State of the Art on: Towards a unifying model for data-intensive applications

Nicolò Felicioni, nicolo.felicioni@mail.polimi.it

1. Introduction to the research topic

In recent years, the need for software applications that can handle large amounts of rapidly varying and heterogeneous data has become greater and greater. We refer to these applications as **data-intensive**. They require a distributed software system to store and process the large amount of data, in order to exploit the resources of many interconnected computers. The research areas involved in this topic are mainly the database research area and the distributed systems and algorithms one. The former is the area that deals with storing and managing data developing databases and database management systems (DBMS), while the latter has the main focus on the creation of distributed platforms and algorithms to process and analyze large quantity of data.

Among the most relevant journals[1] and conferences[2] related to the research topic, there are:

- ACM SIGMOD Conference
- ACM Symposium on Principles of Database Systems
- ACM Transactions on Computer Systems
- EuroSys - the European Systems Conference
- IEEE International Conference on Data Engineering
- IEEE Transactions on Parallel and Distributed Systems
- USENIX Annual Technical Conference
- USENIX Symposium on Operating Systems Design and Implementation
- The VLDB Journal
- VLDB Conference

1.1. Preliminaries

1.1.1 Data management preliminaries

A fundamental concept in database research is the **relational model**, introduced by Codd [6] in the 1970. In this paradigm, data are conceptually represented in form of an n-ary *relation*, that is a set of tuples $(d_1, d_2, ..., d_n)$, where d_i is an element of the correspondent data domain D_i. Relations are represented in tabular form, where rows are tuples and every column corresponds to an attribute, which has a particular data domain. Every row is uniquely identified by a key. Also, relational database usually provide so-called *transactions*. A use case of a database with necessity of processing multiple transactions is called On-line Transaction Processing (OLTP). A transaction is a unit of work that must be atomic (it must complete in its entirety or it must abort in its entirety), consistent (it must be compliant with eventual integrity constraints of the database), isolated (concurrent transactions should

[1] According to the Impact Factor (citation/documents in the last two years) taken from ScimagoJR, `https://www.scimagojr.com`

[2] According to the GII-GRIN-SCIE Conference Rating, `http://gii-grin-scie-rating.scie.es/conferenceRating.jsf` and the H5 index taken from Google Scholar, `https://scholar.google.com`

not interfere among each other) and durable (the effect of a transaction must be stored persistently). This set of properties is usually referred to as ACID [8]. A prototype of a relational database was published in 1976 and it was called System R [1]. The language used to manipulate data present in the database was SQL [3], introduced in 1974 for System R. From then on, relational DBMS almost always used SQL or a SQL dialect to manipulate data. Because of this, when non-relational DBMSs were introduced, they were called **NoSQL** [11]. By definition, NoSQL is a highly heterogeneous category since it is formed by all the data models that are not relational. NoSQL systems were born from the desire to overcome the scalability issues of the classical relational databases, at the cost of relaxing some constraints on strong data consistency and giving up some transactional guarantees.

In last years, a new category of databases started to be relevant in the research domain. These so-called **NewSQL** systems [14] try to reconcile the high scalability and the high performance of NoSQL databases with the guarantees given by the classical relational model.

1.1.2 Data processing preliminaries

In the data processing domain, the increasing size of data motivated the development of a new kind of systems explicitly designed for distributed processing in large-scale compute infrastructures. These systems all trace their roots to the **MapReduce** paradigm [5]. MapReduce is a programming model introduced by Google in 2004 that enables application programs to be written in terms of high-level operations (Map and Reduce) on immutable data, while the runtime system controls scheduling, load balancing, communication and fault tolerance.

The first open-source implementation of the MR[3] paradigm was Hadoop in 2006. From this point on, data processing systems evolved to overcome the limitations of the paradigm (e.g. being constrained to use only Map and Reduce operations) and resulted in the development of Spark (for batch data processing) and Flink (for streaming data processing).

1.2. Research topic

Data-intensive applications are applications that have as their primary challenge the management of large amounts of data, that are rapidly changing and that are highly heterogenous[10], as opposed to compute-intensive applications, where the CPU is the bottleneck.

The development of this kind of applications is fundamental to sustain the increase of volume, production velocity and heterogeneity of data that will arise in the next future [4]. For example, the Internet of Things (IoT) advent is bringing billions of smart devices that will produce enormous quantity of data [12]. Data produced by IoT devices are also rapidly changing and often have to be analyzed in real-time. Another trending sector is the one of autonomous vehicles [7], that contributes to this phenomenum of "data flooding" making cars continuously produce data. Apart from machine-created data, there are also human-created data, like social networks' data, that billions of users created and continue to create, or, more in general, all the data created while surfing the web. These data are usually collected for advertising and for training recommendation systems of various websites [13]. This "information overload" made business strategies data-driven, i.e. taking into account data collected by customers to make strategies and take business decisions. For these reasons, creating software systems that are scalable and capable of handling large quantities of data is relevant in modern computer science research.

[3] Short for MapReduce.

2. Main related work

2.1. Classification of the main related work

In the database world, a first classification of the newly developed and highly scalable databases can be the division between NoSQL and NewSQL. Then, within the NoSQL family systems can be distinguished based on the data model they adopt, that can be document-based, graph-based, and others. In the data processing domain, it is possible to distinguish the two main category of batch processing, that is a category of tools that process batches of data all together, and stream processing, that is a category of tools that process data in real-time, one datum after the other.

A possible classification of some relevant systems present in literature or industry is shown in Table 1.

DATA-INTENSIVE CLASS		SYSTEM
Batch processing		Spark, Hadoop
Stream processing		Flink, Kafka streams
NewSQL		VoltDB, Spanner
NoSQL	Document-based	MongoDB
	Wide column	Cassandra, Bigtable
	Time series	InfluxDB
	Key-value pairs	Redis
	Graph-based	Neo4j, Dgraph

Table 1: A classification of some relevant systems.

2.2. Brief description of the main related work

2.2.1 Batch processing

In the data processing domain, the increasing size of data motivated the development of a new kind of systems able to process a data amount that is so large that can not fit into a single machine and therefore they were explicitly designed for distributed processing in large-scale compute infrastructures. Batch processing systems are a special kind of data processing systems, called like this because they take data statically loaded all in one time (a batch), but some of them also has support for streaming data (i.e. a flow of data that is dynamically produced by a source and reach the system at high speed). These systems all trace their roots to the MapReduce paradigm, a programming model introduced by Google in 2004. In the MR research paper [5] there were present some architectural characteristics that were destined to influence deeply all the later batch processing systems. For example, the fact that a job submitted by the user was split in various tasks and a runtime system, that is not under user's control, should take care of the scheduling on the various nodes. Thanks of the scheduling mechanism, the system could make various optimizations –like load balancing– and it was able to provide fault-tolerance (important aspect since the platform was designed to run on commodity hardware[4]) by dynamically switching to another working node when a node stops running. The first open-source implementation of Google's MapReduce was Hadoop, released in 2006. Despite being promising, the MR approach had also limitations, such as the fact that it was a very fixed and limiting paradigm, since it constrained users to create programs only in terms of the *Map* and the *Reduce* primitives. Another relevant limiting factor was that every stage had to read and write data from disk, that acted as a bottleneck in the processing. The desire to overcome these limitations resulted in the development of Spark.

[4]Cheap, non-specialized hardware.

Apache Spark[5] [16] is a cluster computing platform to process batch of data in a distributed and parallel fashion. It was inspired by Google's MapReduce and Apache Hadoop, but it has some improvements with respect to the previous architectures that make Spark more suitable for modern hardware. Indeed, Spark can exploit modern main memory capabilities thanks to its efficient caching mechanism, that makes it faster than Hadoop up to 100 times in some applications. In addition, Spark introduces new operators –other than to map and reduce– and can process an arbitrary number of stages, supporting also iterative processing. The scheduler ensures that the computation takes place close to the data and this strongly optimizes the processing. Spark supports also stream processing through micro-batching, meaning that the engine accumulate the streaming input up to a certain size to form a micro-batch and then it processes the batch in a batch processing manner.

2.2.2 Stream processing

Stream processing systems are systems that deal with data streams, that are unbounded flows of data. Usually, data streams have to be processed timely since often data coming from a stream are not valuable anymore after a certain period of time. Examples of stream processing applications are real-time computer performance monitoring or real-time stock market analytics.

Apache Flink[6] [2] is one of the most relevant cluster computing platform for stateful computations over data streams. The main difference with Spark is that Flink does not have a scheduler and a job is not split into stages that are scheduled. Instead, a pipeline formed by the selected operators is deployed as soon as the job is submitted. This static piplined approach can decrease the latency but also decrease the throughput of the application. Therefore, Flink is preferred when you have a strict requirement on low latency. Flink supports also batch processing by simply streaming the static batch of data. In this case, though, all the various optimization due to a dynamic scheduling are lost, since Flink statically deploys a fixed pipeline.

2.2.3 NoSQL

The NoSQL family of databases is difficult to define precisely, because in literature the definition can be found with disparate meanings. Usually, the term is used to indicate a database that is non-relational, can be distributed and it is horizontally scalable[7]. But this means that all the databases that have data models that are not relational could be called NoSQL, from the ones that are key-value pairs store to the document-based systems. These systems were born from the desire to overcome the scalability issues of the classical relational databases, at the cost of relaxing some constraints on strong data consistency and giving up some data guarantees. Indeed, it is almost always the case that a NoSQL system does not have transaction with fully ACID guarantees.

Redis[8] is the most known example of a NoSQL key-value database. It has very high perfomance thanks to the memory-oriented execution, a feature that is typical of NoSQL systems. It supports also persistence with the mechanisms of database snapshots and append-only files (AOF). In Redis a value can be a complex data structure, for example list, hash table, set, string and others. It can be replicated and partitioned, but it does not have any transactional guarantee. A completely different approach, but still in the very broad category of NoSQL, is the one adopted by MongoDB.

MongoDB[9] is a database that stores data in form of a JSON-like documents. It supports replication and partitioning. Thanks to the document data format, it is a very flexible datastore that is able to adapt very well to possible schema changes. In fact, MongoDB is called schema-less, meaning that there is no constraint on the structure of the documents that it stores. It supports distributed multi-document transactions (from version 4.2[10]) with slightly less guarantees than the ones of a classical relational database, but it does not have joins. A

[5] https://spark.apache.org/docs/latest/
[6] https://ci.apache.org/projects/flink/flink-docs-release-1.10/
[7] https://nosql-database.org
[8] https://redis.io/documentation
[9] https://docs.mongodb.com
[10] https://docs.mongodb.com/manual/release-notes/4.2

document, in order to reference to another one, should embed the other document (document embedding) or it should contain a field with the primary key of the other one (document linking).

2.2.4 NewSQL

NoSQL systems are valuable tools, especially in those cases when it is important to handle in a distributed fashion large amounts of data. Another big achievement from those kind of systems is the high performance reached in the data management. Albeit there are benefits, there are some weaknesses too. The biggest flaw is that they usually do not support full ACID transactions. However, nowadays OLTP scenarios with great quantity of data are a matter of growing relevance and researchers tried to find a new solution for an efficient distribution of the classical relational database model, with full support for transactions and their guarantees. These types of solutions are called NewSQL, since they try to make scalable as much as possible the traditional relational systems, while preserving all their guarantees. The NewSQL implementations are heterogeneous, so now we present the VoltDB solution as representative for the category.

VoltDB[11] [15] is a main memory transactional database, designed to be highly available and horizontally scalable, while providing all the transactional guarantees typical of classical relational databases. It was the evolution of the H-store [9] research project and it uses as transactions only *stored procedures* defined a priori. The transactions are analyzed at compile time, categorized and stored inside the runtime system. With this approach VoltDB achieves ACID guarantees without paying much coordination overhead.

2.3. Discussion

The presented data systems offer solutions to solve specific data processing and management tasks, but often requirements of a data-intensive application can be more heterogeneous and therefore they cannot be satisfied by any of these systems alone. Because of that, developers in practice build complex architectures that combine multiple systems and then implement application logic in order to orchestrate their interaction. The problem is that, in doing so, they move out of the disciplined programming paradigms of individual systems and lose their benefits in terms of guarantees on the data and transparent deployment and communication. In addition, integrating data systems together necessitates a deep understanding of their semantics, workload assumptions, performance characteristics, deployment strategies, and configuration opportunities. For this reason, the development of a formal *modeling framework*, which defines a high-level programming interface and captures the functionalities and characteristics of data systems, is necessary. Also, this kind of systems usually present intersections among them, therefore a unifying model can be useful to capture the semantics of the converging concepts of different systems. This modeling framework can be a first fundamental step in the direction of a change of paradigm, that leads to a new approach for designing data-intensive application. Following this new paradigm, developers should define the application specifying for example the data to be stored, the guarantees and the performance requirements and a runtime system should determine the best strategies for data format, replication, partitioning and guarantees implementation. In this way, developers no more have to deal with trying to put different and indipendently developed systems together in a sort of "software collage", where the formal guarantees provided by the single systems could be lost.

[11] `https://docs.voltdb.com`

References

[1] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS) 1*, 2 (1976), 97–137.

[2] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[3] Chamberlin, D. D., and Boyce, R. F. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control* (1974), pp. 249–264.

[4] Cilloni, S. Towords a unifying modeling framework for data-intensive tools. Master's thesis, Politecnico di milano, 12 2019. Supervisor: Alessandro Margara.

[5] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters.

[6] EF, C. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (1970), 377–387.

[7] Gerla, M., Lee, E.-K., Pau, G., and Lee, U. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *2014 IEEE world forum on internet of things (WF-IoT)* (2014), IEEE, pp. 241–246.

[8] Haerder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR) 15*, 4 (1983), 287–317.

[9] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P., Madden, S., Stonebraker, M., Zhang, Y., et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1496–1499.

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

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

[12] Nordrum, A. Popular internet of things forecast of 50 billion devices by 2020 is outdated (2016). *Dosegljivo: https://spectrum. ieee. org/tech-talk/telecom/internet/popular-internet-ofthings-forecast-of-50-billion-devices-by-2020-is-outdated.[Dostopano: 11. 8. 2017]* (2017).

[13] Ricci, F., Rokach, L., and Shapira, B. Introduction to recommender systems handbook. In *Recommender systems handbook*. Springer, 2011, pp. 1–35.

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

[15] Stonebraker, M., and Weisberg, A. The voltdb main memory dbms. *IEEE Data Eng. Bull. 36*, 2 (2013), 21–27.

[16] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 15–28.