A Unifying Model for Distributed Data-Intensive Systems

Alessandro Margara
alessandro.margara@polimi.it
Politecnico di Milano
Italy

ABSTRACT

Modern applications handle increasingly larger volumes of data, generated at an unprecedented and constantly growing rate. They introduce challenges that are radically transforming the research fields that gravitate around data management and processing, resulting in a blooming of distributed data-intensive systems. Each such system comes with its specific assumptions, data and processing model, design choices, implementation strategies, and guarantees. Yet, the problems data-intensive systems face and the solutions they propose are frequently overlapping.

This tutorial presents a unifying model for data-intensive systems that dissects them into core building blocks, enabling a precise and unambiguous description and a detailed comparison. From the model, we derive a list of classification criteria and we use them to build a taxonomy of state-of-the-art systems. The tutorial offers a global view of the vast research field of data-intensive systems, highlighting interesting observations on the current state of things, and suggesting promising research directions.

CCS CONCEPTS

• General and reference → Surveys and overviews; **• Information systems → Data management systems**; **• Computing methodologies → Distributed computing methodologies**.

KEYWORDS

Data-intensive systems, distributed systems, distributed database, data management, data processing, big data, model, survey

ACM Reference Format:
Alessandro Margara. 2022. A Unifying Model for Distributed Data-Intensive Systems. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27-30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3524860.3539782

1 INTRODUCTION

Data is a precious resource in today's society as it guides decision-making and affects many aspects of our everyday life. Software applications increasingly become *data-intensive* [12]: they receive large volumes of data continuously produced by people, IoT devices, and other software systems. They need to store data, analyze and integrate it, serve it to a multitude of users, and take automated decisions. Data characteristics such as its scale, the frequency at which it is produced, and the inherent distribution of the actors that generate and consume it make the development of data-intensive applications challenging. As a consequence, the last two decades have seen a blooming of distributed platforms that aim to simplify the development and operation of data-intensive applications to make them more efficient and cost-effective. These systems originate from research and development efforts in various communities, in particular those working on database and distributed systems.

The design of modern distributed databases has been widely affected by mutating workloads, requirements, and operational conditions, including the need to handle unstructured and heterogeneous data, the increased number of concurrent users, their geographical distribution, and the increased availability and reduced costs of main memory and parallel hardware [21]. This led to the development of new classes of databases, some of them providing simple and flexible data models and trading consistency guarantees and strong (transactional) semantics for horizontal scalability [8], others introducing new design and implementation strategies that better adapt to modern compute infrastructures [20].

In parallel, MapReduce [9] pioneered a whole new class of systems for processing static and streaming data at scale on a cluster of machines [6, 25]. These systems allow developers to focus on the logic of their processing tasks and delegate (at least in part) distribution, synchronization, scheduling, and fault tolerance concerns to the system run time.

More in general, the requirements of data-intensive applications are continuously pushing the limit of technology and driving the exploration and exploitation of new approaches that go beyond traditional categories. For instance, many data processing platforms implement libraries to process relational data, thus proposing themselves as suitable engines to execute complex relational queries and making the distinction with database technologies more blurry. Data stores such as VoltDB [22] and S-Store [7] exploit user-guided data partitioning to avoid blocking coordination and to support stream processing features within a relational database core. Queuing services such as Kafka [13] offer persistency and provide libraries with processing and querying abstractions [5].

In summary, a multitude of distributed data-intensive systems proliferated over the years. Each builds on different assumptions, for instance regarding user interaction and execution environment; each adopts different design and implementation strategies, for instance with respect to synchronization and coordination; each offers different guarantees, for instance in terms of processing semantics, data consistency, and fault tolerance. Yet, once we look beyond system-specific details, we can observe that recurring problems and solutions.

Moving from the above motivations, we recently designed a unifying model for data-intensive systems that integrates all key assumptions, design choices, implementation strategies, and guarantees into a coherent view [16]. The model dissects data-intensive systems into a collection of abstract components that cooperate to offer the system functionalities. It precisely defines each component in a system-independent and unbiased way. In doing so, it promotes the understanding of the possible strategies for developing data-intensive systems and the consequences they bring. From the model, we derived a list of classification criteria that we used to precisely describe tens of state-of-the-art data-intensive systems, organizing them into a taxonomy.

In the tutorial, we first present our model and the classification criteria deriving from it (see Sec. 2). Then, we overview our taxonomy of systems (see Sec. 3), highlighting the key common features of each class, and the most notable exceptions in individual systems. In doing so, we guide the attendees through a precise global view of the field of data-intensive systems, their key concerns, recurring strategies of design and implementation, and open problems. Our study of data-intensive systems is the starting point to reflect on the current state of the field and to discuss promising research directions (see Sec. 4).

2 A UNIFYING MODEL FOR DATA-INTENSIVE SYSTEMS

We propose a unifying model that dissects the core aspects of data-intensive systems along multiple dimensions. For each dimension, we extract a list of classification criteria that capture the alternative design and implementation choices we found in existing systems.

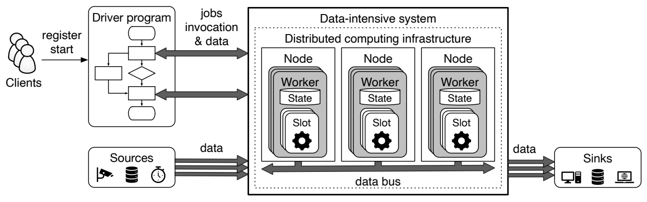

Figure 1: Functional model of a data-intensive system

Functional model. We start from a functional model that introduces the main components that build and interact with a data-intensive system (see Fig. 1). The data-intensive system runs onto a *distributed computing infrastructure* consisting of different *nodes*, each hosting one or more *worker* processes that implement processing and storage functionalities (processing *slots* and *state* stores in Fig. 1). The model defines how external *clients* can interact with the system by defining *driver programs*, which offload *jobs* onto the distributed computing infrastructure. Jobs are split into elementary *tasks* that run within nodes, access their state, and communicate and coordinate among each other by exchanging data over a *data bus*. Jobs may additionally receive data from external *sources* and submit results to *sinks*.

Our functional model captures the presence or absence of these virtual components, their possible interactions, and the strategies for their deployment.

Jobs. We model how different systems define, compile, deploy, and execute jobs. In terms of definition, we capture the key characteristics of the programming abstractions and domain-specific languages that data-intensive systems provide: for instance, this includes the programming paradigm (imperative or declarative), the support for data-dependent control flow and for iterations, if jobs are one-shot or continuous, that is, if they run only once and terminate (e.g., a database query) or if they are deployed within the system and get continuously activated by the arrival of new data (e.g., continuous jobs in stream processing, view maintenance procedures in databases).

In terms of compilation, deployment, and execution, we capture when jobs are compiled, what information is used to optimize the compilation, what are the units of deployment, when and how are deployment decisions taken, how are the resources provided by the computing infrastructure managed.

Data management. Data represents immutable information that tasks exchange to fulfill the job they are part of. We model intrinsic characteristics of data elements, such as their format, structure (if any) and the presence of temporal metadata. We also model characteristics of the communication channel used to deliver data (which we denote the data bus): if it is persistent or ephemeral, partitioned, replicated, directed, unicast or multicast, and the interaction model it induces.

State management. State represents mutable information that tasks can read and modify as part of their execution. As for data, we model characteristics of individual state elements, such as their structure and the way they are physically stored. We further model their visibility (local to a single task or global), the use of state partitioning, and the details about state replication: its goal, implementation strategies, and guarantees in terms of replica consistency.

Tasks grouping. Several systems offer primitives to identify groups of tasks and provide additional guarantees for such groups: group atomicity and group isolation. The former ensures no partial failures for a group of tasks: they either all fail or all complete successfully. The latter limits the ways in which running tasks can interact and interleave with each other. In database systems, these properties are considered part of transactional management, as transactional semantics deals with both atomicity and isolation.

Our model captures the presence of task grouping guarantees, the implementation strategies used to enforce them, and the assumptions on which they build.

Delivery and order. Delivery and order model how the results of jobs execution in terms of output and state changes become visible to external actors, such as clients and sinks. Delivery focuses on individual jobs invocations and studies which guarantees a system provides on their execution in the presence of failures, and under which assumptions. Order, instead, focuses on multiple jobs and defines the order at which their effects become visible.

Fault tolerance. Fault tolerance is the ability to recover from a software of hardware failure. We model fault tolerance in terms of its scope (e.g., recover state or recover the intermediate results of a long-running computation), the assumptions on which it builds, the guarantees it provides (e.g., recovery makes a failure unnoticeable for an external observer, and the algorithms used for its implementation.

Dynamic reconfiguration. With dynamic reconfiguration we model algorithms and strategies that data-intensive systems may enact to adapt to changes in the environment in which they operate. For instance, some systems are capable of automatically scale up (or scale down) if the load suddenly increases (or decreases) to optimize the use of resources.

3 SURVEY OF SYSTEMS

In the tutorial, we use the model in Sec. 2 to survey and analyze many state-of-the-art data-intensive systems. Based on the classification criteria we derived from the model, we created a taxonomy of systems, as shown in Fig. 2. The tutorial will follow such taxonomy and organize the presentation in a hierarchical way.

At a first level, we distinguish between data management systems, data processing systems, and other systems that do not clearly fall into any of the two categories.

Data management systems offer the abstraction of a mutable state store that many jobs can access simultaneously to query, retrieve, insert, and modify elements. They mostly target lightweight jobs, which do not involve computationally expensive data transformations and are short-lived. Conversely, data processing systems perform complex computations (long-lasting jobs) on large volumes of data.

For data management systems, we first distinguish between NoSQL systems [8], which offer weak semantics for replication and group operations, and NewSQL systems [20], which offer strong semantics. Within each class, we further classify systems based on the model they use to represent state. Finally, for system that implement a structured data model with strong semantics, we discuss the implementation strategy to obtain such semantics: through time-based protocols [3], deterministic execution [23], explicit partitioning strategies [22], or primary-based protocols [24].

For data processing systems, we identified three main lines of research and implementation. Dataflow systems organize tasks into an acyclic graph, where edges represent the flow of data from task to task. We observed a clear distinction between dataflow systems that perform deployment at the granularity of individual tasks (when they are ready to be executed) and dataflow systems that perform deployment at the level of entire jobs (when the jobs are submitted by clients). A third class of systems focuses on graph data structures and algorithms.

Concerning the remaining systems, we distinguish among those that implement data processing or computational abstractions on top of a data management core [19], those that explore new programming models [11], and hybrid systems that try to integrate data processing and management capabilities within a unified solution [1, 7].

4 DISCUSSION AND FUTURE RESEARCH

Our unifying model enables us to capture the hidden similarities and subtle differences between systems. The tutorial presents the main messages we derive from our analysis and shows promising strategies for future research directions.

State and data management. We observe that data management and processing systems are mostly complementary with respect to state and data management. The former are designed to handle lightweight jobs that read and modify a globally addressable mutable state, while the latter target computationally expensive jobs that transform input data into output data, and do not consider state at all or consider it only within individual tasks. Due to their orthogonal goals, these systems are frequently used in conjunction within the software architecture of many companies.

Traditionally, their roles were sharply distinct: data management systems handled read-write jobs that mutate the state of the application, while data processing systems performed periodic read-only analytical computations to gather insights from data.

Recent architectural patterns [15], advocate the use of stream processing technologies to continuously execute data analytics and guarantee fresh results. In this role, stream processing systems are starting starting to offer primitives to access the state of their tasks, thus avoiding the need of external systems to store the results of their transformations. In practice, they offer the same abstraction of a key-value data management system. This triggered interesting research on declarative APIs that integrate streaming data and state changes into a unifying abstraction [17]. Few systems such as S-Store [7] and TSpoon [1] further explore the possibility to integrate data management and stream processing workloads.

Further studying the possible integration of stream processing and data management both in terms of programming abstractions and in terms of implementation strategies represents in our opinion an important area of research with vast potential impact.

Coordination avoidance. As data-intensive systems scale horizontally to exploit the resources of many machines, coordination may easily become a bottleneck. As a consequence, avoiding or reducing coordination is a recurring principle in the design of all data-intensive systems [4]. A deep understanding of the performance implications of different coordination avoidance strategies under various workloads is an interesting research problem, which may open the room for dynamic adaptation strategies.

Wide area deployment. Virtually all the systems we analyzed are mainly designed to be deployed within one data center. Some of them support wide area deployment either with some reduction in performance or with relaxed guarantees. At the same time, edge computing paradigm [18] is emerging: it aims to exploit resources at the edge of the network, close to the end users. Designing data-intensive systems that embrace this paradigm and simplify the use of edge resources is an important topic of investigation.

Specialized hardware. The use of specialized hardware was outside the scope of our model, but it is an active area of research, targeting processing accelerators such as GPUs [14], new storage solutions such as non-volatile memory [2], and network solutions such as RDMA [10].

As specialize hardware becomes less expensive and easier to program, studying data-intensive systems that can exploit available hardware resources and easily adapt to new configurations is an interesting and area of research with direct impact on industrial setup.

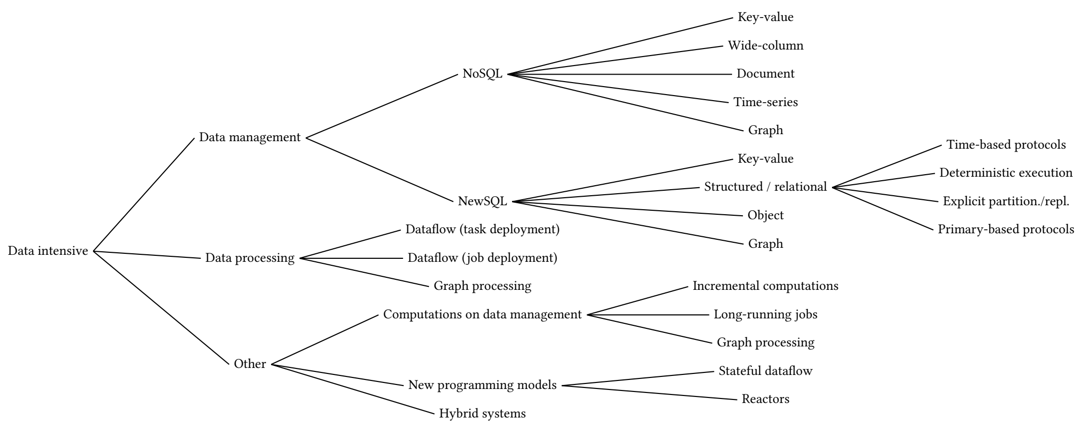

Figure 2: Taxonomy of data-intensive systems

5 CONCLUSIONS

This paper presented the content of a tutorial on data-intensive systems, which is based on our recent work in the area [16]. In the tutorial, we first present a unifying model for highly heterogeneous data-intensive systems. The model provides a comprehensive list of classification criteria to capture the key features of each system. We present a system taxonomy that we build starting from the classification criteria and that we use to overview state-of-the-art systems. Finally, we discuss the main lessons we derived from our analysis and we highlight promising directions for future research.

REFERENCES

[1] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2020. TSpoon: Transactions on a stream processor. *J. Parallel and Distrib. Comput.* 140 (2020), 65–79.

[2] Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In *Proc of the Intl Conf on Management of Data (SIGMOD '17)*. ACM, 1753–1758.

[3] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proc of the Intl Conf on Management of Data (SIGMOD '17)*. ACM, 331–343.

[4] Peter Bailis, Alan D. Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (2014), 185–196.

[5] Bill Bejeck. 2018. *Kafka Streams in Action: Real-time apps and microservices with the Kafka Streams API*. Manning.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.

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

[8] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2, Article 40 (2018).

[9] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[10] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking Stateful Stream Processing with RDMA. (2022).

[11] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *Proc of the USENIX Annual Technical Conf (ATC'14)*. USENIX Assoc., 49–60.

[12] Martin Kleppmann. 2016. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly.

[13] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proc of the Intl Workshop on Networking meets Databases (NetDB)*. USENIX, 1–7.

[14] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB Experience of Building a CPU/GPU Hybrid Database Product. *Proc of VLDB* 14, 12 (2021), 2999–3013.

[15] Jimmy Lin. 2017. The Lambda and the Kappa. *IEEE Internet Computing* 21, 5 (2017), 60–66.

[16] Alessandro Margara, Gianpaolo Cugola, Nicoló Felicioni, and Stefano Cilloni. 2022. A Model and Survey of Distributed Data-Intensive Systems. https://doi.org/10.48550/ARXIV.2203.10836

[17] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proc of the Intl Workshop on Real-Time Business Intelligence and Analytics (BIRTE '18)*. ACM, Article 1.

[18] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *Internet of Things Journal* 3, 5 (2016), 637–646.

[19] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proc of VLDB* 6, 11 (2013), 1068–1079.

[20] Michael Stonebraker. 2012. New Opportunities for New SQL. *Commun. ACM* 55, 11 (2012), 10–11.

[21] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proc of the Intl Conf on Data Engineering (ICDE '05)*. IEEE, 2–11.

[22] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin* 36, 2 (2013), 21–27.

[23] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc of the Intl Conf on Management of Data (SIGMOD '12)*. ACM, 1–12.

[24] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proc of the Intl Conf on Management of Data (SIGMOD '17)*. ACM, 1041–1052.

[25] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.