms, the functionalities they provide, and guarantees given for concurrent operations' execution. Furthermore, we will consider the guarantees related to distributed environments such as those involved by communication through messages and fault-tolerance. Lastly, the proposed models want to suggest a possible future convergence of the two application domains originally born with diametrically opposite characteristics.

In chapter 2 – Background and Motivations – we will describe the context of this work and the ideas that conducted the research. With chapter 3 – The Modeling Framework – we will present and describe the models built in order to carry out systems' analysis. Later, in chapter 4 – Methodology – the conducted experiments will be reported and explained alongside the discovered systems' limitations. In chapter 5 –Taxonomy– the analyzed data-intensive tools will be characterized and described using the presented models. Later, with the chapter 6 –Discussion– we will report our results and we will argue about the model fitting to the platforms. Finally, with chapter 7 –Related Work– the performed affiliate projects and studies from the literature will be reported.

Chapter 2

Background and motivations

2.1 The need for data-intensive tools

Data are nowadays the main source of value in many areas. Business strategies are now said to be *data-driven*, since they employ data collected from customers to improve suggested products. Moreover, narrowly targeted marketing campaigns are built up by the users' preferences extracted from processed data. Also, smart manufacturing companies started to improved their internal production chains by means of data analysis [KLC^{+}16]. Research areas such as physic conducts experiments which produce Petabytes' scale quantity of information. "CERN data centers process on average one petabyte of data per day".[1] The Internet of Things (IoT) advent promises to populate our lives with tens of billion devices by 2020 [N^{+}16]. These sensors and smart devices are already generating and will produce huge volumes of data too. Moreover, there are also others emerging sectors such as autonomous vehicles [GLPL14], smart energy management [ZFY16], and home automation [GP10] that show promise to flood computer systems with even more data. As a consequence, the listed scenarios, such as many others, will require in the near future software tools capable to face the processing and storage challenges imposed by the information exponential growth.

Data-intensive applications require to handle, process and exploit these large amount of data [Kle17]. This type of software is counterposed to the compute-intensive class that, instead, groups all the application scenarios in which the high computation performance is more important than data itself. For example: software simulation, emulation, and optimization algorithms. The development of powerful, flexible and scalable data-intensive applications is fundamental to sustain the increase in volume, production velocity, and formats complexity of data upcoming in the next future.

2.2 Distributed data systems

The data-intensive tools are frameworks leveraging distributed systems architecture to enable the development of data-intensive applications. Designing a software to only exploits resources of a single computer, although very powerful, would be for sure a failing approach. Data-intensive tools have been conceived to ease the development of applications capable of handling the huge volumes of data, combined with high generation rates, and the need for near real-time results. The requirements that led to their development are to overcome the technological and physical

[1] https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone

limitations of a single machine and also to abstract the complex functioning of a distributed architecture to ease developers' life in not having to handle all the infrastructure complexity.

Distributed systems are of fundamental importance to combine resources of many interconnected computers so to exploit in parallel processing power and handle heavy workloads. Unfortunately, data management in distributed systems is notoriously difficult due to the multiple computational resources, heterogeneous with each other, that need to be coordinated. A distributed scenario introduces concurrency and synchronization problems as well as the issues coming from the strategies adopted in the data arrangement, task execution, communication, failure tolerance and synchronization of replicated and partitioned data [TVS07]. To overcome all these obstacles, in the last decade, a wide variety of distributed platforms has been developed. These tools offer specialized programming interfaces to simplify the design and implementation of data-intensive applications.

The data management domain, with the rise of requirements for large scale processing, did outline the inadequacy of traditional relational model of classical databases. The complexity in handling the data required more flexibility and scalability. Workload nature changed as well, including the support to analytical tasks and complemented query tasks [Sto10]. Legacy solutions as databases could not scale due to the high cost in data bases synchronization and in the support to a reliable communication. Therefore, the choice of dropping the support to strong consistency and transactional guarantees in favor of scalability was made through some newly conceived systems. This design choice resulted in the birth of the NoSQL databases [Lea10]. Also, composite solutions were born and they ranged from key-value stores to document-based stores, and to graph databases. In recent times, the research community is looking for new design solutions and architectural structures that can reconcile scalability and strong consistency [Sto12]. The conducted studies are investigating limits and possibility of in-memory data storage and ways to better exploit modern parallel hardware [SW13] [TDW$^+$12].

In the data processing community, the increasing trend in data generation drove the evolution of a new breed of systems explicitly designed for large-scale distributed processing. This class of system, inspired by the Google's MapReduce paradigm [DG08], leverage dataflow model to arrange processing units into a directed graph of operators. Input data then made flow through the network of these functional transformations so to produce final results [ZDL$^+$13] [TTS$^+$14]. To provide task parallelism these systems let run different simultaneous operators on the same or on different machines. Data parallelism instead is achieved by splitting into independent partitions the input streams and directing these to parallel instances of the same operator. The abstraction in the processing definition at the application level enables the frameworks to automate operators' deployment, the exchange of data from and to the system, and to handle hardware failures by means of transparent re-execution of the lost processing.

2.3 Motivations

All the nowadays existing data-intensive tools offer programming interfaces with the objective to solve specific processing and data handling tasks. By the heterogeneity of requirements characterizing the application scenarios none of the data systems can alone satisfy all of them.

In practice, software architectures are becoming way more complex than the ones employed in the past because of the combination of multiple data systems. Then ad hoc logic is implemented to govern their interaction so as to exploit the specialized features each system has to offer. However, in doing so, the disciplined programming paradigms of individual systems lose in validity and benefits in the given guarantees on the data, deployment procedures and communication are no longer exploited.

The microservice architectural style is one of the most popular approaches to build large-scale applications. It works by breaking down the overall application into a suite of independent services that have distinct data management approaches [New15]. A common critique to this architectural style is that it moves the complexity from within each service to the coordination between services for data synchronization [Thö15] [BHJ15]. Furthermore, to be able to correctly orchestrate all the services it is required a deep understanding of their semantics, performance characteristics, deployment strategies, and configuration opportunities. For instance, data analytics tool are specialized in the scenarios of batch or continuous processing. They leverage hardware parallelization as well as in-memory or disk-based storage to optimize throughput and low latency. Instead, distributed databases afford even more configuration possibilities. They come with differentiated data models such as relational, object-based, key-value, and others. Also, they support multiple semantics for replication and by these they often impose trade-offs between performance and guarantees. Many of them do not have transactions' support or impose limits for the operations scope to the single partitions [LM10]. The databases that claim to provide transactions employ sometimes weaker-isolation semantic, which allow or prevent some of the anomalies that might appear during concurrent execution [BDF$^+$13]. Unfortunately, this information is not clearly stated but comes quite hidden inside each system's documentation. Moreover, currently it does not exist a classification model that captures formally the system's behavior. Such a schema would help to verify the meeting of requirements of the under-development applications with systems' capabilities. For example, in the situation of a database privileging high performance over replication consistency, replicas might experience temporary or persistent discrepancies between each other therefore affecting application correctness. This lacking frame, requires from expert engineers a lot of manual work in order to check how each system's assumptions reflect on the overall application needs. By their discoveries they will select, configure, and deploy sets of selected tools and make them cooperate with each other for the project objective.

In summary, the leading approach to develop data-intensive applications exploits multiple tools to solve specific and recurring problems. However, this then delegates to the developer the design of the overall application logic, fully exposing the complexity related to data representation, communication, coordination, deployment, performance modeling, and optimization. As a consequence, data-intensive deployment scenarios nowadays still need to undergo to laborious selection process even before starting the development, and later show high maintenance and operational costs.

2.4 Scope

The data-intensive tools domain exemplifies the idea of two worlds that, through different approaches but also some similarities, provide highly specialized features to process and storage data. The existing data-intensive tools show overlapping functionalities and design choices [Bre00] that advice for a unifying, still unknown underlying paradigm. This thesis work aspires to be the initial step in the understanding and characterization of the traits unifying or differentiating all the range of systems, from stream processors to distributed databases.

The core enabling element for the conception of an ideal new paradigm is a formal modeling framework that defines a high-level notions and structures, and captures the functionalities and characteristics required in data processing and management. In particular, this study examine the systems' adopted computational paradigms, the provided features and expressiveness, as well as the guarantees assured in concurrent operations' execution. Also, typical properties of distributed scenarios such as the reliability of communication channels and methodologies adopted to achieve fault-tolerance at the same time of high performance will be accounted for.

The spectrum of paradigms and implementation aspects though, would require the analysis of a large number of systems making the work unfeasible and also repetitive for similar data-intensive tools. Therefore, we decide to focus our attention by making experiments and studying the major exponents for classes of computational paradigms.

Data-intensive class	**Analyzed system**
Stream processing	Apache Flink
Batch processing	Apache Spark
Streaming platform by producer-consumer	Apache Kafka
NewSQL databases	VoltDB

Table 2.1: Analyzed systems by data-intensive tools class

Apache Flink

"Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale."[2]

Apache Spark

"Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. [...]"[3]

Our second choice was Apache Spark, as exponent for the class of batch processing. Also, the promoted computational model is different from Flink. Data are organized in Resilient Distributed Dataset (RDDS) [ZCD$^+$12] whose portions reside on the nodes. When new applications are deployed, the system moves the processing towards the data for its execution.

Delta Lake

"Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark and big data workloads."[4]

Delta Lake is an extension of Apache Spark that we decided to test because of the additional guarantees and abstractions the package provides. It leverages a different set of APIs with respect to Spark, allowing the developer to reason in terms of tables and through SQL queries. We report the features listed in the documentation state so to clarify the nature of the system. "ACID transactions on Spark, Scalable metadata handling, Streaming and batch unification, Schema enforcement, Time travel feature, Upserts and deletes" are the characteristics of Delta Lake.[5]

[2]Definition from `https://flink.apache.org/flink-architecture.html`
[3]Definition from `https://spark.apache.org/docs/2.4.4`
[4]Definition from `https://delta.io/`
[5]List of features from `https://docs.delta.io/latest/delta-intro.html`

Apache Kafka

Apache Kafka was included in the study due to its hybrid nature. Its architectural and operating patterns deviate slightly from the other systems' choices. Kafka provides to our analysis a different point of view with respect to solutions adopted to solve similar problems in the other systems scenarios. Furthermore, this system, born as a log storage tool, is becoming more and more employed by companies. It is widely employed when microservices architecture are built since it provides the required reliability in the communication channel between services.
"Kafka is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

"[6]

VoltDB

"VoltDB is an in-memory transactional database for modern applications that require an unprecedented combination of data scale, volume, and accuracy." [7]

Among the NewSQL existing databases we chose to study VoltDB. This distributed database, the evolution produced from the H-Store [KKN$^+$08] research project, demonstrates by a series of application examples the capability of exploiting the benefits derived from a coordination-free approach in the implementation of transitional ACID guarantees.

[6]Characteristics from `https://kafka.apache.org/intro`
[7]Definition from `https://www.voltdb.com/blog/category/in-memory-database/`

Chapter 3

The Modeling framework

A model is a representation of a concept. It can be used to describe, explain, and abstract phenomena, processes, and systems that cannot be empirically experienced or are too complex to be directly studied. Models play a central role in science since they are largely used as communication tools for researchers. Furthermore, they become very useful whenever there is the need to make people coming from different domains, hence using diverse terminologies, understand each other. The process of sharing knowledge is way more effective when done through models that provide abstractions and unified terminologies.

We developed a set of models rather than a single one in order to frame out multiple aspects proper to the systems' nature. A single flat model would have failed to bring out the advantages and limitations of different viewpoints mostly orthogonal to each other. For this reason, we develop a modeling framework capable to abstract concepts, operating modalities, and the given guarantees in terms of reliability and execution.

Conceptual model We started by creating a set of terms able to capture the recurring ideas. We propose a first conceptual model with two main objectives. First, it has to abstract constructs common to all the systems so as not to fall in the ambiguity brought by similar terms adopted in different context. Second, it aims to constitute the root basement of a shared terminology used by the other presented models.

Operational model With the emergence of various new systems and the intense evolution of the more established ones, different operating modes have been proposed. Each implementation involves its interaction modalities and employing these to analyze the systems would have concealed principles at the root of any functioning. The objective of operational model is precisely to provide a functioning abstraction as much as possible implementation-independent.

Architectural model In a distributed context a system functioning is also regulated by how and where resources are organized. By the built architectural model we provide a description of how data and computation portions can be arranged on multiple machines. The reported structure also aim to describe which are the considered physical units with respect to the parallelization of portions of data and computation.

After this model, we discussed the allocation and coordination strategies adopted in the resource arrangement in a distributed system.

Guarantees model The correct functioning of a system, as well as interactions clients have with it, are regulated by a set of properties. With this model, we aim to give a precise definition of the guarantees systems' provide in terms of execution correctness, architectures' reliability and durability of stored data in case of failures.

Guarantees' implementation Guarantees' definitions can be implemented with multiple techniques. Some of them are more suitable to distributed context, while others can involve major drawbacks in scalability. With the implementations' guarantees model we propose structure to relate the protocols and implementation strategies used to provide the guarantees.

3.1 Conceptual model

In this section, we identify the criteria that characterize the analyzed systems so as to provide a unified terminology and, at a later time, describe consistently the functional and architectural model.

The following schema describes the concepts' hierarchies and their relationships. It starts from those more abstract ones and it provides further specializations without the loss of generality. Terms and related ideas are unified, whenever possible, under the same concept and linked to each other in order to underline possible interactions between them. This builds up a shared terminology framework that gets rid of the typical misunderstandings raised by similar yet distinct terms used to mean the same concepts.

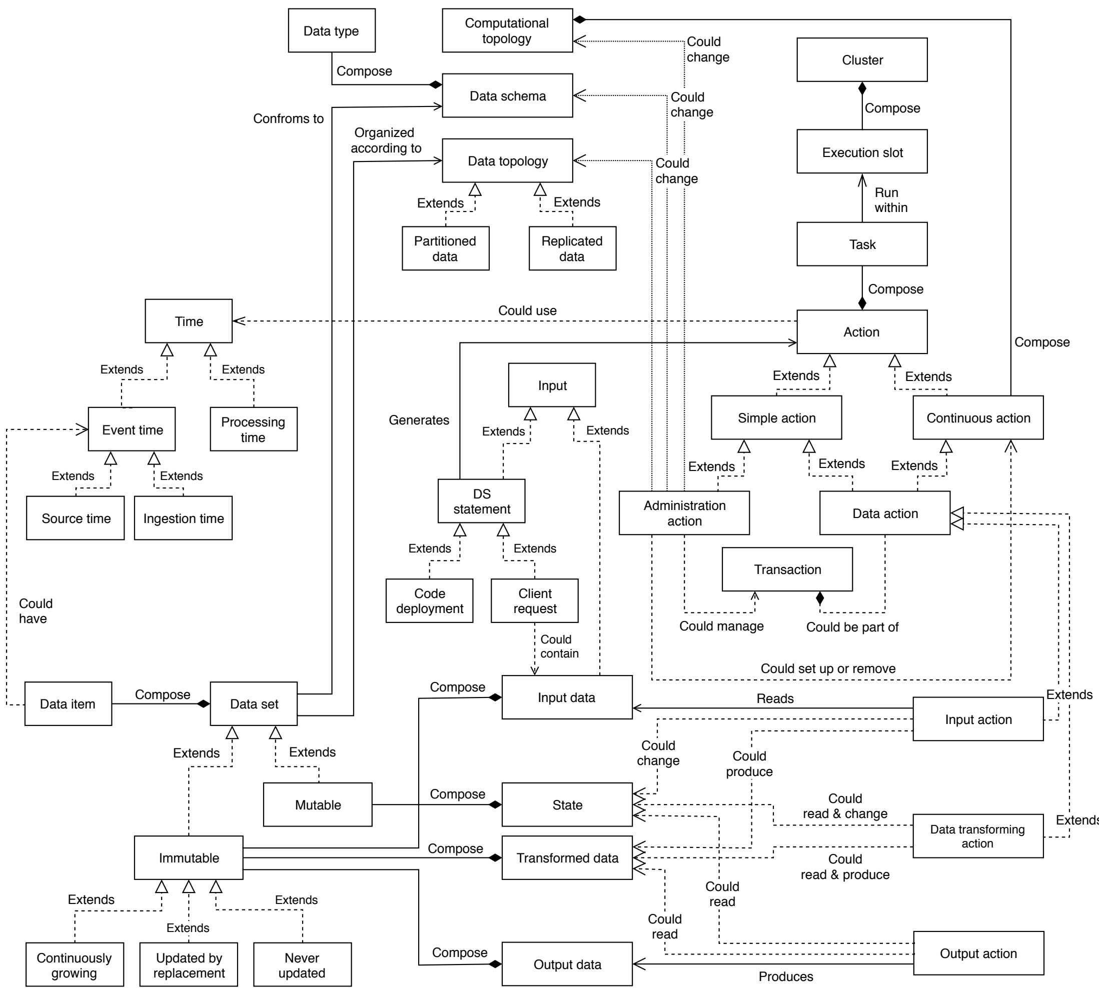


Figure 3.1: Conceptual model

The presented graphical schema represents concepts as boxes. Arrows are used to express connections and also to graphically state their meaning; they want to resemble the arrows' significance of UML standard. Those marked with "Extends" start from more specialized concepts and point to the abstract ones. Made exception for this last type of arrows, in general, dashed lines are intend for optional relationships while solid ones are used for mandatory links.

A graphical representation alone would not be sufficient to argue effectively the semantic behind modeled ideas. In the following paragraphs concepts are described by providing a definition of notions' role in the model and reasons for declared relationships.

Time

Time can be measured in two ways. On the first place, the physical time provided by the internal clock a machine has can be employed. Next, the second way to measure the time is to map the concept onto the occurrence of events. In this case we have logical time, an approach introduced by L. Lamport [Lam78]. The need to introduce this last type of time derives from the impossibility of absolutely synchronize physical clocks that have arbitrary precision. Logical clocks are instead a more practical way used to enforce ordering constraints between events that occur in a distributed system.

Event time Event time is a time attached to each data element. It can be further specialized into source and ingestion time according to the authority that affix it to the data.

Source time The source time is attached by the original system that creates the data element. (e.g. temperature sensor crafting a new value with attached the local time).

Ingestion time It is the time associated by the first component of the processing system that receives the data element from the outside of the system. (e.g. when the value of the temperature arrives to the data processing system).

Processing time In this case, the time is not a logical concept associated to each data element, but it is the wall clock time of the specific task when it is handling the data element. Essentially it is the local machine time at the processing momentum. (e.g. the time read from the physical machine clock while processing the temperature value).

Data type

The data type is the form a single datum has. It essentially corresponds to the computer-science meaning of data type. Thus, it constitute the information that is used to return the meaning of an amorphous set of bytes and to the shape it as an integer number, float, double, string, character... The arrangement of data types into a structure forms up the concept of data schema presented below.

Data item

A data item is a unit of information a system manages. It constitutes the simplest data piece singularly manipulated by tasks to carry out the overall processing. Its internal structure is defined by the data schema, a concept that will be defined in a while.

Stream processors and databases use different terminology to express this same concept. The first call it *event* while the latter call it *tuple* or *row*. This use case is particularly effective to show how the conceptual model we are proposing is useful to standardize terminologies from different areas, that employ dissimilar nouns for the same idea.

Data schema

Data schema is the structure that logically organizes the data types to define the shape data items have. It characterizes data sets since it acts as a frame, a schema, through which data items are handled. This concept encapsulate the logical abstractions the tools provide to handle data sets types.

Table schema All the modeled tools provide ways to handle data sets as if they were tables. We extend the broader data schema concept by the notion of table schema. In these last few years indeed the trend to provide SQL-like functionalities has shown up. Streaming and in general big data-intensive tools are approaching always more to the functionalities of databases and are starting to provide table-like data structures to handle streams. The table schema notion brings the databases' semantic. However, stream processor implement it by providing also additional and more expressive functionalities to account for big data-specific problems.

Custom schema Besides abstractions like tables, simpler and lighter data structures are also used for simplicity and communication efficiency. Custom schemes are flexible and modular structures used to better address applications' requirements. They are defined within and thanks to programming code. The possibility to frame any custom data organization has the advantage of adding flexibility in handling particular formats and also gives room to the platforms for automatic optimizations. However, since custom data schemes are platforms and programming languages-dependent they inevitably introduce compatibility and interoperability limitations with external systems.

Data set

Single unit of data are grouped into logical collections defined as data sets. They are made out of items that have the same data schema and that were group to ease management, processing and design of data logical structures.

Whenever a software needs to load and store data items, it employs the concept of data set to refer to a group of data items. The policies a data set has been created or stored through do classify it as mutable or immutable.

Mutable data set Items that compose mutable data sets can be partially modified, rewritten entirely or deleted. Mutable data set is the form of data used to store a state because this format can be altered partially at the cost of a small overhead of the data structure in memory. This property is highly desirable when it is required, as in the case of state, to store information that is subject to frequent changes.

Immutable data set Conversely, immutable data sets' peculiarity is that they can be altered in no way. Generated most of the time by continuous data transformations once they are written, either in memory or to the disk, they can only be read.

Despite the strict definition of immutability, we extend the notion to better match the one processing and data storage systems give to it.

Never updated The simplest class is the one in which immutable data sets are effectively never updated. Once generated by a task or group of tasks they can be stored, read and used for later computations but never modified.

Continuously growing Continuously growing data sets instead cannot be altered in any way if not by appending data items. They can only keep on growing in dimension by new elements appended to their end or they can be fully deleted.

Updated by replacement Finally, the case of updated by replacement wants to capture the cases in which data sets are indeed immutable but they need to show up some updates by the application which generated them. In order to do so, they gets wholly replaced by a new version of the immutable data set.

Input

An input is what it is put in, sent to, taken into by a process or system. In the studied domain it is specialized into the concepts of DS and input data.

DS statement The concept of Domain Specific statement aspires to abstract all the commands, actions and various kind of invocations that are used to control, query and modify operation of a system. Once interpreted, a DS statement generates one or more actions. An action is the work (and planning) that is performed in order to satisfy a request. So it can be, for example, either the work carried out to produce an output result or the one executed to change the internal structure of the system, that by itself does not produce tangible outputs.

There exist two strategies to instruct a complex platform with data processing and storage requirements. For this reason, the high-level concept of DS statement is below specialized into two more precise sub-classes.

Code deployment Stream processor platforms and modern databases adopt a package-driven approach to specify instructions. They provide libraries and APIs to expose the processing functionalities. Developers employ those to build a software package that collects all the information about required processing and data storage policies. The combinations of API calls allow to define the desired application logic by software code that then its compiled into a package and deployed to the system for execution. The code-deployment mechanism brings the advantage that transformations, data schema and topology information are reported in one place. Therefore the processing logic can be either entirely or not at all deployed to the cluster for execution. Moreover, especially some tools employ this technique to reduce latency and optimize the user requests. By using a single container, complex computational structures become easier to be maintained. Also, the developer does not need anymore to adjust modifications according to the remote data schema since new versions will entirely overwrite it. However, the package deployment approach has the disadvantage of being rigid in the updates. When a procedure needs to be even slightly modified, the platform is stopped and the entire package needs to be deployed again.

Client request The client request concept encapsulates those instructions sent to a system to both alter topologies (data and computational) and to fetch, update, insert data items. They differs from the application deployment statements since they are shorter and do operate less changes per single client request. In fact, to instruct the system about an entire data topology multiple client requests are required while in case of application deployment it would be sent only one DS statement to the platform.

Input data

With the concept of input data we want to model all the information that is sent, read or provided to a system in order to either change its state or produce output data as result of transformations. Input data can be carried to the system in two ways. First method is through clients' requests that might carry one or more data items within their body. When this modality is used, the number of data items sent per-request is not very large. Instead, when the necessity is to process large sets made out of thousands or millions of data items, continuous input actions are adopted. They are set up at the front of the computational topology and constantly ingest new incoming data. Also, when the whole historical large-sized data sets are available these actions take care of gradually read them.

State

State is defined as the status a system is in as a consequence of all the input received so far. It is made out of a mutable data set because during execution of tasks it could change multiple times. State is also the core concept of any stateful systems and is getting way more important also in stream processors; those systems that initially were thought only to barely extract, transform and load data (ETL[1]) without relying extensively on a state.

Operator variables Stream processors use operator variables as room to store state. Intermediate processing results and inputs are withheld in the scope of functions constituting operators. This type of state is made fault-tolerant by using methods that ensure a backup to multiple copies or to persistent memory.

Tables Instead, databases born from requirements of the time that was to store efficiently a state. The concept of table is the main logical abstraction employed for the organization of data sets and their data items (the table rows).

Transformed data

Transformed data sets originate by transformation actions. They are the result of processing either input data or other transformed data set. Since both of the original data sets are immutable too, the transformed version is obtained by making new copies of data items which comprise the desired changes. Transformed data sets are mainly produced by continuous actions. In the process of copy-transformation they can also read and modify the system state.

Output data

We define the data items leaving the system as output data. Same as input data, outputs are immutable data sets with the difference that they do leave the system and are sent towards external clients. They happen in forms of continuously growing, updated by replacement and never updated immutable data sets.

[1]"Extraction-Transformation-Loading (ETL) tools are pieces of software responsible for the extraction of data from several sources, their cleansing, customization and insertion into a data warehouse." [VSS02]

Data topology

Data topology is the physical organization of data with respect to the available execution slots. According to the pursued optimization, the system can organize data sets as replicated or partitioned.

Replicated data Whenever a set of elements is frequently accessed in read mode and its size is relatively small, the most suitable storage strategy is replication. A copy of the same information is made available on each node of the cluster and it is updated by broadcasting the relatively infrequent changes to all the nodes. An advantage of this scheme is that tasks access with low latency the information and, so, there's no need to use network communication in order to retrieve remote data. On the contrary, replication can introduce high overhead to keep copies up to date and consistent with each other. Also, due to the constraints imposed by the main memory size, it is not recommended to store a very large number of items since otherwise they will need to be stored on disk (higher access times).

Partitioned data Partitioning is required when data sets do not fit into a single execution slot. This storage approach acts by breaking down the original collection of items into disjoint subsets and by locating them to execution slots. The partitioning approach is also used for optimizations and efficient parallelization of task executions. This strategy is indeed exploited when *group-operations* are performed, e.g. flatMapWithState. Elements belonging to the same group get collected into the same partition and next computations based on this assumption can be performed with ease within the execution slot(s) hosting the target items.

The two classes shortly presented above constitute a further data classification. It is important to note that this categorization is transversal to the one which defines mutability and immutability. So there exist mutable partitioned data, mutable replicated data as well as immutable partitioned or replicated data.

Action

We define an action as the set of operations generated by the semantic interpretation of a DS statement. An action is then further broken into smaller subsets of operations, called tasks. In turn, these chuck of computation are distributed to execution slots. In order to create a logical arrangement, we say that tasks are executed within execution slots. Presented a bit later, execution slots can be many for a single physical machine and they can host multiple tasks. Hence, many tasks, portions of an action or various actions, can run in parallel on the same machine as long as there are available resource or the planned actions are completed. In general, an action is considered as the work carried out within the system and which is built by the interpretation of a DS statement.

Two classes can be defined with respect to actions execution lifespan.

Simple actions As soon as a simple action achieves the goal it was crafted for it is then discarded. No trace remains into the system made exception for the eventual modifications made to the state or to the data, computational topologies. The execution plan of an SQL query is an example of action belonging to this class. Like it also is, the set of operations carried out by Flink to fetch a queryable state from an operator.

Continuous actions On the contrary, continuous actions keep on running until dismissed. Continuous actions are installed and removed from the cluster by designated administrative, simple actions. Once installed they become part of the permanent system structure. Their role is to wait for data to be fed as input, perform computations and then emit results as continuous outputs. They goes under the name of *operators* when looking at the stream-processor domain. While,for the database field, they are mapped to the set of operations the engine fulfill to keep update an SQL view, in turn being the equivalent of transformed data concept.

The action classification can also be done according to the subject they operate on within the system. Administration actions are those used to change the data types, system schema and also, by setting up or removing continuous actions, the computational topology. Instead, we define as data actions those that are mainly focus to operate over data. This classification, like the one that distinguishes between replicated and partitioned data, is totally independent with respect to simple and continuous classification aforementioned.

Data action

The concept of actions related to data is specialized into 3 sub-classes according to the objective they were generated for.

Input actions This type of actions is in charge of reading new data provided as input to the system. Then, they include them as part of the system by either producing some transformed data or by altering some state though the insertion of the information. Looking at Flink, we map this concept to the operations the platform execute when it interprets the *DataSource* DS statement. Instead, in the case of VoltDB, we correspond this concept to the operations the system carries when it has to add new records to a table.

Data transforming actions Data, in the form of state or transformed data, that are part of the system are modified by data transforming actions. The function they have is crucial for any system since they are the method to change the internal state (insert, update and delete) and to produce new transformed data as result of processing transformations.

Output actions Output actions are those operations performed to read information from the inside of the system and send them out to the outside world. The function they have is also, looking at the general frame, to transport partial results towards the part of the system in charge to unite and build up the complete results. For example, in a partitioned data set, these activities carry the partial information, selected according to a condition, from the partitions to the node in charge of building the output answer. Moreover, the continuous form of output actions is the one that provides what are known as output streams in the stream processors domain.

Administration action

Opposed to data actions, we group all the administrative and system's management related commands into the class of administration actions. Within this group we place all the SQL statements that change a database schema (tables, indexes and views creation as well as the statements to alter them). For stream processors instead, we consider administrative actions the operations generated by interpreting *API* calls of the user application. They are in fact used to instruct the system on the computational structure to be set up on the platform in order to process the data streams. This set of actions extends the concept of simple action since, making an example for stream processors case, by them we point at all the operations which install or remove continuous actions but that then will leave no traces on the cluster once performed.

Transaction

A transaction is the logical grouping of read and write operations. The concept born from the necessity to receive execution guarantees with respect to sets of operations. Applications leverage the transactions' functioning to dramatically simplify their internal operation and design. By them, they can perform constraints' checkups and eventually modify a status without the worry to affect, in the meantime of execution, the overall system's integrity and consistency. During a transaction execution, two particular statements can be used to express the end of the logical set of operations: *COMMIT* and *ABORT*. They define the successful outcome or the failure of a transaction. When a *COMMIT* is attempted by an application, the changes it made are meant to be saved. Instead, in the case of *ABORT*, all the modified information has not to be considered part of the database state. The data management system can also, in turn, impose a transaction to abort because of conflicts with other concurrent operations or due to the violation of some consistency constraints.

Tasks

A task is the minimal portion of execution an action can be split into. Tasks result from the decomposition of actions into smaller units of work that can be sent to workers. Potentially, copies of the same task are crafted to be run in parallel so as to achieve higher performance. The computational topology described below is made up by the logical arrangement of tasks that are sent by the platform engine to execution slots.

Execution slot

An execution slot is a logical processing unit that abstracts computational and memory resource. This concepts is linked to CPU cores since is a widely employed setup the association of one execution slot to one logical CPU core.

Computational topology

This concept captures the logical workflow actions are executed by. It encapsulate the physical workload distribution and its conceptual organization. The computational topology is the structure originated from chaining multiple actions together into a processing graph or sequence of stages. Based on which of these two last approaches is used, this concept is extended in two counterposed alternatives. A static computational topology involves deployment of continuous tasks to execution slots that stay running and makes data items flows through different execution slots and workers. Instead, when a dynamic approach is used, the continuous tasks that are supposed to run on portion of data sets are all deployed to an execution slots and, in order, scheduled for execution. This last solution accounts more for data locality since processed data items ideally remains stored within the same execution slot.

3.2 Operational model

The objective of this section is to capture the operation flow within a system carried out to fulfill processing and storage needs. During the presentation of this model, new constructs related to the physical architecture of a system will be described.

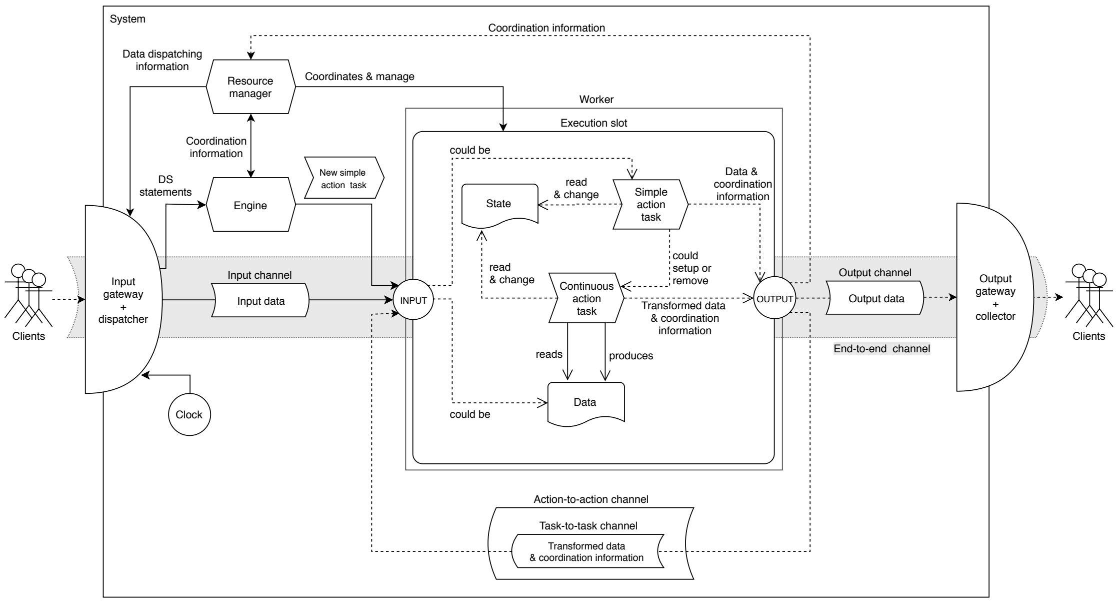


Figure 3.2: Operational model

This scheme aims to propose a model of the platforms' functioning. Its structure wants to suggest a reading key that traces, from left to right, an interaction scenario of input, processing and output of data. The clients, placed on the two sides, are not necessarily different from each others but instead model the sending of inputs and receiving of outputs.

Clients

Clients are representative for user applications and external processing platforms not part of modeled system. They are connected to the execution environment through input and output gateways. Also called connectors, since the they hook up different systems, they are software components that implement a shared communication channel used to exchange data from and to the outside world.

Input gateway and dispatcher

The overall model corresponds to systems that works in a distributed fashion. As a consequence, many are the machines towards which the *connectors* need to send the input data intended for processing. We assign the role of data items dispatcher to the input gateway block. Its function is to settle the problem of towards which worker has to be forwarded the information sent to the system. In particular it is in charge by the dispatching information it received from the resource manager component to route the data items towards execution slots within which tasks are running. The resource manager, aware of available resources and in charge of balance the workload fairly, send the required information to the input gateway.

This acquisition block has a further function: it attaches a time to the ingested data items. This feature is required to be enabled by the continuous input actions that compose this logical block. The time that gets tied to the passing items is described by the conceptual model as ingestion time. Indeed, at the input gateway it is the first moment the system can read a new information.

Lastly, another capability the input gateway encapsulate is to behave as a requests proxy. All clients requires to have an agreed endpoint towards which send DS statements. Since multiple masters could simultaneously exist at run-time, the proxy capability is to route the request towards the right master. So, it indeed acts as stationary and steady interface to outside world against which clients can send DS statements without caring about master being changed after a failure.

Output gateway and collector

This component is the dual to the input gateway and has similar functions. Its role is to collect data from the nodes of the cluster to be sent towards the outside of the system. We model it as logical gateway. The reason for this is that *external systems* as well as the clients expect data to be provided in manageable formats and aggregations. Indeed one of the main goal it has is to act as aggregator for sparse data that need to be merged or united before leaving the system. Providing an example, in the case of a database, imagine we want to run an SQL *SELECT* query with a *WHERE* condition. Information to be queried has been partitioned over a cluster and hence is located into multiple location. After the filtering procedure operated by the nodes, partial results of the query have to be merged to build the consistent, expected final result. All the pieces of information are sent to output gateway to be arranged in the correct way to craft the output the client asked for.

Communication channels

Communication channels are tools by which the data items are moved to the workers, between them and to the external clients. They are a logical abstraction that is then implemented by means of actual communication channels such as sockets. They are an important part in the functioning of system since its by means of them that data are internally moved between execution slots and from-to input-output gateways.

Task-to-task channel The task-to-task channel is the abstraction we employ to specify how tasks communicate with each other. In particular, this transportation means is employed to move single data items to different tasks thus enabling the idea of data flow.

Action-to-action channel We move to a more abstract level to reason about the execution of actions instead of tasks that compose them. These portions of the overall execution can be chained together to define a complex transformations. With the action-to-action channel, we model the logical connection that is built among different actions and that gets used to move data items towards later transformations.

End-to-end channel We enclosed in this operational model also the abstraction of a logical end-to-end channel. It describes the interactions between two external clients, usually different from each other, that leverage capabilities and guarantees offered by a system to carry a data item from one side to the other.

Internal system operation

We describe now how the internal aspects of the system work. DS statements and new input data are the triggers for any computational process. They mutate both the system state and its behavior. Received and passed over by the input gateway, the new instruction reaches the engine and according to the semantic it has it gets interpreted generating an action. The latter being a set of required operations to carry out is split up by the engine into new multiple tasks. Later, they are sent to execution slots. The policy that regulates how and where arrange the tasks is enforced by the engine itself and is adjusted according to the collected coordination information. These last metadata are received from the resource manager. This last component is modeled separated from the engine since many of the analyzed platforms allow to use external resource manager tools. (E.g. Hadoop YARN, Apache Mesos and others.). New tasks are received by the workers, the machines in charge of execution. The worker manager process, presented in the next architectural model, takes care to assign tasks to the local execution slots. These latter processes, internally takes care to effectively perform task operations. The execution of a simple action task can potentially read and/or change the local state. In case it was an part of an administration action a task execution can result in the setup or removal of other continuous data action tasks. A continuous data action task can read or change the state while it process incoming data. The output of an execution slot is made out of coordination information, transformed data and potentially output data. Coordination information are sent to the resource manager for processing statistics. Instead, transformed data can be directed either to other execution slots or towards the output gateway.

3.3 Architectural model

We model the physical infrastructure and resource allocation with the following architectural model. This representation defines the places where the computational units of the system reside and the logic of data allocation on multiple workers.

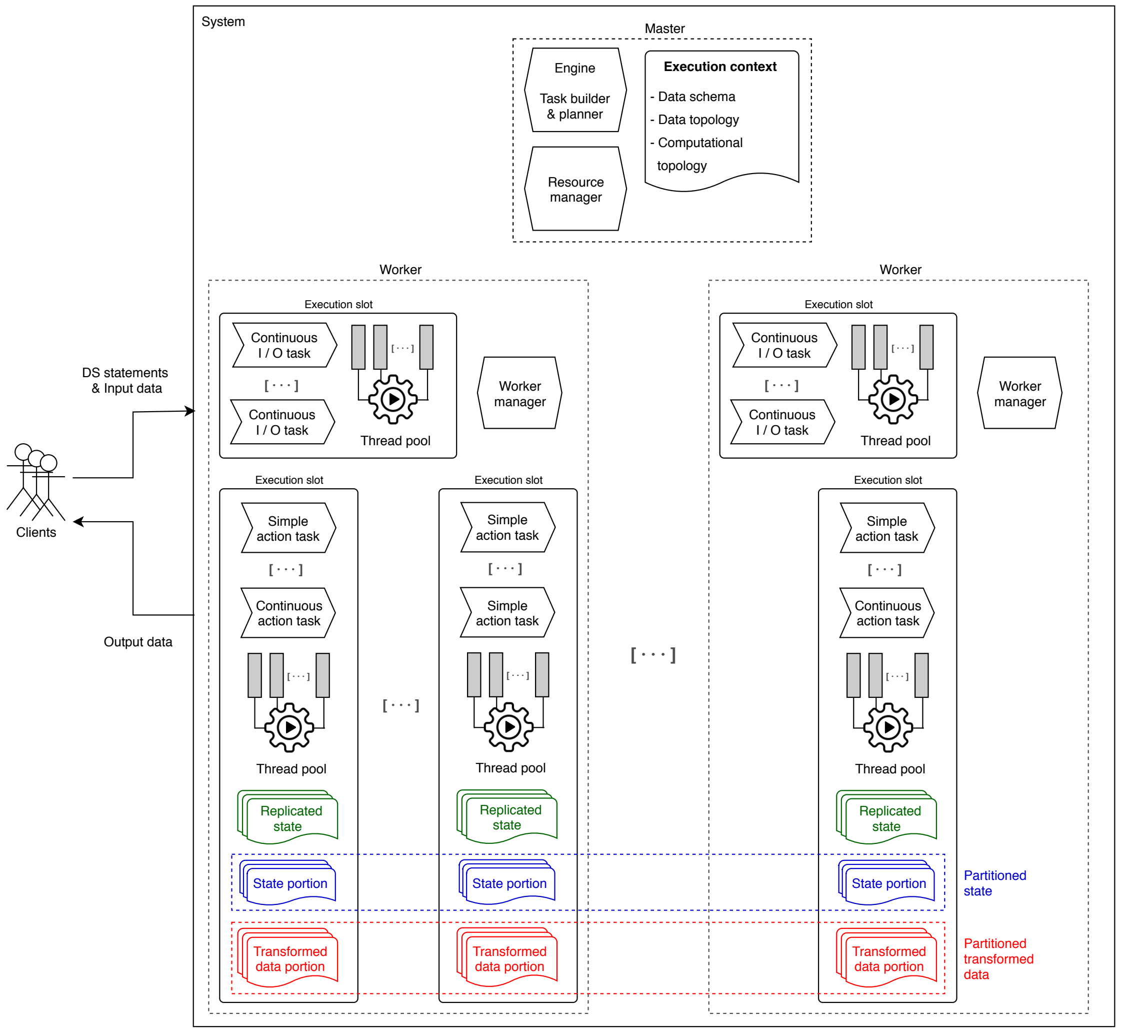


Figure 3.3: Architectural model

This layout's objective is to represent how memory and computational resources are organized with respect to the architecture of the system. Components that are replicated for fault-tolerance purposes have not been modeled for clarity of the schema. They will be described in the guarantees' implementations model later presented.

Master and workers

Any machine of the cluster can be used either as a worker or master. Fault tolerant configurations imply the setup of more than one master node to handle crashes. Their number is usually confined to three, indeed two of the alternative masters always remain idle and wait for the main one to crash. When this happen, an election process starts and a new leader is established among the two to take the place of the previous master.

Worker manager

However, made exception for few masters, a cluster is mainly made out of worker nodes. These servers are managed by running a demon process, the worker manager, that comes alongside the installation of big data processing and storage platforms. The role of this utility process is to make available the local resource by exchanging coordination information and statistics with the resource manager and also other worker managers. Moreover it is in charge of instantiating the required execution slots in the deployment stage.

Execution slots

Software processes running onto workers are employed to host computation and data. We abstract them with the concept of execution slot. They can be multiple on a single same machine. In this case they communicate using shared memory while, instead, when they run on different nodes the communication is carried out through the network task-to-task channels as described in the operational model section 3.2.

What they provide are isolated resource environments that can run tasks as a software threads by means of thread pool. They also store state as well as transformed data.

Some executions slots have been modeled without internal state or transformed data because they are thought to host only input / output tasks. They form the input and output gateway presented into the operational model section 3.2. The execution of continuous input tasks result in data reading and storing operations that provides *sources* and *sinks* for other running tasks.

Engine

The engine is the task builder and planner. Its role is to interpret the semantic of DS statements and to produce actions and, in turn, split them up into tasks. According to the available workers and the parallelism required by the action it then formulates an execution plan that is used to create the desired computational topology as well as to spread the data to form the most effective data topology. Within the engine, we locate also the execution context. This large state variable has the important role to store the administrative information for the cluster. It is used by the engine to store and fetch the current computational topology, the data schema workers are using and the in place data topology. All these information are useful to interpret and plan the execution of new DS statements and meaningful tasks.

Resource manager

The resource manager is a subsystem is in charge to handle and administrate all the physical nodes of the cluster. It manages machines by interacting with the worker managers and provides the engine information about the platform status, load statistics and task completions. it is located inside master node since data processing platforms frequently offer this service by themselves. However, there alternative deployment scenarios employ external resource managers. In this case a different system is installed alongside the main stream processor or database. Among the established resource managers there are YARN, MESOS and Kubertenes.

Resource allocation and coordination

Alongside the model of a system's architecture, it is meaningful to argue about how the allocation strategies can affect the overall system performance and reliability. The following conceptual map summarizes the effects derived by design choices and implementation methods.

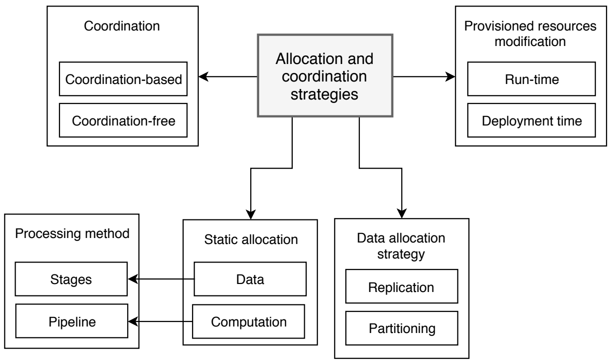


Figure 3.4: Schema of allocation and coordination strategies

Coordination The performance of a system is determined by the architecture and approaches adopted in system's design. They are driven by how many and which guarantees with respect to the correct execution of operations a system chooses to provide. A critical factor determining systems' performance is the coordination model they adopt to provide guarantees in the execution of highly concurrent operations. Classic approaches like two phase locking (2PL) and Multi Versioning Concurrency Control (MVCC) are widely adopt by databases. Stream processors, in turn, started to employ these mechanisms to enforce execution correctness and provide stronger guarantees. In the study conducted by P. Bailis [BFF$^+$14], they demonstrated that in the vast majority of cases, a coordination-free approach would take over classic locking methods. Transactions need to be scheduled in a way so that collisions are impossible to happen. The overhead that would have been caused by locking protocols, required to orchestrate a conflicting executions, is not present. Without locking, they have shown the possibility to obtain a substantial increase in execution throughput, thus giving evidence of how optimistic approaches such as scheduling can overtake pessimistic ones, such as 2PL.

Provisioned resources modification The workload on data-intensive tools varies concerning the amount of data processed in a time unit and, also, based on the number of external-clients requests the system has to satisfy. It the case of overload, processing performance can decrease inducing higher response times. To cope with this problem, system administrators can add new servers to the cluster. However, not all the tools are able to scale-out in order to leverage the provided computational power. Data-intensive tools employ optimization algorithms and abstraction strategies to exploit in the best way available resources. Their architecture and the adopted functioning mechanism imposes the limitations they have in dynamic resources management. There are indeed data-intensive that are not able to handle the addition or removal of new servers and so requiring the restart of running application to re-adjust the workload balance. For this reason, we argue that an important factor in the existing strategies in resource allocation is the ability to undergo, at run-time, to modifications in the provisioned resources.

Data allocation strategy In a distributed systems data sets are stored on multiple servers. The strategy chosen to split or replicate the information is of fundamental importance for system performance and computation efficiency. There exist two possibilities. The first approach is to split up the original entire data set into disjoint subsets. These logical subgroups, named partitions, are positioned over the available nodes so to achieve the higher data-parallelism possible. The partitioning strategy is also exploited when data sets are very large. By carefully deciding how to split up the original collection, it is possible to store massive volume of data split into many chucks. Lastly, the partitioning technique is leveraged by coordination-free guarantees methodologies to ensure the correctness in execution of concurrent operations, sequentially organized for each partition, that will never conflict with each other since physically separated [BFF+14].

The other strategy for data allocation is replication. In this case, a portion of data is copied identically to multiple nodes. By creating many replicas, system performance can improve since clients, and also internal operations, can access in parallel copies of the same data. Data-intensive tools achieve higher execution-parallelism when support this allocation strategy. However, when it is exploited in the wrong way, replication also introduces disadvantages such as execution overhead and discrepancies in the read information. Multiple replicas must be kept synchronized with each other, so to do not undermine the execution correctness of those clients accessing multiple replicas and pretending to read a consistent version.

Static allocation The paradigms adopted in distributed architectures always involve the choice to either keep stationary the data or the computation. The data-intensive tools that decide to maintain data saved over the nodes do employ the network to send the transformations to the nodes. This involves a schedule-based processing approach, like Map-Reduce, since it requires the operations to be scheduled in some order to apply reasonable transformations in parallel to each data set.

With the opposite approach, it is the calculation that is kept stationary on the nodes. The previous method acted by scheduling and sending through the network the stages of processing. In this one, the stages are instead statically distributed over the nodes and they take the name of operators. They consist of a function that applies transformations to the data and that gets executed continuously. To build up complex transformations, multiple operators are logically connected to form processing pipeline. In this approach, the data are moved from one operator to the other to, through numerous transformations, go to constitute the output result.

3.4 Guarantees model

The following diagram shows an hierarchy of the guarantees employed to ensure correct execution, reliability, and durability in data managing systems. The objective is to clarify to the reader which definition in the literature we are referring to by cutting out possible ambiguities originated from multiple similar yet different definitions.

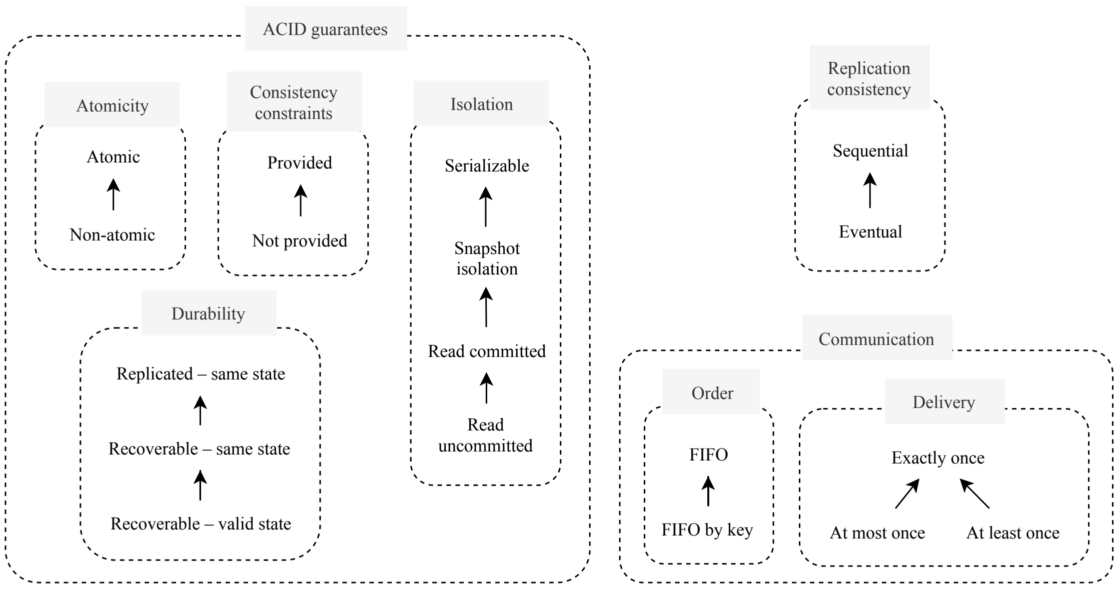


Figure 3.5: Guarantees model

These lattices sketch partial orders based on the semantic "strength" of guarantees considered by this study. The schema reports only a subset of all the existing levels of guarantees since listing them all it is not our objective neither would be useful to characterize the data-intensive tools analyzed. However, all of those used by the platforms are here reported and ordered.

In this model, we considered both ACID guarantees, native of the database domain, and also those required in distributed contexts such as the one of the data-intensive tools studied. The replication consistency and guarantees in communication, such as ordering and delivery of exchanged messages, are very relevant in any distributed system.

Atomicity

The guarantee of atomicity ensures that either all or none of the operations belonging to the same transaction succeed. If completion turns out to be partial due to failures or errors, then none of the already performed changes has to be visible to any of the other operations. In distributed systems, atomicity property is particularly difficult to be guaranteed without compromising performance. Its implementation is often done by the use of locking-protocols (e.g. two-phase locking) for execution synchronization. However, recent paradigms provide atomicity by means of coordination-free approaches. In this case, a transaction is required to be scheduled in same order with respect to concurrent operations at all the replicas, so that either all or none of its instances will succeed.

Consistency constraints

The consistency notion to which the ACID acronym refers is about data contents consistency. This property derives from the fact that databases can guarantee that operations performed by users do not violate constraints defined over the data contents. Also, thanks to the rigid data structure, they allow defining constraints such as referential integrity, uniqueness, mandatory presence of values, and user-defined special checks. In some situations, a system is said to provide consistency constraints when it support cascading aborts. With this function, the system verifies new data or changes against all the imposed consistency constraints. If it discovers a violation, even partial, then the data management system proceeds to restore the last coherent state by doing a cascading rollback of all the modified information.

To avoid the name clash with the concept of consistency of replicas, below described, we will point to ACID consistency through the term "consistency constraints". This guarantee can be either provided or not and has no intermediate levels.

Isolation

The isolation guarantee constrains the interleaving of simultaneous operations execution. It focuses on the prevention of conflicts, later defined as anomaly phenomena, that can happen due to concurrent access to the same data.

In order to describe what kind of isolation guarantees exist, it is fundamental to first define the nature of anomaly phenomena so to then, describe isolation levels in terms of prevented anomalies.

Anomaly phenomena Anomaly phenomena are the reason for inconsistent results as well as misleading data readings. They happen because simultaneous transactions perform read and writes operation in a certain sequence. They were first defined from ANSI SQL-92 standard [X3.92] that, later, has been proven to be inaccurate and incomplete. Berenson and his team [BBG$^+$95] found two more existing anomalies (Dirty write and Lost update) and improved their formal definition.

We below report the reads and writes operations so to exemplify the conflict situations. A write operation w, performed by the transaction 1 to a datum x will be written as $w_1[x]$. Analogously, a read performed by the same transaction on the same datum is written as $r_1[x]$. A read operation, performed on a collection matching the predicate P, has the notation $r_1[P]$.

P0	$w_1[x]$ ... $w_2[x]$ ... $(c_1 \ or \ a_1)$	Dirty Write
P1	$w_1[x]$ ... $r_2[x]$ ... $(c_1 \ or \ a_1)$	Dirty Read
P2	$r_1[x]$ ... $w_2[x]$ ... $(c_1 \ or \ a_1)$	Fuzzy or Non-Repeatable Read
P3	$r_1[P]$ ... $w_2[y \ in \ P]$ ... $(c_1 \ or \ a_1)$	Phantom
P4	$r_1[x]$ ... $w_2[x]$ ... $w_1[x] \ c_1$	Lost update

P0 – Dirty Write This was a missing phenomena in the ANSI-SQL definitions. It sees a transaction T_1 changing a datum x. Then a transaction T_2 modifies the same datum before T_1 can perform a $COMMIT$ or $ABORT$. An issue arise in case either T_1 or T_2 performs a $ABORT$ since it becomes uncertain what should be the correct value written to x.

P1 – Dirty Read (ANSI A1) In this situation a transaction T_2 reads a value that is uncertain. In case of T_1 $ABORT$ the value read for datum x is inconsistent with the actual value.

P2 – Fuzzy or Non-Repeatable Read (ANSI A2) A transaction T_1 reads the value of x but immediately after a transaction T_2 update this value. It turns out that T_1 in any situation of $COMMIT$ or $ABORT$ has in its hands n out of date value.

P3 – Phantom (ANSI A3) When this phenomena occurs, a transaction T_1 performs a read of multiple data items based on a search condition P. Subsequently, T_2 write a new datum that would have matched the predicate P and hence should have been either read by T_1 or written only after T_1 end.

P4 – Lost update This type of anomaly happen when a transaction T_1 reads a data item x followed by another transaction T_2 that updates x. Later, T_1 performs a write over x possibly based on the previously read value and updates the data item, then commits. Thus, this lead to the lost of the value written to x by the transaction T_2.

Isolation levels In the following table we summarize the existing isolation levels defined by means of prevented and possible anomalies.

	Prevented anomalies	**Possible anomalies**
Serializable	- All	- None
Snapshot isolation	- Phantom updates (reads only) - Read skew - Dirty reads - Dirty writes	- Write skew
Read committed	- Dirty reads - Dirty writes	- Write skew - Phantom updates - Read skew
Read uncommitted	- Dirty writes	- Write skew - Phantom updates - Read skew - Dirty reads

Table 3.1: Isolation levels

The definitions just above reported are however constrained to pessimistic implementations such as locking. To overcome the problem and give more room to optimistic and coordination-free schemes a new set of specifications has been formalized in the work "Generalized Isolation Level Definitions" [ALO00]. This paper provides implementation-independent definitions and with them a new generalized version of (portable) isolation levels.

Durability

The guarantee of durability gives the certainty that once a transaction has been successfully completed, the changes it has made will not be lost even in case of a system's failure. More precisely, durability property is defined in terms of tolerance to failures. Based on the methods adopted to achieve fault-tolerance some durability classes can be outlined. In decreasing order with respect to provided guarantees and performance we define the replicated – same state, recoverable – same state and recoverable – valid state guarantees.

Replicated – same state Replicated systems are those characterized by the lower recovery time. Multiple replicas of data are created in order to overcome failures almost without affecting the processing. In case of failure, the workload is redirected to an healthy replica and clients experience only an increase in latency response. Replication strategy is also used to improve system performance. Multiple copies of the same data are in this case exploited to run in parallel multiple read operations.

However, replicated system are better characterize by the notion of F-fault tolerance. Replication guarantee is the contract requiring that at least $F+1$ replicas have saved the changes and so F faults can simultaneously occur being the stored information still safe. When a replica fails, the system takes care to create another copy of the no longer available one. In an hypothetical extreme case, data is still guaranteed to be durable even with only one copy survived.

Recoverable – same state A different approach to durability involves the making of a backup of the intermediate system's state. In case of failure the saved state can be restore and the execution. By this approach, the computation is starter over from the exact same point that was saved during the saving of the state.

Recoverable – valid state A slightly different class of durability is derived by relaxing the constraint of above presented one. Some application scenario tolerate to resume execution after a failure from a state that wasn't the original one. However, it has to be a valid version, acceptable for the execution's correctness. By this durability guarantee it is not required that the recovery procedure will restore the same exact state present before the failure.

Replication consistency

Replication consistency is the guarantee ensuring that different instances of the same data are up to date with the same information after changes to one of the copies occur. In the analyzed data-intensive tools replication is widely employed to provide both fault-tolerance and higher execution throughput by parallel accesses.

Below we describe the consistency guarantees that we encountered in the systems we analyzed.

Sequential consistency When the replicas consistency is ensured to be sequential, a client connecting to any of them is guaranteed to see the outcomes of performed operations as if they were executed in some sequential order.

Eventual consistency By eventual consistency, all the replicas are guaranteed to receive sooner or later the updates performed to the data at any of the replicas. When no new write operation is performed over a datum, the following reads eventually will return the last version of the value.[PV16] To implement eventual consistency synchronization mechanism deterministic re-ordering technique are employed to obtain convergence of simultaneous updates. One strategy is called last-write-wins (LWW) and it acts by tagging with a temporal identifier the latest write made. Since no global clock exists this identifier is usually constituted by the combination of nodes' ids and their logical clocks. Another approach to reach replicas' convergence is to express data modifications by means of commutative operations and broadcast them to all the replicas. This approach is employed by conflict-free replicated data type (CRDT) to solve replication consistency issues, still providing the possibility to update independently and concurrently any of the replicas without coordination.

Communication

We define which are the guarantees related to the communication and in particular that are provided with respect to the communication channels described in the operational model in section 3.2. The transmission of a data item through a communication channel is fundamental to be reliable for the correctness of overall execution. This is where communication guarantees comes into play. When we consider fault-tolerant systems we also have to account for failure and subsequent recovery procedures. No part of the communication can be assumed fully reliable or behaving the way we expect. Hence none of these is immune from failures, also the client can in fact crash too. The number of workers within a cluster can be very large and along with this number grows the likelihood of hardware failures both of the servers and of the network facilities required to connect them.

We below describe the existing levels in terms of the number data item copies that is guaranteed to be delivered. Moreover, we describe the ordering guarantee that analyzed system enforce in the communication channels.

Delivery The sending of a data item may not be followed by the receipt of it on the other side; the communication channel could crash in the meantime. Efforts in development of reliable communication channels capable of tolerating faults on all three fronts such as client, channel itself and server have led to the clear distinction of three types of delivery guarantees.

At least once. This guarantee provides the guarantee to deliver the data item to the recipient at least once. This leaves open the possibility that more than one copy of the same data item is delivered to the recipient. This anomaly arises from the situation in which a crash occurred in the communication channel and the recovery procedure, seeing a pending data item, delivers it one more time. Another condition in which it may happen this phenomena is when the client does not receive confirmation his data item has been received and therefore decides to send it again.

At most once. In this situation the delivery of the data item is guaranteed at most once, thus leaving open the possibility that the data item will never be delivered. This case can occur when a client send a data item to a recipient that suddenly crashes after sending back the confirmation of receipt. When the recipient resumes from crash it will be like the data item has never been received.

Exactly once. This level derives from the sum of the two previous guarantees which, put together, make up the constraint that a data item sent must be received exactly once. This type of guarantee has been proven to be impossible to achieve in the more general sense. Systems claiming to support it do so under the assumptions of configurations made by means of other reliable systems.

Ordering Several servers within a distributed system communicate and coordinate each others through the exchange of data items. Ordering of received massages affect decisions and produced results. Therefore it is of fundamental importance to define whether or not ordering between data items is guaranteed to be preserved. We define the ordering of data items in relation to the one preserved in the communication channels describe in the operational model section 3.2.

FIFO The *First In First Out* policy requires that if a set of data items is sent from the sender in some order then the same order is the one seen by the receiver(s). In this case the ordering at the receiver is the one of the sender but potentially it is interspersed with other data items from different senders.

FIFO by key The more general FIFO guarantee can be weakened by specifying the boundaries within which it is provided. Distributed data-intensive tools employs the technique of partitioning to split large data sets into sub portions. In turn, these are exploited both to physically arrange the data on multiple nodes and to maintain data items that are related to each other together in the same partition. However, the elements within a set can be moved to different partitions due the transformation performed that might rearrange them with a different criteria. In the rearrangement, if the original order of the source partition is preserved we say that the system ensure FIFO by key ordering.

3.5 Guarantees' implementations

The following characterization captures the details about design implementations adopted by systems to enforce guarantees. We provide an abstraction and modeling frame to localize approaches and protocols utilized to accomplish the claimed guarantees.

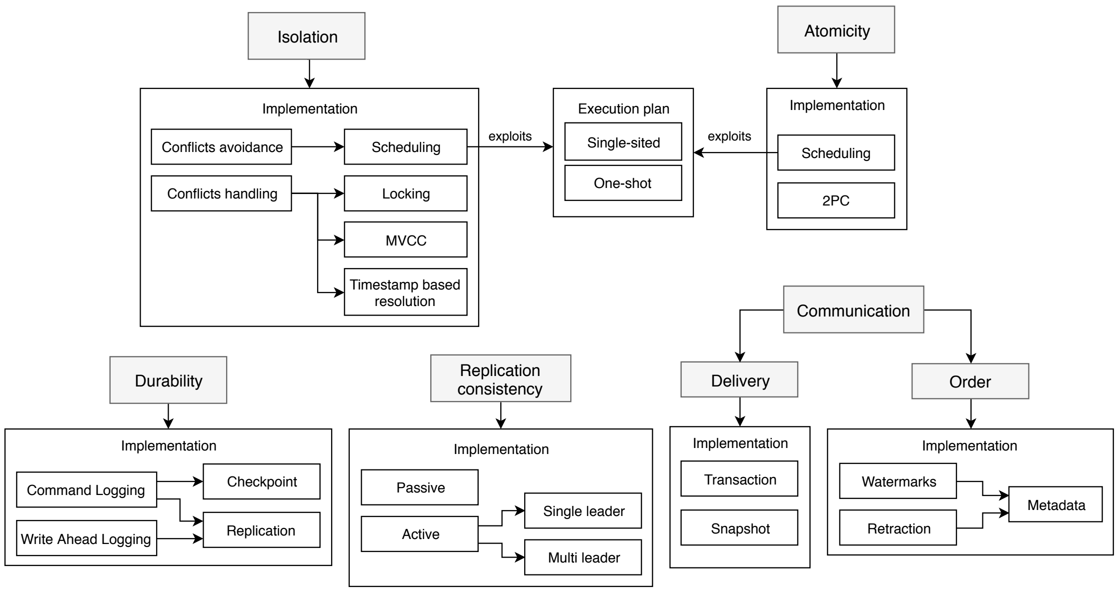


Figure 3.6: Guarantees' implementations model

Atomicity

The atomicity guarantee can be implemented either by coordination-free approach such as scheduling or via the coordination-based method of 2 phase commit (2PC) protocol.

According to the types of operations performed and data accessed by a transaction, we report the classification made by [SMA$^+$07] for types of execution plans:

- **Single-sited** The scheduling can ensure that a transaction takes place on a single physical node, which can autonomously determine the outcome of that transaction.

- **One-shot** The scheduling can decompose the transaction body into a set of execution portions that the nodes can autonomously and safely execute without coordination.

When all the transactions fall either in the class single-sited or in one-shot then each of them can be sent to the related replica and and executed. This will ensure that copies of the transaction execution sent to all the replicas will either commit or abort all. Locally, each of them will be confident that other parallel instances will end up with the same outcome.

However, when the scheduling is not an option due to the transaction structure complexity, coordination-base methodologies have to be employed to ensure atomicity. The two phase commit (2PC) is a distributed algorithm that is employed to coordinate parallel instances in the execution of atomic transactions. Its design ensure the effectiveness in achieving the goal even in case of failure (in most of the situations no manual intervention is required). It consists of two phases:

1. The *voting phase* requires a coordinator to prepare all the sites involved in the transaction to take necessary steps in either committing or aborting. In this step, all parties involved are called to vote for either commit or abort based on the their local execution outcome.

2. In the *commit phase* instead, the coordinator which collected all the votes from the participants decides whether to allow or not the distributed commit. In the situation in which even only one of the parties involved voted for abort, all the others are asked for abort too. If instead, the voting ended up in all the parties agreeing on commit, then the coordinator communicate the outcome and the commit take place.

Isolation

There are two main approaches to provide isolation guarantee. At first, operations can be scheduled so that, even if executed concurrently, they do not interfere with each other originating anomaly phenomena. In this case, a careful scheduling is required and it leverages the same technique described for atomicity based on the type of execution plan required to perform the transaction.

The other approach to isolation is conflict handling. In this case, different methods can be used to avoid the presence of anomalies derived by simultaneous accesses to the same data.

Locking Locking is a synchronization mechanism that enforces access limits of a resource in scenarios where there are multiple contenders. In particular, the locking technique is applied by means of the 2 phase locking protocol (2PL) that is a concurrency control method to provide the serializable isolation level.

The protocol allows transactions to acquire locks over modified resources so to deny concurrent accesses by other transactions. Its behavior can be summarized in two phases:

1. With the *expansion phase* locks can only be acquired and none of them can be released.

2. The *shrinking phase* instead allow only to release locks and once started no locks can be acquired again by the transaction.

MVCC The Multiversion Concurrency Control is a method employed to provide concurrent access to the versions of the same resource. This concurrency control method provides consistent views with respect to points ”in time”. Indeed, if a transaction is reading some data that gets modified in the meantime by another simultaneous operation, it could happen that the transaction will read half-modified pieces of information. To solve this problem one possibility is to make wait the write operation for the read to finish, but in this case, we would be using a locking mechanism as described above. Instead, if we choose to make the read continue in its process, we can save the new changes from the write as a ”new version” of the same data. Later reads will then read the new version but concurrent one will still continue to read a consistent version. MVCC is indeed employed to provide the snapshot level of isolation.

Timestamp-based resolution The timestamp-based approach uses timestamps to serialize the concurrent executions of transactions. This algorithm ensures that in case of conflicting reads and writes their execution happens in timestamp order. Older operations are always given higher priority and hence are executed at first. Opposed to locking approaches, that manages the order between the conflicting transactions before their execution, the timestamp-based approach handle conflicts as soon as they are detected. This strategy employs timestamps to verify whether modified data are causing anomaly phenomena. When no conflict is detected transactions are allowed to commit and conclude their execution successfully. On the contrary, if by checking timestamps of modifications, the system detects a conflict, one of the transactions is immediately aborted.

Durability

Data durability can be guaranteed with two diverse strategies Command Logging [MWMS14] or Write Ahead Logging [MHL+92]. Command Logging approach acts by storing a copy of the performed operations with their input data. This fast method has a very low footprint in the execution but has the side effect to require a bit more of computation in the recovery procedure after a failure. In fact, when it is required to restore a state the previously saved operations are replayed so to reproduce what happened in the same order and obtain the same final result. However, since histories of commands can very too long frequent snapshots of the system state have to be performed in addition to this method so to flush older parts of execution histories. To implement the command logging strategy both checkpointing and replication are adopted.

Write Ahead Logging is a technique employed to provide durability as it is Command Logging but acts in a different way. It focuses on the state of the system instead of the commands executed to change it. All modification made by operations are first recorded into a log and then they gets applied to the state of actual system state. It is implemented by means of data replication.

Checkpoint The checkpointing technique consists of saving a snapshot of the system state every predefined amount of time or in case of direct requests for it. This approach is employed to provide fault tolerance and hence data durability. Applications or processing portions can resume execution from the last point that was saved before the failure. This feature is crucial for the long-running computation.

Replication Data replication is used to ensure durability by creating multiple copies the same information. In case one of the replicas is lost due to a hardware or software failure, it gets recreated to a healthy location so to ensure tolerance for the next eventual failure. The data replication should be transparent by external users but it is internally exploited to achieve higher performance by means of simultaneous data accesses.

Replication consistency

Two replication strategies exist. With active replication, replicas perform read or writes operation and based on which of them is allowed to carry out write operations we distinguish the single or multi-leader types.

Instead, the strategy of passive replication acts by making plain duplicates of data sets to guarantee durability in case of failure. This technique creates backup copies that later can be used during recovery procedures to restore the system state.

Single leader In this configuration, only one of the replicas, the leader, can accept and execute write operations to change data. All the other replicas are allowed only to execute read operations thus favoring higher performance.

Multi leader With multi leader replication, more than one replica is allowed to execute both read and write operations. This implementation is more suitable for applications with high rates of write operations that might tolerate weaker consistency constraints.

Communication

In a centralized system software processes can exchange information with each other making the assumption of fully reliable communication channel .In fact, this supposition turns out to be valid as long as the system does not present any fault. In the latter situation the processes would also be interrupted as well and they would not communicate further. In a distributed scenario no reliability assumption can be made about the communication channel. Data-intensive tools implement techniques and protocols to provide more reliable communication between distributed portions of the processing.

Communication delivery

The delivery of a sent data item is not obvious if the communication channel is subject to breakdowns. As it is not, the delivery of a single copy of the sent data item in case the strategies adopted in failure recovery do not consider for the re-delivery issue of the same data item.

Transaction The mechanism employed to guarantee a single and exact delivery of one data item is the transaction. The coordination between sender and receiver is fundamental to guarantee a data item has been sent and actually received. By means of transactions it is possible to detect whether a sent data item has been actually delivered to, processed and saved by the receiver. The latter would indeed commit the transaction confirming the successful outcome only after it has the guarantee that data item has been processed or saved in a fault tolerant way. In the case it crashes in the middle of receiving a data item and committing the system can infer the delivery is not succeeded. Moreover, the same deduction is done in case of the communication channel fails and the data item get lost or in case the sender crashes.

Snapshot By means of distributed snapshots, the data-intensive tools are able to understand in case of failure which where the data items sent and that potentially went lost in failures. In this situation, the recovery procedure restores the state saved before and replays the data items marked as sent but for which there is no evidence of delivery. However, this method does not guarantee alone that a data item would not be delivered twice in case of receivers' crashes before the next snapshot.

Communication order

Data and coordination data items are sent through the network to allow workload execution in a distributed fashion. However, the data items sent are subject to the routing mechanisms of the network infrastructures. This means that the order of data items at the sender is not guaranteed to be the same seen by receiver. The packets' congestions, as well as network hardware failures and recovery, can cause part of the sent data items to take different paths and arrive at the destination no more ordered the same way they were sent.

Watermarks Stream processors at first leveraged the concept of watermarking. They need to handle data coming from multiple sources that, generating data, send them through the network. In many cases, it happens that the transferring procedure introduces random delays in the journey of a data item. The receiving of datum at a later time with respect to the currently received data items is known as a *late arrival*. A watermark is a particular data item flagged with a timestamp which indicates that no other older data has been generated after it. Thanks to its content, when a watermark is received the system is notified that all late arrivals with respect to the timestamp transported have not be considered. The sequential reception of watermarks marks the flow of what are called epochs.

Retraction Ordering of received data influence produced results. This fact is even more true when the transformations and processing performed is base on the event time and late arrivals occur. We remind that a the late arrival phenomena happen when, due to network-introduced delays, a data item arrives longer later with respect to when it was generated. With the retraction technique, emitted results can be treated again to fix imprecise or partial outputs by including data items arrived later. The downstream system to the one that employs retraction has to support this technique. Indeed, it has to cope with the possibility that an input, coming from the upstream processing, can be changed at any time to improve its correctness. The implementation of this mechanism involves the attachment of further information, the metadata, to a result. Metadata allows systems receiving processed data items to understand whether they are meant to be a result, an update, or a delete operation for the already received outputs.

Chapter 4

Methodology

The goal of this project is to characterize data-intensive tools in terms of adopted computational paradigms and provided functionalities expressiveness. During our study, we identified critical scenarios that analytical tools, such as stream and batch processors, have a lot of difficulties to implement. These type of data-intensive tools are indeed mainly able to materialize asynchronously computed results. They have very limited capabilities and provided minor features in satisfying on-demand processing requests. On the contrary, the databases excel in the support of synchronous iterations such as queries; they are less efficient in stream processing operations even though they can perform them.

Since the objective of this thesis work is not to evaluate performance, we mainly directed our focus on the deeply understand of expressiveness limitations by reproducing with other tools an application case.

We decided to re-implement a demonstrative application taken from the examples collection of VoltDB[1]. In this section, we will present some of the experiments conducted and the encountered limitations. We will also explain the solutions we employed to adapt the original application architecture to different functioning mechanisms.

4.1 The Metrocard application

The scenario we used employed was take from the VoltDB demonstrative examples. The Metrocard application manages the entrance service of the underground turnstiles of a hypothetical metropolitan transportation company. With respect to the database schema, the data entities defined are the following.

Station The *station* entity saves the name of the station and the fare amount that must be charged to the user for the entrance to the underground at that location.

Card The *card* is representative for a user in the system and holds the residual money amount the user has. One of the field characterize the nature of payment plan the card is subject to. At first the card can be simply used in pay per use mode. Each time a swipe is attempt to enter a turnstile the entrance will be permitted only if the card residual amount is greater than or equal to station fare. The second payment plan is subscription. Users are always

[1] `https://github.com/VoltDB/voltdb/tree/master/examples`

allowed to enter any metro station, regardless of the fare, until the subscription period expiration is not over.

The benchmark crafted by the example simulates high velocity transaction processing for the metro cards entering multiple metro stations. A set up stage before the benchmark execution is in charge to create 50.000 fake cards and to load the 60 stations so to have an initial state. Then, the benchmark execution is made of transactions with different objectives. At first, the majority of them tries to swipe randomly picked cards at arbitrary stations simulating the users' entrances attempts. This submit to the service requests for swipes that can receive either a positive or negative responses. When a card holds a valid subscription plan or has sufficient credit to enter the station the application reply with a positive answer. Instead, when either these conditions ar not met or the card (randomly picked) does not exist a negative answer is sent back. Lastly, to simulate a more realistic scenario, some transactions are in charge to, during the benchmark, create new cards, top-up some of them with credit replenishes as also submit renewals of the subscriptions.

4.2 The different interaction mode

The interaction with an asynchronous execution system, like stream processors are, require different interaction ways with respect to synchronous call. An application interacting with a database usually saves and reads data through direct requests. This approach outlines an active, synchronous and request-based interaction method that makes the application wait for the expected answer as a reply to the request submitted. However, asynchronous systems do not behave in the same way. The sending of a request, is not necessarily immediately followed by an answer. The system takes note of the request, perform the processing and later it makes available the answer for the client to be read. But, it is up to the client again act in the first place and read the result. The change in the interaction modality described above is what had lead to the first failure in the attempt to replicate the same synchronous approach of the database application on an asynchronous system. The original queries had to be reshaped in a different form so to be sent as data to the system. We have tried as much as possible to do not change the application semantics by the queries' conversion. We materialized the requests by emulating the invocations with single data-items. In turn, each one of them had to produce as output the result expected by the originally submitted queries.

4.3 Apache Flink experiments

Static load of data

The first problem we have to solve was to load the initial data employed by the benchmark such as stations and cards. These entities are loaded in the preparation stage and are then used by the simulation to generate a workload in the system.

With some of the data-intensive tools it was possible to load directly from file the stations. For example, in the case of Flink we tried to exploit the provided *Table API*.

```java
final String filename = params.get("filename", "data/stations.csv");
[...]
// Source
final TableSource csvSource = CsvTableSource
        .builder()
        .path(filename)
        .field("station_id", Types.SHORT)
        .field("name", Types.STRING)
        .field("fare", Types.SHORT)
        .field("weight", Types.INT)
        .build();
tableEnv.registerTableSource("stations", csvSource);

Table orderedById = tableEnv.scan("stations").select("*").orderBy("id");

DataStream<Row> dataStreamStations = tableEnv.toAppendStream(orderedById, Row.class);
dataStreamStations
        .map(t -> Tuple2.of(0, t.toString()))
        .returns(TypeInformation.of(new TypeHint<Tuple2<Integer, String>>() {
        }))
        .keyBy(0)
        .map(new StationEntity.MapToQueryableTable());
```

Figure 4.1: CSV file loading by means of Flinks' Table API

The attempt shown in **??** was created to load the data of stations ans so to perform event enrichment. Databases employ the SQL *join* operation to fetch results by linking related data to each others. Unfortunately, besides what is stated by the documentation of Flink, Table APIs are not fully interchangeable with Stream APIs since we encountered the runtime error reporting that "*OrderedTables can not be converted to appendStreams*"

After this first attempt we needed to change the way stations were "statically" loaded. A simple yet effective solution was to send the stations as a data stream before starting the benchmark. They can be saved by internal variables state and it would be as if they were statically loaded.

```
Iterator<Station> stationIterator = getStations();

while (stationIterator.hasNext() && this.isRunning) {
    Station station = stationIterator.next();

    // add the station to the random collection for later
    stations.add(station.getWeight(), station.getStation_id());
    max_station_id = Math.max(max_station_id, station.getStation_id());

    ctx.collect(Tuple3.of(STATION, "INSERT", station));
}

System.out.println("Stations loaded.");
System.out.println("Max station id is: " + max_station_id);
```

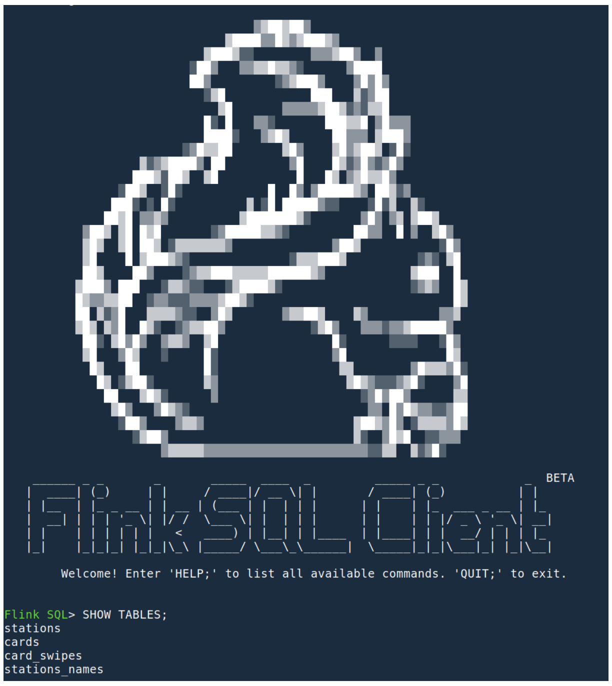
Figure 4.2: Stations loading as stream in DataSource class

This approach was successful and allowed us to stream the stations entity into the system.

Flink SQL client

In the Flink project, a tool called SQL client allows submitting pure SQL queries to the cluster. This standalone client runs within the shell and leveraging the Flink's Table & SQL API it executes queries written in SQL language. However, its functioning is based on a table program that must be written either in Java or Scala. Moreover, a configuration must be provided with the execution of the tool to declare to the systems source and sink tables and the schema they have.

Despite the strong limitations that can be inferred by its documentation page, we decided to investigate this tool anyway. We wanted to understand if it would have been useful in reproducing the Metrocard application by means of the same queries adopted in the original scenario.



```
                                                                    BETA

        Welcome! Enter 'HELP;' to list all available commands. 'QUIT;' to exit.


Flink SQL> SHOW TABLES;
stations
cards
card_swipes
stations_names
```

Figure 4.3: Flink SQL client interface

We managed to load static tables by means of the required configuration. Very simple queries were possible and we managed to perform some example of user-defined function from the documentation. However, this tool turned out to be quite limited provided functionalities and expressiveness of queries that were supported. In many situations, we got errors reporting that the query was not still supported, thus suggesting this tool is still very limited and in too early development stage.

```
[ERROR] Could not execute SQL statement. Reason:
org.apache.flink.table.api.TableException: Cannot generate a valid execution plan for the given query:

FlinkLogicalSort(fetch=[10])
  FlinkLogicalTableSourceScan(table=[[default_catalog, default_database, card_swipes]], fields=[card_id,
station_id], source=[CsvTableSource(read fields: card_id, station_id)])

This exception indicates that the query uses an unsupported SQL feature.
Please check the documentation for the set of currently supported SQL features.
```

Figure 4.4: Error returned in the execution of a query due to unsupported SQL statements.

Queriable operator state

We wanted to understand capabilities in a new functionality of Apache Flink. This features-rich stream processor allows, by mean of specific APIs, a client not belonging to the execution cluster to query an operator state.

With this experiment we test the feasibility of synchronous interaction with the deployed operators. The idea was to save the cards state to single operators and change it by sending "materialized" queries as input data items.

```java
public class StationsListState extends RichMapFunction<Station, Station> {
    private transient ValueState<Long> count;

    @Override
    public void open(Configuration config) {
        ValueStateDescriptor<Long> descriptor =
                new ValueStateDescriptor<>(
                        "count", // the state name
                        TypeInformation.of(new TypeHint<Long>() {
                        }));
        descriptor.setQueryable("query-name");

        count = getRuntimeContext().getState(descriptor);
    }

    @Override
    public Station map(Station value) throws Exception {
        if (count.value() == null) count.update(0L);

        count.update(count.value() + 1);

        return value;
    }
}
```

Figure 4.5: The *map* function class to implement queryable state variable

The code above reported show the implementation of the *map* function. Each station on which the map function is applied increments a counter. The latter will be queried when the client connects to this operator.

```
final SingleOutputStreamOperator<Station> stationDataStream = source
      .filter(new FilterFunction<Tuple3<String, String, byte[]>>() {
          @Override
          public boolean filter(Tuple3<String, String, byte[]> value) throws Exception {
              return value.f0.equals("STATION");
          }
      })

      .map(new MapFunction<Tuple3<String, String, byte[]>, Station>() {
          @Override
          public Station map(Tuple3<String, String, byte[]> value) throws Exception {
              return (Station) getFSTInstance().asObject(value.f2);
          }
      })

      // key by a fake key in order to have a "single partition" state
      .keyBy(v -> 0)

      .map(new StationsListState());
```

Figure 4.6: The *map* function applied on the input stations stream

When new data streams are sent as input they could be split up into multiple partitions. Flink could decide to apply data-parallelism on the stream and by this the idea of a single operator holding all the stations wouldn't work. To solve this issue, we forced the system to create a single partition and thus to save all the stations to only one site. This is exactly the role of the function *keyBy* in the above code. Lastly, instead of saving the stations that would have required to implement complex data structures extending Flink's APIs, for the sake of pure testing we saved only the count of stations.

After describing how the streaming application had to manage the state, we implement the client to query it.

```
JobID jobID = JobID.fromHexString(jobId);

QueryableStateClient client = new QueryableStateClient(host, port);

// the state descriptor of the state to be fetched.
ValueStateDescriptor<Long> descriptor =
  new ValueStateDescriptor<>("count",
                             TypeInformation.of(new TypeHint<Long>() {}));

CompletableFuture<ValueState<Long>> resultFuture =
        client.getKvState(jobID, "query-name", 0, BasicTypeInfo.INT_TYPE_INFO, descriptor);

// now handle the returned value
System.out.println(resultFuture.get().value());
```

Figure 4.7: The client performing the query to the ValueState variable of the operator

Flink queriable state works by referencing the exact operator the state is stored by. For this reason, it is required for the client to now the JobID that correspond to the Flink's JobManager holding the execution of the operator's task. With the identifier, only available at runtime when the *map* function is deployed, a client can submit the request to fetch a state by stating the "query name". This parameter is used to identify which state query among the possible many present on a single operator.

Unfortunately, this mode of interaction in querying an operators' state was not scalable enough and with sufficient ease to fully implement the Metrocard application. It would have required long and inefficient workarounds to retrieve by the client entities stored by an operator. In fact, the serialization and deserialization procedures required to transmit programming objects through the network would have been very complex to implement for the stations and cards. Moreover, the querying approach wouldn't have scaled properly to be comparable with the VoltDB application performance. The client, after querying, would have been required to implement by itself the benchmark and send the new "materialized" queries as data items to the streams.

The synchronous interaction with the external environment could not be feasible for common applications' implementations. We decide to move to another approach adapting the application functioning to work in an asynchronous matter. Query invocations have been materialized as input events to the system. They are handled thanks to a filtering applied on the generic data structure by which different types of entities can be distinguished. Moreover, in order to employ this method, we had to make a further adaptation due to the APIs implementation constraints. Flink applications require to know which are the input data structure in order to correctly deserialize them. Our testing scenario have different data structures since they can be either Stations, Cards or materialized queries like CardSwipes, ReplenishCard. The problem of sending heterogeneous formats through a standardized channel was solved by utilizing a constant generic structure. By means of a *Tuple3*, a simple but customizable type, we have been able to send diverse entities to the system that then were deserialized thanks to the information carried by the first field of the tuple.

For static data loading, in this scenario we employed the method described earlier of stream into the system data before starting the benchmark. In the below reported case, this approach has been adopted for Stations and Cards loading.

The broadcast stream and state operator We investigate more on how to perform the event-enrichment activity in the most generalized way as to conform the database capability to do so. By Flink functionalities we discovered the broadcast stream feature. This API allows to specify a data flow to be sent in broadcast to all the operators. Then, the stream on which the broadcasted one is connected can be processed by a particular map function able to handle both broadcasted items and regular ones. The stations, since broadcasted are received by all the operators which in turn store them in a Map data structure. Then when a CardSwipe is received, made the assumption that stations have been streamed before any other data, the map function performs the event-enrichment by attaching the station fare attribute to the CardSwipe.

```java
// stations data streams
final DataStream<Station> stationsStream =  source
    .filter(v -> v.f0.equals(STATION))
    .map(v -> (Station) v.f2);

// main data stream: source - stations
final DataStream<Tuple3<Entity.TYPE, String, Entity>> sourceStream = source
    .filter(v -> !v.f0.equals(STATION));

 /** Broadcast the station stream **/
// a map descriptor to store the id of the station and the station itself.
MapStateDescriptor<Long, Station> stationStateDescriptor =
        new MapStateDescriptor<>("StationsBroadcastState",
                                 BasicTypeInfo.LONG_TYPE_INFO,
                                 TypeInformation.of(new TypeHint<Station>() {}));

// broadcast the rules and create the broadcast state
BroadcastStream<Station> stationsBroadcastStream = stationsStream
    .broadcast(stationStateDescriptor);


/** Enrich the main stream **/
final KeyedStream<Tuple3<Entity.TYPE, String, Entity>, Long> sourcePartitionedByCardId = sourceStream
    .keyBy(new KeySelector<Tuple3<Entity.TYPE, String, Entity>, Long>() {
        @Override
        public Long getKey(Tuple3<Entity.TYPE, String, Entity> t) throws Exception {
            switch (t.f0) {
                case CARD:
                    return ((Card) t.f2).getCard_id();

                case CARD_SWIPE:
                    return ((CardSwipe) t.f2).getCard_id();

                case REPLENISH_CARD:
                    return ((ReplenishCard) t.f2).getCard_id();
            }
            return 0L;
        }
    });


final DataStream<Tuple3<Entity.TYPE, String, Entity>> enrichedSourceStream = sourcePartitionedByCardId
    .connect(stationsBroadcastStream)
    .process(
        // type arguments in our KeyedBroadcastProcessFunction represent:
        //   1. the key of the keyed stream
        //   2. the type of elements in the non-broadcast side
        //   3. the type of elements in the broadcast side
        //   4. the type of the result, here a string

        new KeyedBroadcastProcessFunction<Long,
                                          Tuple3<Entity.TYPE, String, Entity>,
                                          Station, Tuple3<Entity.TYPE, String,
                                          Entity>>() {
              // indentical to our stationStateDescriptor above.
              // In the map, stations are stored by their id and constitute the state.
              private final MapStateDescriptor<Long, Station> stationStateDescriptor =
                      new MapStateDescriptor<>("StationsBroadcastState",
                                               BasicTypeInfo.LONG_TYPE_INFO,
                                               TypeInformation.of(new TypeHint<Station>() {}));

                          @Override
                          public void processBroadcastElement(Station station,
                                                              Context ctx,
                                                              Collector<Tuple3<Entity.TYPE, String, Entity>> out)
                          throws Exception {
                              ctx.getBroadcastState(stationStateDescriptor).put(station.getStation_id(), station);
                          }

                          @Override
                          public void processElement(Tuple3<Entity.TYPE, String, Entity> value,
                                                     ReadOnlyContext ctx,
                                                     Collector<Tuple3<Entity.TYPE, String, Entity>> out)
                          throws Exception {
                              if (value.f0.equals(CARD_SWIPE)) {
                                  CardSwipe cardSwipe = (CardSwipe) value.f2;

                                  Station station = ctx
                                    .getBroadcastState(stationStateDescriptor).get(cardSwipe.getStation_id());

                                  ((CardSwipe) value.f2).setStation(station);
                              }
                              out.collect(value);
                          }
        });
```

Figure 4.8: Station stream broadcasted and event-enrichment performed on CardSwipes.

After the stations load, the next problem was to save the card as system state. Stream processors do not provide explicit functionalities to handle this concept. We decided to store the card entity by means of a variable in the map function scope. In doing so, the consequence is that there must be one operator holding one card and as a consequence it is of fundamental importance to let the system know to do not send two different cards to the same operator. We achieve this objective by partitioning the input stream using as key the card id. This strategy was very helpful and effective also to send the queries intended to modify the card state (such as CardSwipes) towards the correct operator. Indeed all the entities, made exception for stations, are directed towards parallel instances of the card state operator according to the card id.

```java
/** Do computation as if it was a Database receiving requests and
    emitting results/intermediate computations **/
final SingleOutputStreamOperator<Tuple3<Entity.TYPE, String, Entity>> partialComputation1 = source
    .keyBy(new KeySelector<Tuple3<Entity.TYPE, String, Entity>, Long>() {
        @Override
        public Long getKey(Tuple3<Entity.TYPE, String, Entity> t) throws Exception {
            switch (t.f0) {
                case CARD:
                    return ((Card) t.f2).getCard_id();

                case CARD_SWIPE:
                    return ((CardSwipe) t.f2).getCard_id();

                case REPLENISH_CARD:
                    return ((ReplenishCard) t.f2).getCard_id();
            }

            return 0L;
        }
    })
    .flatMap(new CardState());
```

Figure 4.9: The handling of materialized queries by means of the map state function and emission of results.

```java
public class CardState implements
FlatMapFunction<Tuple3<Entity.TYPE, String, Entity>, Tuple3<Entity.TYPE, String, Entity>> {
    // The state of the card saved in this map
    private Card card = null;

    @Override
    public void flatMap(Tuple3<Entity.TYPE, String, Entity> value,
    Collector<Tuple3<Entity.TYPE, String, Entity>> out) throws Exception {
        String action = value.f1;
        Entity entity = value.f2;

        switch (value.f0) {
            case CARD:
              handleCard(out, action, (Card) entity);
              break;
            case CARD_SWIPE:
              handleCardSwipe(out, action, (CardSwipe) entity);
              break;
            case REPLENISH_CARD:
              handleReplenishCard(out, action, (ReplenishCard) entity);
              break;
        }
    }


    /** Handle the arrival of an entity of type: card **/
    private void handleCard(Collector<Tuple3<Entity.TYPE, String, Entity>> out,
    String action,
    Card card) {
        if (action.equals("INSERT")) {
            this.card = card;
            out.collect(Tuple3.of(RESULT,
            CARD.toString(),
            new Result(1, "Card saved.", card.toString())));
        }
    }

    /** Handle the arrival of an entity of type: CardSwipe **/
    private void handleCardSwipe(Collector<Tuple3<Entity.TYPE, String, Entity>> out,
    String action,
    CardSwipe cw) {
        [...]

        if (card == null) {
            // no card has the id of the swipe
            out.collect(Tuple3.of(RESULT,
            CARD_SWIPE.toString(),
            new Result(0, "Card Invalid.", cw.toString())));
            return;
        }

        if (card.getEnabled().equals(0)) {
            // the card of the swipe is not enabled
            out.collect(Tuple3.of(RESULT,
            CARD_SWIPE.toString(),
            new Result(0, "Card Disabled.", cw.toString())));
            return;
        }

        // read card values
        [...]

        // get the station
        Station station = cw.getStation();
        if (station == null) {
            out.collect(Tuple3.of(
            RESULT,
            CARD_SWIPE.toString(),
            new Result(0, "Station id not found.", cw.toString())));
            return;
        }

        // read station values
        Long fare = station.getFare();
        String stationName = station.getName();

        Timestamp currentTime = new Timestamp(new Date().getTime());
```

Figure 4.10: The *map* function employed to store the card state and execute materialized queries – part 1.

```
    // check balance or expiration for valid cards
    if (cardType == 0) { // pay per ride
        if (balance > fare) {
            // charge the fare
            card.setBalance(balance - fare);

            // create a new Activity
            out.collect(Tuple3.of(
            ACTIVITY,
            "INSERT",
            new Activity(card.getCard_id(), currentTime, cw.getStation(), 1L, fare)));

            // return the result
            out.collect(Tuple3.of(
            RESULT,
            CARD_SWIPE.toString(),
            new Result(1, "Remaining Balance: " + card.getBalance() / 100, cw.toString())));
            return;
        } else {
            // insufficient balance
            // create a new Activity
            out.collect(Tuple3.of(
            ACTIVITY,
            "INSERT",
            new Activity(card.getCard_id(), currentTime, cw.getStation(), 0L, 0L)));

            if (notify != 0) {  // only export if notify is 1 or 2 -- email or text
                out.collect(Tuple3.of(
                CARD_ALERT_EXPORT,
                "INSERT",
                new CardAlertExport(card.getCard_id(),
                currentTime.getTime(),
                stationName, owner, phone, email, notify, "Insufficient Balance")));
            }
            out.collect(Tuple3.of(
            RESULT,
            CARD_SWIPE.toString(),
            new Result(0, "Card has insufficient balance: " + (balance / 100), cw.toString())));
            return;
        }
    } else { // unlimited card (e.g. monthly or weekly pass)
        [...]
    }
  }


  /**
  * Handle the arrival of an entity of type: ReplenishCard **/
  private void handleReplenishCard(Collector<Tuple3<Entity.TYPE, String, Entity>> out,
  String action,
  ReplenishCard rc) {
  /*
  CREATE PROCEDURE ReplenishCard PARTITION ON TABLE cards COLUMN card_id PARAMETER 1 AS
  UPDATE cards SET balance = balance + ?
  WHERE card_id = ? AND card_type = 0;
  */
  // replenish only cards of type 0
  if (this.card.getType().equals(0)) {
      this.card.setBalance(this.card.getBalance() + rc.getBalance());

      // send out result
      out.collect(Tuple3.of(
      RESULT,
      REPLENISH_CARD.toString(),
      new Result(1, "ReplenishCard", rc.toString())
      ));
  }
}
```

Figure 4.11: The *map* function employed to store the card state and execute materialized queries – part 2.

Statistics computation Each CardSwipe attemp in the Metrocard application leave a trance of its outcome. A table *Activities* record all the logs of successful and unsuccessful card swipes. Then by means of aggregation views over this table, some statics such as number of swipes per second are calculated. The VoltDB SQL dialect allows with a special syntax to express conditional constructs in the evaluation of some columns during group operations. This feature was employed also in the computation of statistics and brought some additional complexity during the reimplementation by means of queries over streams. We processed the resulting stream by leveraging interoperability of Stream APIs and Table APIs. It has been useful indeed to retrace the same operations done by VoltDB in SQL by means of equivalent APIs in Flink.

The following code snippet demonstrate the statistics computation.

```
Table entries = tableEnv
   // LIMITATION: it's very important to do a select and where before doing the groupBy.
   // Field of the filter condition cannot be accessed in a where after a groupBy!
   .scan("activity")
   .select("*")
   .where("activity_code = 1")

   .groupBy("station_name, second")
   .select("second, " +
           "station_name, " +
           "card_id.count AS activities, " +

           // Activity code = 1: entries
           "activity_code.count AS entries, " +
           "amount.sum AS entry_total, " +

           // Activity code = 2: purchases
           "0L AS purchases, " +
           "0L AS purchase_total");
```

Figure 4.12: Calculation of statistics over results stream

Results

After the benchmark re-implementation we tested its performance. The simulation has been run for 1 minute in order to have a mean value of the operations throughput.

The resulted throughput is 203704 operations performed per second. The value is comparable to the average one from the execution of VoltDB benchmark. However, in this case, no transactional guarantee is provided by Flink since it lacks of their support. VoltDB instead provides throughput values measured by transactions / sec which could be considered more valuable since the stronger execution guarantee of transactions.

4.4 Apache Spark experiments

Spark provide a set of APIs called “structured streaming” to implement stream functionalities by means of micro-batching. We tried even for this system to replicate the Metrocard app example so to highlight expressiveness and feasibility limitations. We adapted the meaning of procedures done in the case of VoltDB application to the one offered by Spark.

In the testing made with Spark we employed the same compromised adopted in Flink. Since the asynchronous nature of stream processing we decided once again to materialize queries to the system as if they were input data.

Structured streaming data frame

The data structure abstraction employed in the structured stream APIs is the data frame. It resemble the databases' table concept but improving the flexibility and supporting streaming-related notions. This tabular frame can have in its columns sub structures that still are treated as tables. Thus, it is possible to have sets of nested columns as type of an outer column.

However, its flexibility and expressiveness gets limited by the complexity in automatic de-serialization of heterogeneous structures such as those we want to send as input to the system. Every stream processor, indeed, tries to understand the data format it receives by, when stated, trying to deseriliaze received data. In our situation we need to send different entities through the input stream and, as occurred in Flink, some issues mined our work in doing so. The first idea has been to exploit nested structures to send entities as did in Flink as generic types and, later in the processing, describe to the system how to handle these data structure. However, each item in data frames is deserialized entirely in one single process when it is employed in transformations. Due to this constraint in the functioning of this functionality we decided to flatten the entities schema into one single table reporting the columns of all of them. An additional column was employed to recognize which kind of entity did correspond to the row.

```
val source = spark
  .readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()
  .selectExpr("CAST(value AS STRING)")

/** Map the EntityLine fields to the dataframe columns */
val entitiesDf = source
  .select(from_json($"value", getSchemaOf[EntityLine]) as "data")
  .select("data.*")

entitiesDf.printSchema()
/*  root
     |-- entity_name: string (nullable = true)
     |-- action: string (nullable = true)
     |-- station_station_id: long (nullable = true)
     |-- station_name: string (nullable = true) [...]
     |-- card_balance: long (nullable = true)
     |-- card_enabled: integer (nullable = true) [...]
     |-- card_swipe_card_id: long (nullable = true)
     |-- card_swipe_station_id: long (nullable = true)
     |-- replenish_card_card_id: long (nullable = true)
     |-- replenish_card_balance: long (nullable = true) */
```

Figure 4.13: Reading of a stream from socket in Spark and unpacking of the structure to the DataFrame type of Structured Streaming

A declarative SQL test

Spark SQL is a Spark module for structured streaming. It provides interfaces with information about both the data and computation being performed. This abstraction package allow to perform SQL queries over data sets by using same execution engine of the Dataset API when computing a result.

By the performed experiments with this module we wanted to understand if it was possible to express the data transformation logic employed in Flink to develop the same Metrocard application.

Back to the imperative streaming approach

After the failing declarative method we decided to explore the feasibility in remplementig the Metrocard appplication by means of the regular structured streaming API without SQL.

Broadcast variable for static event enrichment To perform the join operation once again we leveraged the equivalent technique in stream processing of event-enrichment. At first, it was necessary to load the stations into each operator so to, then, be able to pick the station add its fare to the card swipe in the dataflow.

However, the broadcast stream feature from Flink is not supported by Spark. The more similar concept the framework provides is the a broadcast variable. This API allows at once to load a data set when the application is starting, and to send it to whichever operator that require to read it. Unfortunately, the loaded data can not be altered once they have been read from the source file. Thus, the data can not be modified once the application is started. We loaded the stations as a broadcast variable accepting the fact that this solution limits the possibilities of the original Metrocard example. Stations in that case could be modified by simply updating the table they were stored into. Next joins would have read the updated data. In the case of Spark, a station update would required to restart the entire application in order to load the update and accordingly perform the event-enrichment of card swipes.

```
/** load stations as dataset */
val stationsArray = spark.read
  .json(jsonStationsPath)
  .as[Station]
  .collect()

/** broadcast the stations dataset */
val broadcastStationsArray = spark.sparkContext.broadcast(stationsArray)
```

Figure 4.14: Reading of stations and broadcasting of the state variable containing them.

After the creation of a broadcast variable its content is made available by the framework to the operators. We use it as show below to select the right station during the map function invocation and perform event enrichment with the station fare information added to the card swipe record.

```scala
def enrichCardSwipe(r: Row): Row = {
  // just return back any row which is not a cardswipe
  if (r.getAs[String]("entity_name") != "cardswipe") return r

  val entityName = r.getAs[String]("entity_name")
  val action = r.getAs[String]("action")
  val cardId = r.getAs[Long]("card_id")
  val stationId = r.getAs[Long]("station_id")

  // retrieve the station associated to the current card swipe
  val filteredStations = broadcastStationsArray
    .value
    .filter(_.station_id == stationId)
  [...]
  val station = filteredStations(0)

  val values = Map(
    "entity_name" -> entityName,
    "action" -> action,
    "station_id" -> stationId,
    "card_id" -> cardId,
    "fare" -> station.fare // enrich!!
  )

  makeNewRowWithSchema(values, schema)
}

// perform event-enrichment by adding the station fare to card swipes
eEntitiesDF = entitiesDF.map(enrichCardSwipe(_))(RowEncoder(entitiesDF.schema))
```

Figure 4.15: Event-enrichment function adding the station fare to card swipes.

The flatMapGroupsWithState After the event enrichment, we focused on saving the cards as state thus with also the card remaining credit. One more time we exploit the mechanism of partitioning to differentiate the cards and operations destined to them towards separate operators. Each operate is in charge of storing one card and to handle the request for card swipes and replenish directed to that card.

```
def updateCardState(card_id: Long,
                    rows: Iterator[Row],
                    state: GroupState[Card]): Iterator[Result] = {
  var out = ListBuffer[Result]()

  for (r <- rows) {
    r.getAs[String]("entity_name") match {
      case "card" =>
        // update the state save for the current card_id
        state.update(
          Card(
            r.getAs[Long]("card_id"),
            r.getAs[Int]("enabled"),
            [...]
            r.getAs[Int]("_notify")))
        out += Result(1, "Card", s"card_id[$card_id] Card stored!")

      case "cardswipe" =>
        if (!state.exists) {
          out += Result(0, "CardSwipe", s"card_id[$card_id] Card swipe not matching any existing card.")
        } else {
          val cardState: Card = state.get
          cardState.balance -= r.getAs[Long]("fare")
          state.update(cardState)
          out += Result(1, "CardSwipe", s"card_id[$card_id] Done! Updated amount: " + cardState.balance)
        }

      case "replenishcard" =>
        if (!state.exists) {
          out += Result(0, "ReplenishCard", s"card_id[$card_id] Replenish card not matching any existing card.")
        } else {
          val cardState: Card = state.get
          cardState.balance += r.getAs[Long]("replenish_balance")
          state.update(cardState)
          out += Result(1, "ReplenishCard", s"card_id[$card_id] Done! New amount: " + cardState.balance)
        }

    }
  }
  // return the iterator of results
  out.iterator
}
```

Figure 4.16: Materialized queries execution against the stored card state.

4.5 Delta Lake experiments

Delta Lake is an application layer levering the the Apache Spark framework for execution. The documentation states that it brings ACID guarantees for the management of *data lakes*[2]. Attracted by the promise of stronger guarantees in a distributed processing platform like Spark we decided to investigate also this recently released processing system.

The database-like approach for big data tables Delta Lake tables are a custom ad hoc format to treat large data sets of items as if they were SQL tables. The delta-tables have the objective to enforce a structure over potentially unstructured and heterogeneous data sets. Also, the format support by design what it is called the "time travel (data versioning)" feature

[2] "A data lake is usually a single store of all enterprise data including raw copies of source system data and transformed data used for tasks such as reporting, visualization, advanced analytics and machine learning." Source: https://en.wikipedia.org/wiki/Data_lake

that allows accessing a past table status given the time at which it is desired to read the data. Delta-tables can be loaded into the application either from files or from the Spark structured streaming APIs. Then SQL queries can be performed by means of direct SQL statements or via programmatic API resembling SQL instructions. These set of features sound very promising in reimplementing the Metrocard application.

Registered tables and queries The reimplementation of the Metrocard application is done with the same approach used by the original VoltDB implementation. Hence, we went back to use queries and synchronous interaction in interacting with the base of data.

We loaded the stations data, previously saved as JSON format, at the beginning of the benchmark and saved it in the delta-table format for later use.

```
spark
    .read
    .option("inferSchema", "true")
    .json(stationsJsonPath)
    .write
    .format("delta")
    .partitionBy("station_id")
    .option("overwriteSchema", "true")
    .mode("overwrite")
    .save(deltaEntities("stations"))
logInfo("Stations loaded.")
```

Figure 4.17: Stations' reading and saving as Delta Table.

Cards generation at the beginning of the benchmark has been done programmatically as did in the Metrocard application. Due to the dramatically slow performance in storing data, that later will be discussed, at each card-generation iteration we chose to first generate all the cards and only after to save them as delta-table.

```
val cards = ListBuffer[Card]()
for (i <- 0 to cardsToBeGenerated) {
  cards += generateCard
    if (i % 5000 == 0) {
        logInfo("  " + i)
    }
}
logInfo("Cards generated.")

logInfo("Appending " + cardsToBeGenerated + " cards to the delta table...")
cards
  .toDF // schema of DF is automatically inferred from case class
  .write
  .format("delta")
  .mode("append")
  .save(deltaEntities("cards"))
logInfo("New cards generated now appended.")
```

Figure 4.18: Cards generation and saving as Delta Table.

Below we report the code execute to charge the card with the station fare by means of the Delta APIs.

```
// UPDATE QUERY
DeltaTable
    .forPath(spark, deltaEntities("cards"))
    // predicate and update expressions using SQL formatted string
    .updateExpr(s"card_id = $cardId", Map("balance" -> s"balance - $fare"))
```

Figure 4.19: Update of the card balance decreased of the station fare via Delta APIs.

Results - the failure in fast data changes Thanks to the SQL-like syntax the Delta API did effectively allowed us to reproduce with a more narrow implementation and the same exact semantic the entire Metrocard application. However, the system did not performed as expected in execution speed. The adoption of the Delta synchronous interaction layer, which leverages an asynchronous platform such as Spark for queries' execution, has been demonstrated to be totally inappropriate for our scenario.

The average throughput obtained by three distinct executions of the Delta Lake-based benchmark is: 0.435 queries per second (Benchmark duration: 90 seconds).

We hypothesize that one of the possible reasons causing this very low throughput is that the platform has been designed only to process big data. Even though the system has the support of SQL expressiveness and queries it actually employs Spark to perform the work. This makes this tool very suitable and most likely very effective when performing analytical queries over large data sets. In our scenario, we operated on a single machine and with a relatively small amount of data.

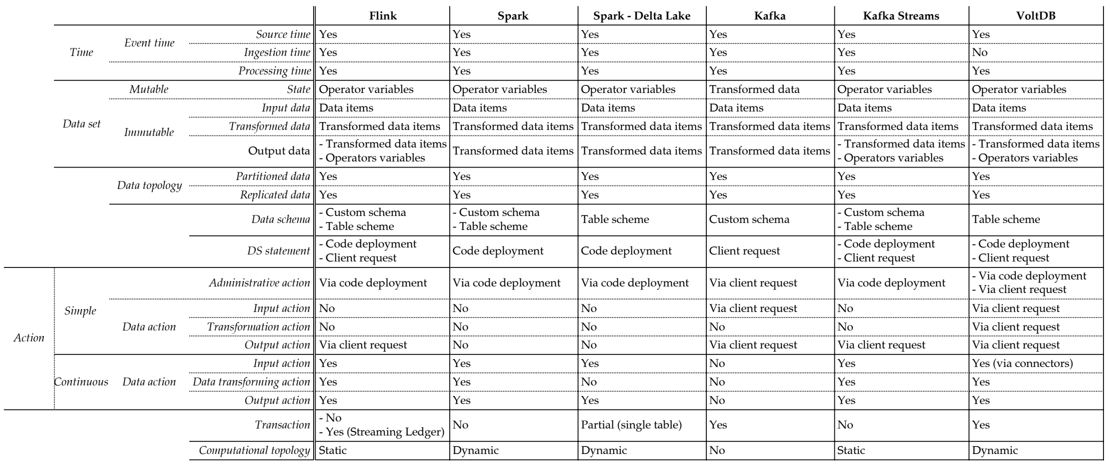
Chapter 5

Taxonomy

In this chapter, the concepts ensued by the presented models are corresponded to the analyzed systems. The objective is to validate and justify models' structures providing a clear layout to highlight platforms' qualities, limitations, and implementations.

5.1 Conceptual model

The concepts formalized and described by the conceptual model in section 3.1 are employed below to describe which of the concepts' specializations better apply to the analyzed tools.



				Flink	Spark	Spark - Delta Lake	Kafka	Kafka Streams	VoltDB
	Time	*Event time*	*Source time*	Yes	Yes	Yes	Yes	Yes	Yes
			Ingestion time	Yes	Yes	Yes	Yes	Yes	No
			Processing time	Yes	Yes	Yes	Yes	Yes	Yes
	Data set	*Mutable*	*State*	Operator variables	Operator variables	Operator variables	Transformed data	Operator variables	Operator variables
		Immutable	*Input data*	Data items	Data items	Data items	Data items	Data items	Data items
			Transformed data	Transformed data items	Transformed data items	Transformed data items	Transformed data items	Transformed data items	Transformed data items
			Output data	- Transformed data items - Operators variables	Transformed data items	Transformed data items	Transformed data items	- Transformed data items - Operators variables	- Transformed data items - Operators variables
		Data topology	*Partitioned data*	Yes	Yes	Yes	Yes	Yes	Yes
			Replicated data	Yes	Yes	Yes	Yes	Yes	Yes
			Data schema	- Custom schema - Table scheme	- Custom schema - Table scheme	Table scheme	Custom schema	- Custom schema - Table scheme	Table scheme
			DS statement	- Code deployment - Client request	Code deployment	Code deployment	Client request	- Code deployment - Client request	- Code deployment - Client request
Action	*Simple*		*Administrative action*	Via code deployment	Via code deployment	Via code deployment	Via client request	Via code deployment	- Via code deployment - Via client request
		Data action	*Input action*	No	No	No	Via client request	No	Via client request
			Transformation action	No	No	No	No	No	Via client request
			Output action	Via client request	No	No	Via client request	Via client request	Via client request
	Continuous	*Data action*	*Input action*	Yes	Yes	Yes	No	Yes	Yes (via connectors)
			Data transforming action	Yes	Yes	No	No	Yes	Yes
			Output action	Yes	Yes	Yes	No	Yes	Yes
			Transaction	- No - Yes (Streaming Ledger)	No	Partial (single table)	Yes	No	Yes
			Computational topology	Static	Dynamic	Dynamic	No	Static	Dynamic

Figure 5.1: Data-intensive tools' characterization by means of the conceptual model

All the analyzed data-intensive tools adopt the notion of time in all their specializations. VoltDB makes an exception for the ingestion time that is not supported out of the box and connectors to external systems are not clear whether if they provide or not this option.

All the systems except for Kafka save the state into operators variables. The case of Kafka is a bit fuzzy since it considers also the data into its queues as the state. This system considers as "global state" the information saved in its queues. This peculiarity combined with the support to transactions allows Kafka to provide strong guarantees such as exactly-once delivery to external systems leveraging it as a reliable communication channel.

Immutable data provided as input and being the results of transformations are concepts established among all the data-intensive tools. However, output data are not treated always in the same way. Flink and Kafka streams allow to query operator states meanwhile this is not supported by Spark, Delta Lake and VoltDB. The output data of VoltDB reports "operator variables" because we wanted to remark that it is an in-memory distributed database and thus, in the execution of a read query, the system actually provides back to the client the saved information by an operator in the cluster.

Each data-intensive tool has shown support for replication and partitioning, although in different forms and for various purposes.

With the latest releases of Flink, Spark and Kafka the support to tabular data structures is getting more and more expressive and powerful. These systems indeed are improving their processing capabilities going beyond the custom schema for data that they already support. They allow to use SQL-like queries to declare data filtering and transformations. Delta Lake does not support any custom data format since it is purely based on the concept of table implementing and leveraging for its functioning the Delta Tables. VoltDB, being a database by definition, only support table schemes.

DS statements, by the conceptual model we developed in section 3.1, can be of two form: code deployment and client requests. The first involves the sending of a software package to the system and its the main method employed by Flink, Spark, Delta Lake and Kafka Streams. Also VoltDB supports this interaction since we associate it to the capability of the system in hosting the stored procedure. Flink and Kafka streams also support client requests to receive the queries to their operators, while for VoltDB this interaction is the principal since it synchronously execute SQL queries.

Administrative actions are performed by Spark, its extension Delta Lake, and Flink by the deployed code. Kafka and VoltDB mainly perform them by means of client requests.

No simple input data action is supported by the stream and batch processor, nor by Delta and Kafka Streams. These systems indeed process inputs only ingested by continuous data actions. Kafka allows instead, as VoltDB does, to send input within client requests. No simple transformation action can be performed by any of the systems except of VoltDB. The databases allows to perform data updates by means of client requests: *UPDATE* queries. Simple data output actions are supported by Flink and Kafka Streams in the sense that they are employed to query the state of operators. Their expressiveness is quite limited. VoltDB instead has full support to simple data output actions since they are the computation performed for the *SELECT* queries.

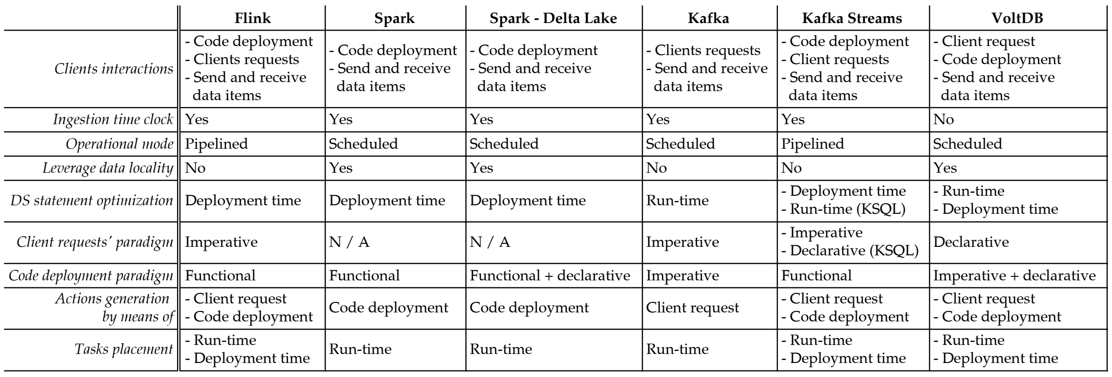
Continuous data input, transforming and output actions are fully supported by Flink, Spark, Kafka Streams and VoltDB. The latter implements input by means of additional connectors that can be plugged to the system. Delta Lake, due to its design is not really supporting continuous data transforming actions but rather it performs long-running operations that then at some time will terminate. Kafka does not support at all continuous data actions.

The transaction concept shows how the systems have focused their attention to different features since their design. Flink does not support the transactions at all. A recent extension called Flink Streaming Ledger [dA18] introduced transactional guarantees through data controlled procedures similar to the VoltDB stored procedures. Also, Spark does not support transactions and the same is for Kafka Streams. The pure version of Kafka instead allow to express the logical windows of operation by means of its APIs. VoltDB support the strongest transactional guarantees of all the systems.

Flink and Kafka Streams architecture acts by statically allocating the transformation functions over the nodes and made the data flows through them. However, not all the analyzed systems act by means of this principle. We categorized Spark, Delta Lake and VoltDB together in the class of systems that employ a dynamic computational topology. The notion refers to the fact that these platforms execute the requests by sending the computation towards the nodes. They also do so dynamically in the sense that the process is carried out according to the type of requests and by the time they have to be executed.

5.2 Operational model

In this section, we report the differences in the systems' functioning. The objective is also to highlight the cases in which they manifest similar or same execution approaches.



	Flink	**Spark**	**Spark - Delta Lake**	**Kafka**	**Kafka Streams**	**VoltDB**
Clients interactions	- Code deployment - Clients requests - Send and receive data items	- Code deployment - Send and receive data items	- Code deployment - Send and receive data items	- Clients requests - Send and receive data items	- Code deployment - Client requests - Send and receive data items	- Client request - Code deployment - Send and receive data items
Ingestion time clock	Yes	Yes	Yes	Yes	Yes	No
Operational mode	Pipelined	Scheduled	Scheduled	Scheduled	Pipelined	Scheduled
Leverage data locality	No	Yes	Yes	No	No	Yes
DS statement optimization	Deployment time	Deployment time	Deployment time	Run-time	- Deployment time - Run-time (KSQL)	- Run-time - Deployment time
Client requests' paradigm	Imperative	N / A	N / A	Imperative	- Imperative - Declarative (KSQL)	Declarative
Code deployment paradigm	Functional	Functional	Functional + declarative	Imperative	Functional	Imperative + declarative
Actions generation by means of	- Client request - Code deployment	Code deployment	Code deployment	Client request	- Client request - Code deployment	- Client request - Code deployment
Tasks placement	- Run-time - Deployment time	Run-time	Run-time	Run-time	- Run-time - Deployment time	- Run-time - Deployment time

Figure 5.2: Data-intensive tools' characterization by means of the operational model

In the operational model we defined client interactions as all the possible ways external applications can interact with data-intensive tools. All those that require a user-developed application to receive processing instruction support the code deployment interaction: Flink, Spark, Spark - Delta Lake, Kafka Streams and VoltDB. Flink and Kafka streams provide also client interactions because they allow, by means of special APIs, to build an external client able to query the state of the operators. Client requests in case of Kafka and VoltDB are instead the normal way these systems allow the user interactions. Kafka allows users to write and read items from queues programmatically while VoltDB processes the SQL query as all OLTP[1] systems do. Data items are sent and received by all of the platforms as well as almost all of them allow the attach to the input data the ingestion time. The case of VoltDB is an exception to this since, even though it supports connectors to receive data items, the system does not implement out of the box the concept of ingestion time. Custom made solutions could do so.

With respect to the adopted operational paradigm data-intensive tools can be classified in two families. We classify as pipeline processing systems Flink and Kafka Streams because of the dataflow graph they deploy to carry out transformations. The computation remains stationary and continuously running on the nodes while data flows through them. All the other analyzed systems employ a scheduled approach in the sense that they deliver the required execution to the interested nodes when they need to perform user requests.

[1]OLTP stands for "Online transaction processing" and it is a class of systems that have the capability to carry out transactional requests as soon as they are sent to the system

When data locality is leveraged in the functioning of a system, the information is kept as much as possible located at the place it was originally stored. The objective is to transfer the least possible amount of data through the network. Data-intensive tools adopting a pipelined operational approach do not exploit data locality while those that employ scheduling usually do.

DS statements are the user instructions sent to the system. They need to be interpreted so to produce the actions in charge to perform the semantic meaning of what the request asked the system. Another characterization property of data-intensive tools is the moment at which actions are optimized for their distributed execution. Almost all the systems, except Kafka, do optimize DS statements at deployment time, so when they receive the application code from the user. Also, for the systems that support client request with potentially complex semantic (no simple read queries – i.e. queryable operator state) clients' DS statements get optimized at run-time to build an effective execution plan.

The way client requests express their objective influence the possible interactions and reasoning employed in their formulation. Declared by means of formal languages they do conform to the programming paradigm the language involves. The imperative paradigm, derived from Java programming code, is used in Flink, Kafka and Kafka Streams for client requests. VoltDB with SQL and the KSQL on top of Kafka Streams are instead declarative languages.

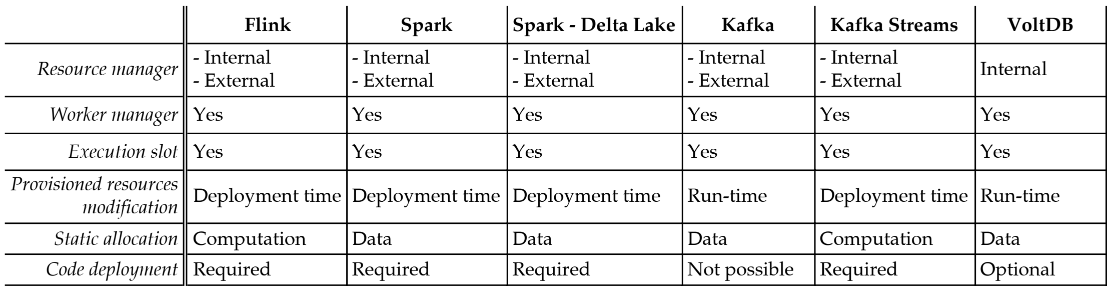
The software package deployed to the data-intensive tools can be as well characterized in terms of paradigm adopted. Flink, Spark, Delta Lake, and Kafka Streams employ a functional approach to let the user define what the final result is and let the platform taking care of build the processing required to achieve it. Delta Lake supports also the SQL language and this characterizes the system with a functional and declarative paradigm. Kafka exposes imperative APIs to let clients read or writes data to its queues. With VoltDB deployed code we mean the stored-procedures. This application logic that gets installed within the database is characterize by the mix of both imperative Java code and declarative SQL queries.

Actions are generated by the interpretation of DS statements. Based on the interaction supported by the system they can be generated by means of deployed code or a client request. All the platforms made exception for Kafka generate actions after code deployment (VoltDB does so for stored procedures). Client requests, instead, trigger actions generation in the case of Flink and Kafka Streams after a queryable state request while in the case of Kafka they generate actions since they are the only possible interaction.

Tasks placement is a critical factor for optimization, and execution plans. Flink, as well as Kafka Streams, does so at deployment time since it interprets the processing required, formulates an execution plan, and based on the available resources places the operators over the nodes. Spark instead place tasks at run-time since, by the scheduled nature of the system, it sends execution towards the nodes when it is required. Delta Lake does the same of Spark since it is based on it. Kafka, as VoltDB does, employs the tasks to instruct workers for the data to be saved or retrieved and hence "place" or, better, deliver them at run-time. We stated "run-time" in the task placement also of Flink and Kafka Streams to include the run-time execution of the client requests made to retrieve a state.

5.3 Architectural model

The data-intensive tools' architecture is now described leveraging the architectural layout outlined by the relative model.



	Flink	Spark	Spark - Delta Lake	Kafka	Kafka Streams	VoltDB
Resource manager	- Internal - External	- Internal - External	- Internal - External	- Internal - External	- Internal - External	Internal
Worker manager	Yes	Yes	Yes	Yes	Yes	Yes
Execution slot	Yes	Yes	Yes	Yes	Yes	Yes
Provisioned resources modification	Deployment time	Deployment time	Deployment time	Run-time	Deployment time	Run-time
Static allocation	Computation	Data	Data	Data	Computation	Data
Code deployment	Required	Required	Required	Not possible	Required	Optional

Figure 5.3: Data-intensive tools' characterization by means of the architectural model

Almost all the platforms support external resource managers, made an exception for VoltDB. External resource managers are software not part of the data-intensive tool that are employed to administrate the cluster nodes. The reason for this lack is probably due to its ad hoc resource management mechanisms employed in the transactions' scheduling that needs to be very carefully handled to exploit coordination-free execution.

Every system has its own worker manager installed on the nodes to handle tasks and provide back execution statistics, as well as all of them employ the execution slot. Each of the latter is often corresponded to one partition so to parallelize at the most execution and enabling the single-site, one-shot transactions correctness in the case of VoltDB.

It is only for the cases of Kafka and VoltDB that the provisioned resources can be modified while the system is running. All the others do not allow it, requiring the temporary interruption of execution and then its resume.

With respect to static allocation of either the computation or data, we have classified all the tools that adopt a "pipelined processing" such as Flink and Kafka Streams with static computation allocation.

The stream and batch processors, and Kafka Streams require the deployment of an application to receive instructions about processing they have to perform. They in fact leverage a user-defined application code to understand how the data transformations should have to be carried out. Kafka, on the other hand, does not allow to deploy application code since the interaction with this system is provided only through statements sent via client requests. VoltDB is instead the hybrid system with respect to this feature sine it supports code deployment as well as client requests. It allows deploying portions of application code called "Stored procedures" that enable the user to define as many SQL queries as he/she prefers and interpose them with imperative logic expressed through Java code.

5.4 Guarantees model

The following employ the guarantees model definitions to characterize the interactions with the analyzed systems.

The guarantees for below described continuous action data actions are intended considering the multiple tasks composing a single action. In fact, when concurrent interactions guarantees are examined we consider continuous data tasks and the isolation enforced between their execution. For the definition of continuous data tasks and actions please refer to the conceptual model, section 3.1.

Apache Flink

Interactions' guarantees

	Client request	**Continuous data actions**
Atomicity	No	No
Consistency constraints	Not supported	Not supported
Durability	N / A	Recoverable — valid state
Replication Consistency	N / A	N / A
End-to-end order	Sequential	FIFO by key
End-to-end delivery	N / A	- At least once - Exactly once

Table 5.1: Interactions' guarantees in Apache Flink

The poor set of guarantees that Flink provides clearly emerges from table 5.1. The vanilla version of the stream processor does not support the concept of transaction neither it really supports client requests made an exception to those read queries fetching operators state. The stream processors provide durability at level of recoverable – same state thanks to checkpointing and event reply (from repayable source – e.g. Kafka). It does not employ replication as durability technique hence no guarantee is applicable to this. The order of data the system process is guaranteed to be FIFO only within the same partition, thus the FIFO by key guarantee. End-to-end delivery is guaranteed to be at least once thanks to the durability mechanism employed. However, when external systems are coupled in the use of Flink, like Kafka but not only, the stream processor also guarantee exactly once delivery. Software connectors exploit transactions in reading new data and writing the processed ones. For this reason, by the transactional guarantees of the external system also Flink is able to ensure that data items are delivered end-to-end exactly once.

Isolation guarantees in concurrent interactions

	Client request	**Continuous data tasks**
Client request	Serializable (reads)	
Continuous data tasks	Serializable (single partition)	None

Table 5.2: Isolation levels for concurrent interactions in Apache Flink

Client request are only read queries directed to an operator. They read only from a single partition at a time by how Flink APIs are designed. This guarantees an isolation level between concurrent request being serializable. Also for client requests concurrent to continuous data tasks, the isolation level is considered the same. No guarantee is instead provided when concurrent continuous data tasks conflicts with each others on the same state.

Apache Spark

Interactions' guarantees

	Client request	**Continuous data actions**
Atomicity	Not supported	No
Consistency constraints		Not supported
Durability		- Recoverable — same state - Recoverable — valid state
Replication consistency		Sequential
End-to-end order		No (micro-batch)
End-to-end delivery		- At least once - Exactly once

Table 5.3: Interactions' guarantees for Apache Spark

Spark does not support client requests hence no guarantee can be argued about them. Continuous data actions are neither atomic nor allow to enforce data consistency constraints. Spark structured streaming API allows schema definition but not checks such as referential integrity of foreign keys or primary keys can be set up.

In terms of durability the batch processing provides two levels. Spark provides recoverable – same state level when deterministic and commutative transformation are performed to transform sets of data. In this case, when recovering from a failure the system will start from the last checkpoint performing again the computation but eventually in different order. If commutativity and determinism holds then it is guaranteed the state restored is the same of before the failure. Instead, when transformations do not have these properties only the durability level of recoverable – valid state is provided.

Replication is provided by passively replicating RDDs, Resilient Distributed Data sets [ZCD+12]. Due to this we state that consistency among RDDs is sequential since they are indeed exact same copies.

No end-to-end order is guaranteed by Spark. Due to the micro-batch approach, the processing is carried out on all the data items, within a batch, as faster as possible and in parallel. No order in which they gets transformed is guaranteed since it depends on the scheduling of machine threads operating the transformation.

Isolation guarantees in concurrent interactions

	Client request	Continuous data tasks
Client request	N / A	
Continuous data tasks	N / A	None

Table 5.4: Isolation levels for concurrent interactions in Apache Spark

No client request is supported in Spark, thus there is no guarantee that can be considered about it.

As in the case of Flink, no isolation guarantee is provided between continuous data tasks.

Apache Spark - Delta Lake

Interactions' guarantees

	Client request	Continuous data actions
Atomicity	Partial	N / A
Consistency constraints	Not supported	
Durability	Partial	
Replication consistency	N / A	
End-to-end order	N / A	
End-to-end delivery	N / A	

Table 5.5: Interactions' guarantees for Apache Spark Delta Lake

Delta Lake is an abstraction layer to handle heterogeneous data formats at scale. "ACID guarantees are predicated on the atomicity and durability guarantees of the storage system" [2]. Thus this data-intensive tool deeply leverages the underlying storage systems guarantees.

The table 5.5 shows the very poor guarantees that this system actually provides, despite the strong claims of its documentation page. No consistency constraints can be enforced on the data schema, if not the types of the columns and their ordering with respect to each other. The durability and atomicity guarantee is really provided by Delta Lake. This tool indeed leverages the underlying storage system capabilities to support these guarantees, that otherwise would not be provided at all.

Isolation guarantees in concurrent interactions

	Client request	Continuous data tasks
Client request	Partial - Snapshot isolation	
Continuous data tasks	N / A	N / A

Table 5.6: Isolation levels for concurrent interactions in Apache Spark Delta Lake

[2]Definition from `https://docs.delta.io/latest/delta-storage.html`

Client requests isolation is by means of separate files that store the modified data. The way Delta Lake isolate requests is by means of JSON files that record the modification made by writes. This is the reason why we stated ”partial” support to isolation. If the underlying filesystem provides atomic file visualization and access (to multiple instances) then Delta Lake ensure snapshot isolation level for concurrent client requests.

The system employs and optimistic concurrency control approach. If two writes concurrently change the same table, at first, two new files to store the changes are created. Then, on the attempt of committing, the systems checks whether the modified data creates conflicts. If there are not any, the write operations are allowed to commit and succeed. However, if conflicts are found then one of the two operations is aborted with the a “concurrent modification exception” [3].

Apache Kafka

Interactions’ guarantees

	Client request	**Continuous data actions**
Atomicity	Yes	N / A
Consistency constraints	Not supported	
Durability	Replicated — same state	
Replication consistency	Sequential	
End-to-end order	FIFO by key	
End-to-end delivery	- At least once - Exactly once	

Table 5.7: Interactions’ guarantees for Apache Kafka

Apache Kafka guarantees the atomicity of client requests since the APIs provide synchronous reliable operations to the clients. No consistency constraints over the data can be specified. Tolerance to hardware failure for data durability is guaranteed with replication; it derives that the guarantee level for durability is replicated – same state. Replication happens synchronously to the requests and replicas consistency is guaranteed to be sequential. The end-to-end order guaranteed is FIFO by key. Topics can be partitioned by key and Kafka enforce a FIFO order for the data sent with the same key. The end-to-end delivery guaranteed by default is at least once. Exactly once can be achieved making the assumption that the performed operations are pulling from a Kafka queue and writing back on another Kafka queue.

Isolation guarantees in concurrent interactions

	Client request	**Continuous data tasks**
Client request	- Read committed - Read uncommitted	
Continuous data tasks	N / A	N / A

Table 5.8: Isolation levels for concurrent interactions in Apache Kafka

[3] `https://docs.databricks.com/delta/concurrency-control.html`

Kafka allows to setup two different isolation levels: read committed and read uncommitted. Transactions, natively supported by Kafka, will read in the first case only committed data and in the second case any data written.

Kafka Streams

Interactions' guarantees

	Client request	**Continuous data actions**
Atomicity	No	No
Consistency constraints	Not supported	Not supported
Durability	N / A	Recoverable — same state
Replication consistency	N / A	Sequential
End-to-end order	Sequential	FIFO by key
End-to-end delivery	N / A	- At least once - Exactly once

Table 5.9: Interactions' guarantees for Apache Kafka Streams

Kafka Streams, by means of dedicated APIs, allows to query an operator state from an external client similarly to how Flink does. However, this feature does not come with strong guarantees as can be seen in the table above.

Continuous data actions are guaranteed to be fault tolerant with a data durability level of recoverable – same state by means of checkpointing and reply of events. Replication consistency is the same as it is in Kafka since the synchronous interaction with queues and the way writes are propagate before their confirmation. As in Flink, Kafka Streams preservers end-to-end order within the same partition with a FIFO by key level.

The end-to-end delivery is guaranteed to be at level of least once. In the case reliable messaging system are used as sources and sinks, e.g. Kafka itself, the provided guarantee gets stronger up to exactly once.

Isolation guarantees in concurrent interactions

	Client request	**Continuous data tasks**
Client request	Serializable (reads)	
Continuous data tasks	Serializable (reads)	None

Table 5.10: Isolation levels for concurrent interactions in Apache Kafka Streams

The concurrent reads to operators state are not actually isolated with respect to each other but since no writes are allowed we can state they are isolated at the same level of serializable requests.

VoltDB

Interactions' guarantees

	Client request	**Continuous data actions**
Atomicity	Yes	Yes
Consistency constraints	Supported	Supported
Durability	Replicated — same state	Replicated — same state
Replication consistency	Sequential	Sequential
End-to-end order	N / A	N / A
End-to-end delivery	No	Exactly once

Table 5.11: Interactions' guarantees for VoltDB

VoltDB supports transactions and strong guarantees in their execution. The NewSQL database does enforce all the ACID guarantees hence providing atomicity, consistency constraints enforcement, durability and a strong isolation level for concurrent transactions. It employs replication to undergo failures ensuring no committed data is loss. The durability level provided is replicated – same state due to the active strategy employed. Replication consistency level is sequential since scheduled transactions are guaranteed to be performed in the same order at all the sites. The end-to-end order concept has no meaning for a database. Data can be stored by input client requests and other ones can fetch them specifying any order they want. We state for the end-to-end order that there is no guarantee. This claim derives from the approach in which Stored procedures execute SQL queries. Through the Java code multiple queries can be specified, within a stored procedure, and asynchronously executed with a single method invocation. No completion or execution order is guaranteed by VoltDB. Their result will be available in any order to the requesting run-time as soon as it is available. For continuous data actions we state data VoltDB provides provides exactly once delivery because of the synchronous trigger invocation mechanism employed in the support provided to stream processing.

Isolation guarantees in concurrent interactions

	Client request	**Continuous data tasks**
Client request	Serializable	
Continuous data tasks	Serializable	Serializable

Table 5.12: Isolation levels for concurrent interactions in VoltDB

VoltDB provides the higher isolation level for any of the possible concurrent interactions. Thanks to the employed coordination-free approach it is able to provide this strong guarantee with almost not affecting the computation performance. As already mentioned, when the transaction structure, hence made by operations action cross-partition, does not permit the adoption of only scheduling as a conflict avoidance mechanism, the database falls back to regular coordination-based protocols such as 2PL.

5.5 Guarantees' implementations

In this section, interactions with data-intensive tools are described in terms of protocols and approaches adopted in the implementation of the provided guarantees.

Apache Flink

	Client request	**Continuous data action**
Atomicity	N / A	N / A
Consistency constraints	N / A	N / A
Isolation	Scheduling - single sited	N / A
Durability	N / A	Checkpoint
Replication consistency	N / A	N / A
End-to-end order	N / A	Watermarks
End-to-end delivery	N / A	Snapshot

Table 5.13: Guarantees' implementations employed by Apache Flink

The isolation of concurrent client requests is inherited automatically by the scheduling the system adopts in its execution. Client requests in Flink are only allowed to fetch variables of operators, thus all of them are read operations. We specified "single sited" due to the way Flink implements these requests. The APIs structure forces the developer to query only one state variable at a time and fetching it from a single operator representing thus a single partition.

Continuous data action are made reliable with respect to failures through checkpointing. With this method, the systems takes care periodically to do backup copies of the state variables and to persist the reached point in the processing. In case of failure, the computation is resumed from the last available checkpoint and by replying to the last missing operations the system recovers the current processing state.

In a distributed scenarios multiple are the reasons that can induce delays in message deliveries. Flink uses the watermarking technique to filter incoming messages by excluding those whose arrival was expected a long time before. The end-to-end delivery is guaranteed by the creating periodic snapshots of the state that tracks which are the messages being sent. In case of failures, saved snapshots are evaluated to restore the processing and in case of lost messages, they gets replayed.

Apache Spark

	Client request	**Continuous data action**
Atomicity	N / A	N / A
Consistency constraints		N / A
Isolation		N / A
Durability		Checkpoint
Replication consistency		Passive
End-to-end order		N / A
End-to-end delivery		Snapshot

Table 5.14: Guarantees' implementations employed by Apache Spark

Spark employs checkpointing technique to provide data durability. Asynchronous copies of RDDs are created when transformations succeed thus employing a passive replication technique. We marked the end-to-end delivery as provided through snapshot since Spark acts by recomputing the an RDDs transformation in case of failure and does so only once. If parallel re-execution of the same task happen, one of them is aborted by the system. Due to this behavior we can consider the last available valid RDDs as part of the last snapshot the system can rely on.

Spark - Delta Lake

	Client request	**Continuous data action**
Atomicity	N / A	N / A
Consistency constraints	N / A	
Isolation	MVCC	
Durability	N / A	
Replication consistency	N / A	
End-to-end order	N / A	
End-to-end delivery	N / A	

Table 5.15: Guarantees' implementations employed by Spark Delta Lake

We marked many of the guarantees' implementations as not applicable since either Delta Lake does not provide them, e.g. consistency constraints, or because the partial support to them is obtained by other tools and not directly from Delta Lake, such as it is atomicity and durability. Isolation is enforced through atomicity of the data storage Delta Lake relies on, but in this case the system also takes care to enforce the guarantee by performing itself the checks for conflicts. It implements a Multiversion Concurrency Control mechanism in which for concurrent writes multiple version of the data are saved and, later, checked for conflicts.

Apache Kafka

	Client request	**Continuous data action**
Atomicity	2PC	N / A
Consistency constraints	N / A	
Isolation	MVCC	
Durability	Replication	
Replication consistency	Passive (synchronous)	
End-to-end order	N / A	
End-to-end delivery	N / A	

Table 5.16: Guarantees' implementations employed by Apache Kafka

Apache Kafka natively support the concept of transaction. We categorize client requests as if they were transactions. To coordinate distributed commits and provide atomicity Kafka employs 2PC protocol. This queue-based system however does not provide any consistency constraint over the data since also no data schema can be enforced. We assigned MVCC to the technique employed for isolation since Kafka keeps track of committed and uncommitted data with a state variable but it saves both as data versions. Durability is ensured by replication indeed each partition of each topic is replicated for fault tolerance. Passive and synchronous replication is the method employed to ensure replicas consistency.

Kafka Streams

	Client request	**Continuous data action**
Atomicity	N / A	N / A
Consistency constraints	N / A	N / A
Isolation	Scheduling - single sited	N / A
Durability	N / A	Replication
Replication consistency	N / A	Passive (synchronous)
End-to-end order	N / A	Watermarks
End-to-end delivery	N / A	Transaction

Table 5.17: Guarantees' implementations employed by Apache Kafka Streams

Kafka Streams has the objective to enable continuous data actions but it does so with weak guarantees. The supported client requests are only the read queries to fetch operators' state. As for the case of Flink, the guarantees in their execution are characterized in the same way being only isolated as a consequence of how APIs are structured.

Then, for continuous data actions no atomicity and consistency constraints are guaranteed. Durability is implemented by replication as it is in the vanilla version of Kafka which in turn keeps replicas consistent with each other by passive replication. Kafka Streams indeed deeply leverages the Kafka APIs to implement continuous processing and so provide streaming functionalities.

VoltDB

	Client request	**Continuous data action**
Atomicity	- Scheduling - 2PC	- Scheduling - 2PC
Consistency constraints	- Scheduling - 2PC	- Scheduling - 2PC
Isolation	- Scheduling - 2PL - MVCC	- Scheduling - 2PL - MVCC
Durability	Replication	Replication
Replication consistency	- Active — single leader - Passive	- Active — single leader - Passive
End-to-end order	N / A	N / A
End-to-end delivery	N / A	N / A

Table 5.18: Guarantees' implementations employed by VoltDB

The stronger guarantees provided by VoltDB are powered by new implementation techniques such as coordination-free isolation and atomicity. The NewSQL database provides atomicity and isolation by scheduling transactions deterministically on each partition of data. When the transaction structure does not allow the use of a coordination-free approach, such as when operations span across partitions, the system falls back to coordination-based protocols such as 2PC and 2PL. Moreover, MVCC strategy is also employed to ensure higher performance and optimize long-running operations such as the system snapshots. VoltDB employs both active and passive replication but for different purposes. Active replication is employed for inter-cluster replicas ensuring consistency and enabling high availability. Then, the configuration option of this database also allow to setup multi-cluster scenarios in which passive replication is used to propagate changes. [4] Continuous data actions are implemented through the use of triggers. This involves that all the guarantees valid for client requests, i.e. the queries, are the same valid for continuous data actions.

[4]https://www.voltdb.com/product/features-benefits/high-availability/

Chapter 6

Discussion

In this chapter we discuss results emerged by the characterization made with tables in the taxonomy, chapter 5. The objective is to highlight the proofs to formulated hypothesis expressed in background and motivations chapter 2. We will provide evidence of the theorized unifying models' effectiveness, of their solidity and ability to carefully describe the systems. They indeed promote the speculative idea of future convergence to a unified architecture of data-intensive tools.

Conceptual model

All systems support the concept of time and they do it for almost all time specializations. The only exception is made for ingestion time that is not supported by VoltDB. However, event time is the most relevant one since it is widely adopted by the majority of application scenarios as the type of timestamp with more value with respect to the others.

Every system implements the notion of state. Stream processors, however, use it mainly as internal information to support data item transformations. Databases make it be the central object around which they are built. Kafka also proposes itself as a tool to save a state in its persistent communication channels. However, it does so without providing sound data structures and any of the features such as tables and JOIN operations as databases do.

Input of single data items is supported by each data-intensive tool. Still, the systems' implementations sometimes affect the way in which processing of transformed and output data is carried out.

As designed to operate in distributed scenarios, all the systems support both the possible optimizations about data placement. Replication and partitioning techniques are adopted to obtain lower latencies in some cases while, in others, they are the only possibility to handle large data volumes.

The stream processors together with Kafka are getting closer to database abstractions providing functionalities to query and filter the data according to a tabular logic. VoltDB, as a database, is the only one of this data schema lacking the capability to handle custom and nested data. This could be brought back to constraints imposed by the SQL logic and relational nature.

With respect to the generation of administrative actions, those employed to alter the internal system structure as well as data topology, stream processors manifest the limitation to be instructed only by means of application code. Instead, Kafka and VoltDB can also generate administrative actions from received client requests; for example to then set up continuous data actions.

Simple output actions are supported by the majority of systems. Spark is an exception as once the processing has started it does not even allow to check the status of the operators as, instead, it is possible to do in Flink.

Overall, all data-intensive tools support continuous data actions. In the case of VoltDB, it employs connectors to external systems specifically made. While, in the case of Kafka, it implements additional layers to support continuous processing instead of only acting as a reliable channel to also store data.

Transactions are not implemented by stream processors. The semantics of transactional guarantees with respect to continuous data streams is still uncertain. However, the effort these systems are making in trying to fill this gap is evident. By means of application layers that can be installed on top of Flink and Spark, respectively Streaming Ledger and Delta Lake, these systems provide an implementation of the transactions according to the semantics they have chosen.

The emerged computational topologies define how different architectural choices have led opting to opposite approaches in the arrangement of tasks' execution. They either send the processing towards the data, being the later steady to the workers, defining the dynamical approach. The other possibility is instead to allocate tasks running and fixed at execution slots, then push the data flows through them to apply transformations.

Operational model

In the systems' functioning, some shared features have been highlighted by the model we built. Data-intensive tools interact with external clients in multiple ways. There's a sign from the stream processors showing the intention of stream processors to support pull-type invocations. Even the databases, in turn, with the intention of providing stream processing capabilities, have concentrated efforts to provide clients particular interactions to handle these ideally unbounded structures. However, the approach of these two worlds is still subject to severe limitations. For example, interaction with the state of an operator in Spark is not possible, whereas in Flink it only occurs through by means of specific configurations and through the development of a dedicated client that uses the API provided. VoltDB emulates the behavior of data ingestion and continuous input of data while it actually doesn't fully support it as an internal feature. The data items taken into are inserted into a regular database table. If no discard policies are put in place once the data has been processed the stream information are saved as regular data. In a similar way, even transformations' results are not exactly pushed out to external systems as stream processors do. It is necessary the external system interested in querying the result to actually fetch them connecting to VoltDB. However, we have to highlight that this database, with the concept of stored procedures, supports the deployment of user-defined application code. This is a common factor to the stream processors that also use the code deployment concept to receive instructions.

With respect to computation and data-arrangement choices, systems functioning is assigned to the classes: pipelined and scheduled. In the first case, the data-locality factor is not exploited since the idea is exactly one to arrange statically the processing and make the data flow through a "logical pipe". Instead, when the data-locality advantages are exploited. systems move the operational code towards data location and try to send information via the network as less as possible.

All systems create execution plans for optimizing the required operations. According to types of interaction, all of them plan the execution following the receipt come with compiled code or that is inferred from the client request. When the last is the case, actions' execution plans are computed at run-time.

The criteria that characterize the clients' requests induce expression paradigms. Flink and Kafka allow clients to send the request with the classic imperative paradigm. VoltDB queries since are expressed by means of SQL, instead conform to a declarative semantics. Moving the focus to the code deployment case for a DS statement, we discover that in this scenario some cases require different communication paradigms with respect to client request interactions. Stream processors indeed stand out for their strong functional paradigm adoption while, when we approach the database world, in addition to the declarative mode, also the imperative one is used in the case of VoltDB stored procedures.

The tasks, actions' execution units, are generated following the interpretation of the DS statement received. The generation of the action to be performed is followed by the formulation of an execution plan that in turn drives the split of action into tasks. In the code deployment situation, the planning is carried out at deployment time. Instead, for the client requests received during operation, the systems will generate the plan at run-time. The task placement is carried out according to the operating policies of the tools. All of them made an exception for Spark, do the operator placement at deployment time. Then, doing another exception for the stream processor by definition, Flink, all the systems also support placement at run-time. By the taxonomy in fig. 5.2, it emerges that VoltDB due to its support to stored procedures and online transactional queries really operates both approaches.

Architectural model

All the systems employ a single master coordination strategy. They are able indeed to achieve fault-tolerance by master replication.

Stream processors, born to be run on a large number of servers, and therefore with so many resources to orchestrate, have supported external software managers since the beginning. Instead, for VoltDB this support is not provided, not even partially. The reason for this it is that, probably, resource management constitute an integral part of system functioning and administration; it is in fact used to provide high throughput and transactional guarantees.

All the data-intensive tools adopt the concept of worker manager to administer the single server part of the cluster. They also employ the idea of execution slot, with some implementation nuances, to carry out tasks execution.

However, it is important to note hoe rigidity of some paradigms and architecture does not allow to scale available resources at run-time.

From the characterization tools' architecture, it emerges how the systems carry out execution of transformations by means of deployed code or client requests. Stream processors can be instructed only through ad hoc applications that employ the APIs from the framework. Instead, Kafka and VoltDB support this mode as an additional functionality; as the main interaction mechanism they allow client request to both alter the internal system structure and to manipulate data

Guarantees model

In the developed modeling framework we include also a model of the guarantees provided in interactions with the systems. Client requests are poorly supported by Flink and they are not at all provided by Spark. Continuous data actions are instead provided with more guarantees by both the systems. In Flink they neither imply atomicity nor consistency constraints since it is not possible to specify them. However they enforce data durability since both Flink and Spark are able to restore the execution in case of failure. Flink restoring from a checkpoint will rebuild a valid state, not necessarily the one present before the failure. Spark instead, can either rebuild the same state in case of deterministic transformations (RDDs computation) or rebuild a valid state in the others. Replication consistency is not applicable in case of Flink since no replica is created but only backups. We stated that Spark provides sequential replication consistency since the copies of the computed RDDs do show transformations in the same, sequential order. In terms of end-to-end order, Flink give the guarantees of FIFO ordering when considering the data in partitions (by key). Ordering is not supported by Spark due to the batch processing. When sets of items in the batch gets processed there is no guarantee in the order they will appear after transformations. In terms of end-to-end delivery, Flink guarantees at least once delivery of items. If the deployment is coupled with reply-able sources and sinks that support retraction, the possible guarantee is stronger and become exactly once delivery. The same set of delivery guarantees holds for Spark continuous data actions. When concurrent operations are executed, it rises the isolation problem to avoid possible conflicts with each other. Since Flink allows only fetching the state of an operator, it is by definition a single partition request and hence read operations from client requests are guaranteed to be serializable with respect to each other and to continuous data tasks. Spark does not provide any isolation guarantee in terms of concurrent data tasks.

Delta Lake, the Spark extension claiming to bring ACID guarantees to Spark, supports only client requests. However no specific guarantee is provided by this layer. No atomicity in enforced for single requests if not the underlying distributed filesystem does not provide atomic visibility of files. No consistency constraint can be specified with its APIs. Durability is not provided by this application layer by itself, instead it relies on durability of external filesystem. Concurrent client request are guaranteed to be isolated at snapshot isolation level.

Kafka provides atomic push and pull operation to send messages through its queues. No consistency constraints can be enforced on them since also no fixed data schema can be specified. Durability of data from client requests is guaranteed at level of replication – same state and the replicas consistency levele is sequential. The end-to-end order is guaranteed to be FIFO by key and the delivery of messages happen with either at least once semantic if the downstream system is not Kafka, or with exactly once if the transaction operated in the client request move the message to still a Kafka queue. Concurrent client interactions are either guaranteed to be isolated at read committed level or read uncommitted level. No support for continuous actions is present in the vanilla Kafka version.

Kafka Streams is the additional application layer that brings stream processing to Kafka. It supports only fetching of the operator state like Flink does and hence ensuring the same guarantees Flink provides in terms of client requests. Continuous data actions do not provide atomicity and neither checking of consistency constraints but enforce durability at recoverable – same state level. Replication consistency is provided with the sequential level and the end-to-end order is like for stream processors guaranteed to be FIFO by key. Guarantees of message delivery are the same of Kafka since the underlying employed queuing system for streams is the same.

VoltDB is the tool that provides the stronger guarantees. It ensure atomicity as well as consistency constraints checks both for client requests and continuous data actions. Durability of data is guaranteed with replication – same state level. Consistency of replicas, thanks to active replication, is at sequential level. The end-to-end delivery of continuous actions is guaranteed to be exactly once since each trigger creating transformed version of tables always writes data replicated too and only once.

Guarantees' implementations

The Flink isolation level stated for the client requests is automatically enforced by the functioning of requests to operator that are obligated to be single sited (single partition). To provide durability Flink employs the checkpointing technique. Watermarks are instead used to provide end-to-end order and discard late arrived messages. The end-to-end delivery is instead derived by the logical snapshots the systems employs.

Spark, as Flink does, implements the durability by meas of checkpointing and defines the replication consistency level as recoverable – same state since it employs passive replication. Action.

The Delta Lake layer, without considering the underlying filesystem guarantees do not provide many guarantees. It only enforce isolation by means of MVCC.

In the case of Kafka we have instead a wider range of guarantees. Atomicity gets implemented by 2PC protocol and isolation with MVCC. Durability is instead ensured by replication which consistency is ensure by means of passive replication implementation.

Kafka streams add to the guarantees provided by Kafka those for continuous actions. Replication is handled in the same way, by replication and consistency of replicas is ensured by passive replication. Watermarks are employed as in Flink to ensure end-to-end order meanwhile the guarantee of exactly once delivery is provided by means of transactions.

VoltDB, as NewSQL database, has been built to bring the strong guarantees of classical databases at scale. It employs coordination-free techniques such as scheduling to both provide atomicity and ensure consistency constraints are checked. Indeed, when the system is able to execute a transaction on single sites it leverages a careful scheduling to run in parallel without coordination still providing the highest guarantees. When the transaction operations fail to be scheduled in a safe way it falls back to the standard coordination protocol of 2PC. Isolation is guaranteed to be serializable still by scheduling of transactions when it is possible. Again when this is not the case the system falls back to classic techniques such as 2PL and MVCC. Data durability is ensured by means of replication and replicas are coordinated for an active replication strategy by a single leader.

Chapter 7

Related work

The presented work analyzed modern data-intensive tools ranging from pure stream processors to the new generation database management system. Given the interdisciplinary nature of this thesis, the related work hence refers to both these domains that here below will be discussed.

Database systems are born by necessity to save structured data and relate them to each other. Since the very first designed system, SystemR [ABC$^+$76], this type of data-intensive tool has improved by increasing performance and enforcing higher transactional guarantees but still remaining bound to a centralized architecture. The impossibility to efficiently scale out this structure led to the emergence of NoSQL databases. The compromise they imposed to achieve scalability was to weaken, if not almost totally cancel, ACID guarantees, one of the classic OLTPs most desirable feature. With this database generation, the application side interaction model has also changed by removing support for the SQL standard language. It favored an object-oriented model [ADM$^+$90] integrated with programming languages and platforms that unfortunately introduced also other problems like vendor lock-in [Kra14] [BBSR13]. Moreover, it has shifted part of the complexity to managing transactions and consistency on the application side.

Later, a return to SQL semantics guided the birth of the subsequent generation. With the name of NewSQl, these new data engines "preserve SQL and offer high performance and scalability, while preserving the traditional ACID notion for transaction" (from [Sto12]). In fact, the objectives pursued during their conception were to bring back the benefits of SQL standardization, typical of the legacy generation, and also to achieve the scalability feature of NoSQL.

Both in the academic field with prototypes such as H-Store [KKN$^+$08] and in the enterprise with Google Cloud Spanner [CDE$^+$13], NewSQL have proved to be the optimal approach for high performance, horizontal scalability and OLTP-style transactional guarantees. However, this new kind of systems has required the whole restructuring of the initial architecture so to turn the original limitations of distributed environments into performance and reliability advantages [SMA$^+$07]. The correctness guarantees of concurrent transactions' execution in a distributed environment have been one of the most difficult problems to solve. Models such as SAGAS [GMS87] have proved not to the right solution, at least not for databases. With the increase in conflicts' rate, caused by the high number of concurrent operations, the system does not scale accordingly and the overhead introduced by the changes' rollback becomes unsustainable. The initial difficulty in not being able to efficiently enforce guarantees when the computation is distributed derived from their implementations, especially for those originally employed for atomicity and isolation.

The study and redefinition of existing conflicts phenomena have permitted to move away from anomalies' definitions that for the classic database generation were in essence implementation-dependent (es. locking as the only possibility to achieve isolation) [BBG$^+$95]. So, with definitions no more based on isolation models inadequate to distributed contexts, the new coordination-free methodology born [BFF$^+$14]. A careful scheduling of transactions enable an execution free from any checks that, by design, can not occur. This ease the replication management and promote with the concept of Highly Available Transactions [BDF$^+$13] scalability without compromising performance. The coordination-free strategy is enabled by the profiling of applications to categorize them in terms of type performed transactions (single-site, one shot, general) so to implement coordination avoidance; the key factor that allowed the birth of NewSQL databases, those systems this thesis work has analyzed.

In the last decade the produced amount of data and related fast-processing requirements determined an hectic development in stream processing technologies. Many are the systems that both academia and the industry have proposed to overcome databases' limitations. Stream processors evolution can be categorized in two generations. The first, studied by [CM12] date back to mid 2000s and had two main objectives. Firstly, they outlined the abstractions to define the semantic of queries over data streams. Secondly, some of them focused on the ability recognize patterns of interest from raw data, the complex event processing systems (CEP) [ENL11].

Stream processors introduced SQL-like declarative languages to let describe processing required on input streams so to produce either one or more output flows [BBD$^+$02]. The Aurora stream-processing model was the first introducing the concept of directed operators ' graph [ACÇ$^+$03] and to carry out overall flow transformation by multiple chained operators deployed on different servers [AAB$^+$05].

This new paradigm has been one of the leading ideas that drove the second-generation of stream processors development.

The research on Big Data processing investigated new models to handle efficiently the computation on large volumes of streaming data over distributed environments. One of the most important programming model the second generation of stream processor introduced was programming model of MapReduce [DG08]. This functional abstraction did automate the deployment process of computation portions towards the data locations and the aggregation of the sparse obtained results. Later, the expressiveness of MapReduce was enhanced to enable arbitrary complex directed graphs of operators to be specified [ZCF$^+$10]. In addition to this, fault tolerance techniques have been developed to overcome possible hardware failures. Due to the long-running computations and the large clusters made out of several servers, the probability of hardware failure is actually a certainty. Thus, since it is not feasible to restart the total processing since the beginning after failed hardware repairs, fault-tolerance techniques such as Resilient Distributed Datasets (RDD) [ZCD$^+$12] have been developed.

The future generation of data-intensive tools that is expected to rise will be likely driven by the convergence of these two worlds. Proofs of this appear in both the databases and stream processors literature. Newly proposed tools are making efforts to implement features that were initially proper to the opposite area they come from. The S-Store NewSQL database, an evolution of H-Store, aim to combine both OLTP transactions and stream processing by means of a push-driven trigger approach [CDK$^+$14]. Also, with the FlowDB prototype there is the effort from the stream processing community to propose a new model able with the objective to integrate into distributed stream processor data management capabilities [AMC17].

Chapter 8

Conclusions

The objective of this thesis work was to study modern data-intensive tools so as to understand their features in terms of functionalities, architecture, and the adopted paradigms in data management and processing. The performed analysis have been supported by the development of a modeling framework able to abstract similar concepts and highlight systems' characteristics. To experiment the boundaries of frameworks' expressiveness and discover their limitations we have developed a set of tests with the data-intensive tools analyzed.

During the development of data-intensive applications, there were some recurring constant challenges. At first, we had to rethink the interaction modality completely, since stream processors are asynchronous systems. Our objective was to re-implement employing them an application based on synchronous interactions. Thus, we converted queries by materializing them as input data. This adaptation allowed us to obtain the same interaction semantic though ad hoc implementation of the application behavior. The state was managed by map functions implemented to mimic the original application queries' behavior. Each time new input data, on behalf of queries invocations, was sent to the system, these functions did handle them accordingly and also considered their internally stored state in the output generation.

The second challenge we faced was to imitate the SQL JOIN operation. We encountered many difficulties in re-implementing this functionality. Stream processors do not have a uniform set of APIs to save a state and to perform databases-like operations such as joins between tables. We solved this issue through the equivalent functionality of event-enrichment. With this procedure, a map function attaches additional information to the data flowing in a stream. Its output is an "enriched" stream that contains initial values expanded with other information.

The last of the recurring problems we encountered was to reproduce the checks made in between synchronous interactions that took decisions based on a fetched value. In the original scenario, since interactions were synchronous after they retrieved a result from the database, they could decide how and if to perform the following queries. However, this modality was not compatible with asynchronous systems. We implemented an ad hoc solution to emulate this condition-based behavior using a *map* function that, by evaluating its internal state, behaved in the same way.

The conducted experiments, combined with the developed modeling framework, supported the systematic analysis of systems' features. Moreover, the models developed have been useful to organize the results and discoveries done into an accurate characterization that captures common aspects and diversities of the data-intensive tools. We summarize them below.

All the analyzed data-intensive tools rely primarily on the same abstract concepts. The notion of time has been proven to be supported and almost equally employed by every data-intensive tools. All of them exhibit the state notion even though, especially for non-databases ones, it appears by weaker data structures. Inputs, transformed, and data outputs are all supported. VoltDB provides strong support to standalone interactions and a bit less flexibility when dealing with streams. On the contrary, stream processors are characterized in the opposite way. The functionalities in data allocation such as partitioning and replication are concepts common to all the systems, hence demonstrating that even if data management and data processing paradigms differ, there is a common understanding in data allocation for distributed scenarios. The data schema concept is more rigid in VoltDB since it strongly relies on tables and SQL logic without leaving room to custom data types. Stream processors instead allow custom data schema definitions but also started to mimic the data management style of databases by supporting queries expressed with SQL-like APIs. All the systems, in general, employ administrative actions to alter the computational topology and data schema. But how systems interpret DS statements to generate administrative activities outline the limitation of the system like Flink and Spark to be instructed only through the deployment of application code. Kafka and VoltDB, instead, support both simple client requests and code deployment for administrative actions. The simple output actions, such as those to fetch data once, are supported by the majority of data-intensive tools except Spark. Overall, all systems support continuous data actions. The transaction concept is the crucial factor that outlines the division from stream/batch processors and database systems. However, recent projects such as the Streaming Ledger from Flink and Delta Lake from Spark, delineate the efforts made by the respective communities in providing support to stronger guarantees even for those systems originally focused on pure data transformations. Lastly, for the computational topologies, systems have been classified as statically or dynamically deploying the computation.

The analyzed systems have been modeled and also characterized by their architectures and allocation strategies. All of them rely on resource managers to organize, monitor, and coordinate execution on the cluster. Made exception for VoltDB that is limited to the use of its internal resource manager, all the other systems also support external tools. In general, all the data-intensive tools employ worker managers, software processes on each node to monitor and coordinate execution as well as employ execution slots to carry out the processing tasks. The modification of available resources is another critical aspect of the flexibility and scalability of the architectures. Kafka and VoltDB support resources modification at run-time while the other platforms require to restart the running application to then re-adjust the workload distribution. We also evaluated the functioning paradigms adopted to keep stationary either data or the computation. Flink and Kafka Streams maintain stationary the computation while instead Spark, Delta Lake, and VoltDB do the opposite moving it towards the nodes. Flink and Spark require the deployment of application code to receive new instructions; for the other systems is instead an optional feature.

The analysis highlights a convergence under some functioning mechanisms and concepts in data-intensive tools. Each of them shows support for stream processing and state management. However, APIs to handle the internal state are still heterogeneous with each other. Systems demonstrate the increasing willing in supporting tables. They leverage SQL-like procedures to query both state and streams. There is a good shared understanding of how data allocation strategies, such as partitioning and replication techniques, enable high performance and efficient workload distribution. However, there is still a missing convergence in transactions, and isolation guarantees support. It is noteworthy that the development of data-intensive tools is carried out by different companies. Each of them decides the features and guarantees to implement in its system according to the market objectives it has. There are tools that, despite the good marketing claiming for "ACID guarantees", actually weakly support them and only provides specialized implementations that can not be generalized for non-specific use cases. Nevertheless, coordination-free approaches are increasingly used by multiple systems since they allow to provide strong isolation guarantees without synchronization protocols. In the future, we expect this strategy to become increasingly popular and more studied. Most likely, this paradigm will be widely adopted since it enables scalability and performance increase without compromising guarantees in the concurrent execution, a key factor for the next-generation data-intensive tools development.

This work focused on expressiveness, programming, and execution paradigms of data-intensive tools. Future research includes the efficiency of employed distributed algorithms. It would be helpful to understand how different implementations of the same feature can impact on performance. After this initial step, it would also be interesting to compare and characterize data-intensive tools in terms of performance in different application scenarios. Lastly, the analysis could be expanded by the inclusion of NoSQL databases, whose characterization would be interesting in terms of supported features against provided execution guarantees.

Bibliography

[AAB$^+$05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[ABC$^+$76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[ACÇ$^+$03] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.

[ADM$^+$90] Malcolm Atkinson, David Dewitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. The object-oriented database system manifesto. In *Deductive and object-oriented databases*, pages 223–240. Elsevier, 1990.

[ALO00] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 67–78. IEEE, 2000.

[AMC17] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. Flowdb: Integrating stream processing and consistent state management. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 134–145. ACM, 2017.

[BBD$^+$02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[BBG$^+$95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[BBSR13] Alexandre Beslic, Reda Bendraou, Julien Sopenal, and Jean-Yves Rigolet. Towards a solution avoiding vendor lock-in to enable migration between cloud platforms. In *MDHPCL@ MoDELS*, pages 5–14. Citeseer, 2013.

[BDF+13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[BFF+14] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, 2014.

[BHJ15] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer, 2015.

[Bre00] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[CDK+14] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, et al. S-store: a streaming newsql system for big velocity applications. *Proceedings of the VLDB Endowment*, 7(13):1633–1636, 2014.

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[dA18] data Artisans. White paper: data artisans streaming ledger - serializable acid transactions on streaming data. 2018.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[ENL11] Opher Etzion, Peter Niblett, and David C Luckham. *Event processing in action.* Manning Greenwich, 2011.

[GLPL14] Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *2014 IEEE world forum on internet of things (WF-IoT)*, pages 241–246. IEEE, 2014.

[GMS87] Hector Garcia-Molina and Kenneth Salem. *Sagas*, volume 16. ACM, 1987.

[GP10] Carles Gomez and Josep Paradells. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine*, 48(6):92–101, 2010.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[KLC+16] Hyoung Seok Kang, Ju Yeon Lee, SangSu Choi, Hyun Kim, Jun Hee Park, Ji Yeon Son, Bo Hyun Kim, and Sang Do Noh. Smart manufacturing: Past research, present findings, and future directions. *International Journal of Precision Engineering and Manufacturing-Green Technology*, 3(1):111–128, 2016.

[Kle17] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[Kra14] Nane Kratzke. Lightweight virtualization cluster how to overcome cloud vendor lock-in. *Journal of Computer and Communications*, 2(12):1, 2014.

[Lam78] L Lamport. Time, clocks, and the ordering of events in a distributed system, 1978.

[Lea10] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[MHL+92] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[MWMS14] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615. IEEE, 2014.

[N+16] Amy Nordrum et al. Popular internet of things forecast of 50 billion devices by 2020 is outdated. *IEEE spectrum*, 18(3), 2016.

[New15] Sam Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[PV16] M Vukolić P Viotti. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 2016.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[Sto10] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.

[Sto12] Michael Stonebraker. New opportunities for new sql. *Communications of the ACM*, 55(11):10–11, 2012.

[SW13] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[TDW+12] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[Thö15] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[TVS07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[VSS02] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for etl processes. In *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 14–21. ACM, 2002.

[X3.92] ANSI X3.135-1992. Database Language — SQL. Standard, American National Standard for Information Systems, 1992.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

[ZFY16] Kaile Zhou, Chao Fu, and Shanlin Yang. Big data driven smart energy management: From big data to big insights. *Renewable and Sustainable Energy Reviews*, 56:215–225, 2016.

Ringraziamenti

In primo luogo desidero ringraziare il prof. Alessandro Margara che, permettendomi di lavorare al suo progetto, mi ha guidato in un lavoro svolto insieme con sua grande disponibilità, pazienza, cortesia e attenzione sia negli esperimenti condotti che nella stesura di questa tesi. Ringrazio il prof. Gianpaolo Cugola per la conoscenza che mi ha trasmesso e per l'aiuto fornito nelle sfide progettuali emerse durante il percorso.

Un sentito ringraziamento a tutti quegli amici che mi sono stati vicini, che sono stati parte di momenti di vita indimenticabili, arricchiti grazie a loro da grandi emozioni. Grazie a Gianfilippo, Giada, Eleonora, Veronica, Marija, Burcu, Luca, Enrico, Riccardo e Giulia per avermi incoraggiato tra le tante sfide e difficoltà di questi anni aiutandomi ad affrontarle con tenacia.

Un grazie con affetto a quelle zie e a quegli zii che hanno creduto in me, che hanno saputo incoraggiarmi e sostenermi quando più ne ho necessitato. Grazie a Luisa, Ciccio, Emanuela, Emilio e Simona per aver seguito con interesse il mio percorso di studi, per averne preso parte a vostro modo e per aver condiviso con me la gioia dei momenti in cui sono riuscito nelle sfide affrontate.

Un ringraziamento speciale lo rivolgo alla mia fidanzata Ilaria che con la sua costante presenza, l'affetto e la comprensione nei miei riguardi, è stata sempre al mio fianco condividendo le gioie e le sofferenze di questi anni accademici.

Ringrazio mia sorella Silvia per la comprensione avuta nei miei confronti e per il suo aiuto sempre pronto nelle situazioni in cui da solo non sarei riuscito.

Infine, vorrei volgere un particolare ringraziamento ai miei genitori per avermi dato la possibilità di studiare, facendo sacrifici a loro volta, e per aver creduto in me durante tutto questo percorso universitario, impossibile senza l'aiuto da loro giunto in molteplici forme.

Senza di voi, nessuno dei risultati che ho ottenuto sarebbe stato lo stesso.