A unifying model for data-intensive systems

Nicolò Felicioni
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
nicolo.felicioni@mail.polimi.it

***Abstract*—New demands that arose in recent years led to the development of systems able to handle large quantities of data, which can be rapidly varying and heterogeneous. Examples of such systems come from different research areas. The NoSQL databases, developed for scalability and high availability, and the NewSQL databases, which aim to bring transactional guarantees with a new and faster architecture, come from the database research area. Within the data-processing domain, research was conducted on the creation of systems for batch processing, when the data is static, and for stream processing, when the data changes in real-time. All those systems try to solve specific issues. However, nowadays, applications that are called data-intensive have heterogeneous requirements. Hence, developers usually employ several of these tools manually integrated with ad-hoc application logic. With this approach, benefits related to transparent deployment, specific optimizations, and correctness guarantees are lost. In this paper, we noticed how the aforementioned systems have recurring concepts and features, and thus we defined a unifying model to capture them. This model can be used to describe data-intensive systems accurately and also to highlight which are the functional, architectural, and correctness concepts that those systems share.**

***Index Terms*—database, data processing, distributed systems, stream processing, real-time analytics**

I. Introduction

Recently, the need for software applications that can handle large amounts of rapidly varying and heterogeneous data has continually increased.

The reasons for the emergence of such requirements are numerous. Let us take, for example, the *Internet of Things* (IoT) field as one of the most representative. Due to this trending sector, smart sensors spread all over the world are flooding the internet with large quantities of data. Usually, data collected by IoT sensors must be analyzed in real-time. Also, such data are highly heterogeneous. Other emerging fields relevant noting are the spread of smartphones, autonomous vehicles, and many others.

As a consequence, some software applications nowadays have data – the quantity of data, the heterogeneity of data, or the speed at which it changes – as their primary challenge. We call those applications *data-intensive* [20].

In order to satisfy the requirements of this kind of applications, software systems with the purpose of reliably and efficiently handle, store, and process the data started to gain relevance. We call those *data-intensive systems*.

Traditionally, data management was done via *database management systems*. This research area was born from the ideas of Edgar Codd, who introduced the concept of the *relational model* [14]. The software used to access and manage a relational database was called *Relational Database management system* (RDBMS). One of the first RDBMS prototypes was *System R* [4], from the 1970s. Until the 2000s, more or less every newly developed RDBMS was heavily influenced by the design choices made by System R. This led to RDBMSs relying on assumptions made on much older hardware, and therefore to inefficient architectures.This issue amplified with the passing of the years when data-intensive requirements started to arise. For instance, the spread of the web caused a massive increase in data generation. Such a large quantity of data was not expected by the researchers that developed System R. For these reasons, in the 2000s, a new family of database management systems (DBMSs) was born, called *NoSQL*. The name was a statement of purpose that made clear their refuse of the old relational model in favor of a new data-intensive architecture. After this wave, researchers focused on mixing the benefits of the relational model with the ones of the NoSQL systems. These efforts eventually resulted in the development of the so-called *NewSQL* systems [24], more or less during the 2010s.

In parallel, within the distributed systems research area, the development of *MapReduce* [16] in 2004 was a breakthrough for distributed data processing. It consisted of a programming framework, developed by Google, used to parallelize the processing of large quantities of data stored in a distributed cluster of nodes. In this way, it facilitated the extraction of useful information, through functional primitives, from large quantities of data, which is particularly relevant for answering analytical queries. Since the data is stored and fixed during the processing, these frameworks for processing static stored data were also called *batch processing systems*.

From this point, the continuous evolution of distributed data processing systems led to the development of *stream processing systems*, which are systems that process data that arrive in real-time and at a fast rate. The fact that the data is not stored in static batches may cause several issues. This research area emerged because streaming data processing is fundamental in some use cases, e.g., for stock market data processing, where data quickly become stale, and so it is of utmost importance to process them in real-time.

The existing data-intensive systems all provide support in solving specific tasks. However, data-intensive requirements

are often heterogeneous, and therefore can not be satisfied by any of those systems alone. For this reason, in practice, developers must use a complex architecture combining several data-intensive systems and write ad-hoc application logic that dictates their interaction. Nevertheless, in doing so, they lose all the benefits provided by the individual systems in terms of correctness guarantees, performance, transparent deployment, and communication. Furthermore, usually, system documentations are ambiguous and confusing, and this may make harder the integration process. Also, data-intensive systems face recurring issues, and sometimes they significantly overlap. Therefore, with the aforementioned approach, the architecture would be inefficient and redundant.

In this work, we present for the first time a unifying model that aims to capture recurring concepts of the various components of all data-intensive systems. This model brings several contributions to the community. For instance, it introduces a new vocabulary, more general and unbiased than the ones coming from different research communities. In particular, in this paper, we aim to capture functional, architectural, and correctness guarantees concepts of a data-intensive system. Furthermore, using our model, we aim to make more precise the comparison among data-intensive systems, also the ones coming from different domains. It may be useful to understand which are the overlaps in terms of features provided or guarantees ensured across distinct areas.

The remainder of the paper is organized as follows. In Section II, we introduce our unifying model for data-intensive systems, with definitions of each component and of some classification criteria that can be used to understand the differences and the similarities of the systems. In Section III, we present a taxonomy of data-intensive systems taken from both industry and research. The taxonomy is done by showing how each system fits into our model and using the classification criteria previously defined. We tried to include the most relevant and representative systems. In Section IV, we discuss the results that emerged from the taxonomy. In Section V, relation with respect to the related work is discussed. Lastly, in Section VI, we draw the conclusions, summarizing our work and our contributions, and pointing out possible directions for future work.

II. Model

In this section, we introduce a model to capture the key design choices behind data-intensive systems. Fig. 1 shows the components of this model.

The core of the data-intensive system is constituted by the *workers*, which are the physical processes running the system. Each worker may have a certain number of *slots*, which are the abstraction of the computational resources. Each slot consists of a single thread of execution. Basically, the slots aim to model the physical cores, but with a single-threaded execution.

A data-intensive system is a system that can process and manage large quantities of data. In this model, with *data* we mean immutable information elements. Data can be exchanged between the *client* and the system. The client is the principal actor that can interact with the system, i.e. the abstraction of the application that will make use of the system to store and process data. To interact with the system, the client must first send a *driver program*, that is a set of instructions that specifies the computation that it desires to run on the system. Then, it can trigger the computation sending an *invocation*, possibly with some *data* that will be taken as input. Notice that we consider invocations as a special type of data. For example, a driver program could be a parametric stored procedure of a database that supports them. Therefore, the client should first send it and then invoke it, sending an invocation together with the parameters needed (the input data). When the driver program is submitted, it is received by the *driver receiver* component of the system, which is responsible for receiving driver programs and forwarding them to the *program executor* component. A driver program is composed of a set of instructions, to be executed by the program executor, interleaved with one or more *jobs*. A job is a parallel computation specified by the client within the driver program that has to run on the various workers of the system. Whenever the program executor wants to deploy a job, it emits a *logical job plan*. We can abstract this plan as a graph that specifies a sequence of steps to follow in order to execute the job, and in which order they should be followed. This graph is given in input to the *job optimizer* component, which will build and optimize a physical execution plan starting from the logical one. This physical execution plan is the *physical job plan*. As an example, this flow can be found in many DBMS, where the logical and the physical job plans are the logical and physical query plans, and the job optimizer is the query optimizer.

The vertices of the physical job plan are called *tasks*. A task is a set of instructions that has to be executed by a slot. To be executed, a task has first to be *installed*, and then it can be *invoked*. Possibly, the invocation of a task may carry some additional data. Some tasks can install other tasks during their execution.

The physical job plan is finally passed to the *task scheduler*, which forwards the installations to the various workers, trying to optimize performance as much as possible, for example taking into consideration data locality or trying to balance the overall load on the workers.

Concerning data locality, the *data bus* plays a key role. Indeed, the data bus is the component through which every exchange of immutable information (i.e. invocations and data) happens. It can be seen as a very generic abstraction of a communication channel. The data bus can also be *persistent* if it can store data during the communication. If that is the case, there may be data locality optimizations to take.

At the moment of the invocation, a task is executed, and after the execution, it can provide data in output to a client (that can be the same client that submitted the program or another one), communicate with other tasks by exchanging invocations or data through the data bus, or return some *feedback information*. The last-mentioned behaviour is usually present only in the case of *iterations* or *periodic job*. The former happens when a job that is described in the program

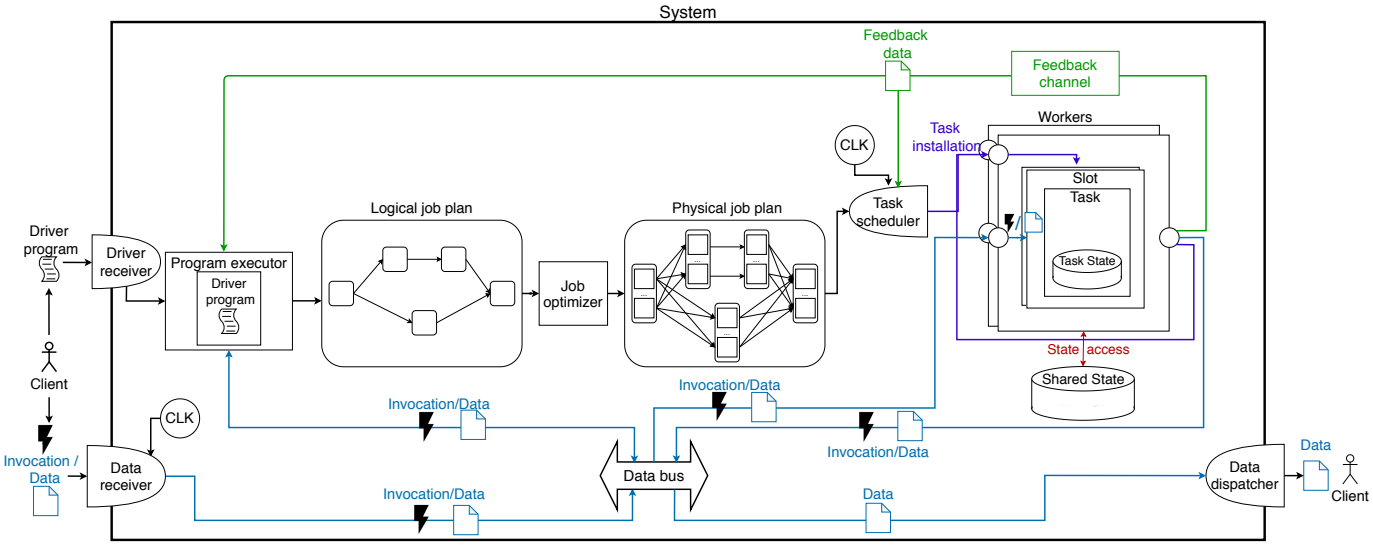

Fig. 1. Abstract model of a data-intensive system.

inside an iterative loop, and the loop condition is dependent on the job execution's result. Because of this, the feedback information must go back to the program executor, which will then evaluate if the condition is satisfied or not, and choose which job to launch next accordingly. A typical example of this is a machine learning program, which will launch many times a parallel computation and, at the end of each computation, will evaluate some objective function before deciding what to do for the next iteration of the loop. The latter instead is a job that is specified to be periodically executed, with no additional conditions. Therefore, the feedback information must return to the scheduler to acknowledge the execution of the job. The scheduler then can send again the task installation for that job, in case it wants to change the task deployment, based on the *clock* information. In the periodic job scenario, the clock connected to the data receiver is in charge of sending periodically the invocations for that job.

A system may also maintain a *shared state*, which is a set of mutable information elements, which represents the evolving state of the system during the executions of the various programs. These mutable information elements are called *state elements*. This abstraction is fundamental to model those systems which have mutable information to manage. For instance, in a DBMS, the set of stored information elements is shared and mutable. Therefore, it is a shared state.

Operators can be declared as *stateful* within the driver program, and the tasks corresponding to a stateful operator will maintain a state information in their *task state*. This is the norm in the stream processing domain, where there are operators that can maintain *windows* across invocations.

The client can decide which part of the program specify as a *transaction*. We model this possibility with two primitives that a client can insert in a program, that are "*begin transaction*" and "*end transaction*". The part of the program included between these two primitives will be the transaction. A transaction is a part of the program for which the client desires *transactional guarantees* [18] to hold. In the next section, we will provide further detail on those guarantees.

The arrival of an invocation in input of a transaction causes the transaction to begin the execution. We refer to the execution that starts after a transaction invocation as a *transaction execution* (TE) [22]. Transaction executions can either succeed (*commit*) or fail (*abort*). An abort can be one of two types:

1) *System-induced aborts*, when the system decides to abort the TE, for instance because of a node failure
2) *Logic-induced aborts*, when the transaction logic forces an abort (calling the primitive "*abort transaction*") in some particular cases, for instance if the balance of a bank account would be below zero

The fundamental difference between the two kinds of abort is that the first is *non-deterministic*, meaning that if two nodes execute the same transaction starting from the same state, they can end up in a different final state (because one of the two TE may receive a system-induced abort, while the other may not), while the second is *deterministic*, i.e., in absence of system-induced aborts, if two nodes execute the same transaction starting from the same state, they can only end up in the same final state, because either both committed or both aborted [30], [17]. Hence, we will use system-induced abort (respectively, logic-induced abort) and non-deterministic abort (respectively, deterministic abort) as synonyms.

A transaction can be defined as *deterministic* if, given multiple workers starting from the same state, the result of the transaction execution is the same among all of them [29]. This means also that non-deterministic events (such as hardware failures) can not cause a transaction abort anymore, because, otherwise, the deterministic property defined before does not hold. Instead, the failed worker will wait for its recovery and replay the input starting from its failure (how to do it

depends on the particular implementation), and, because of determinism, the state will be the same as if there would have been no failures. In this paper, with deterministic, we also mean that the transaction is *statically analyzable*. This means that the elements that the transaction wants to read and write are known before its execution.

Furthermore, transactions can be:

1) *Single-slot*, if the scheduler manage to dispatch the tasks composing the transaction's physical plan into a single slot
2) *One-shot*, if the scheduler manage to dispatch the tasks composing the transaction's physical plan such that they are all concurrent
3) *General*, if it is neither single-slot nor one-shot

These properties will be exploited for correctness purposes that we will describe in further detail later.

Classification criteria

Here we provide some additional definitions based on our model, which we will use as classification criteria to characterize the systems under analysis.

Functional concepts

Invocations. Systems can adopt three different approaches to handle invocations: *pull*, *push*, or *periodic*. With the pull approach, tasks are executed by explicitly sending an invocation to it. On the contrary, with the push approach, invocations are implicitly sent with the input data. Periodic processing, instead, means that the execution is periodically invoked by the internal clock, without any external invocation.

Shared state. A client can make different uses of the shared state. For example, it may take a decision based on the evolving value of the state of the system. This opportunity can be relevant in some use cases, but useless in others. Indeed, not all the data-intensive systems maintain a shared state. Therefore, the presence of the shared state is a classification criterion.

Task state. Similar to the shared state, not all systems provide primitives for declaring stateful operators. Therefore, such systems do not support task states.

Job scheduling. The program executor can behave in two ways:

- When it receives the program, it will wait until an invocation of that program is received. Upon program invocation, it starts executing the program and, when it finds a job during the computation, it will emit the job;
- First, it parses the program, it sends forth all the jobs found in it, and finally, it waits for an invocation that will trigger the actual program execution.

In the former scenario, we say that the system adopts *dynamic job scheduling*, whereas, in the latter, we say job scheduling is *static*.

Job optimizer. Some systems do not have a job optimizer, while others do. Hence, the presence of a job optimizer is a criterion for classification.

Iterations. Similarly, support for iterations is not provided by every system. Thus, we use this feature as a classification criterion.

State and data management

Persistent data bus. All the systems have a data bus, which is necessary for data exchange, but only some of them have a persistent bus.

Replication. Replication means that the same information elements are stored on multiple workers, possibly on different nodes. In replication, we have the concepts of *leader replica* and its *follower replicas*. A leader replica is a set of information elements stored into a worker and each of its follower replicas is an exact copy of the leader replica, stored in other workers, possibly on different nodes. Replication can be classified on how it handles leader and follower replicas. The two macro-categories of replication are *active* replication and *passive* replication. Active replication can be *single leader*, *multi-leader*, or *leaderless*. In single-leader mode, every element is present into exactly one leader replica, while it can be replicated into multiple follower replicas. The worker that stores the leader replica is called *leader* for those elements, while the others are *followers*. If a client wants to write a state element, it must send its request to the leader for that element, which will then propagate the information to the followers. For a read request instead, a client can send a request to any worker that has the element requested, even if it is a follower for that element. Multi-leader is like single-leader, but there is more than one leader for every state element. If this is the case, it means that two clients can update the same element present in two different leaders, possibly leading to inconsistency. Leaderless replication means that the client can read and write an element from every worker of the system, therefore all the replicas are leader replicas. Also with this replication strategy there can be inconsistencies.

Partitioning. Partitioning means breaking a set of information elements into multiple non-overlapping partitions, to be distributed on the various workers. In principle, with simple partitioning, every element belongs to exactly one partition. There are two basic techniques to partition: *random* or *based on the content*. For instance, a content-based partitioning policy for an online store could be to partition by the customer, thus ensuring that all the information related to a single customer is stored within the same worker.

Correctness guarantees

Atomicity. The atomicity property ensures that either all of the effects of a transaction execution are visible or none of them is visible. The most popular protocol for implementing atomicity is *Two Phase Commit* (2PC) [8]. Some systems prefer to use partial implementations of an algorithm that guarantees atomicity relying on some preconditions whenever they hold, because 2PC may decrease the throughput and increase latency. For example, if the *scheduling* achieves to create a single-slot transaction, the system does not need a complex commitment algorithm like 2PC, because the transaction can not execute in parallel, due to the slot's characteristics. Indeed, in this case coordination is not needed at all. Significant

simplifications arise also when the transaction is deterministic. If such is the case, the transaction will not abort for non-determinstic reasons. This can result in a simplification of a commitment algorithm because the algorithm does not need to check if any of the nodes aborted non-deterministically. Relying solely on logic-induced aborts has also disadvantages though. The main one is the fact that a worker can not use aborts to cope with overload on its resources, and this could cause the system to struggle in some cases of heavy workloads [25].

Consistency constraints. Clients can specify a predicate P on the values of state elements that must be true after the execution of a transaction. If it is not true, the transaction must be aborted and none of its effects should be visible. The implementation of these constraints is strictly related with the implementation of the atomicity guarantee. Indeed, systems that offer those constraints usually implement them in this way: first, execute the transaction as it is, then, before the commitment protocol, check if the constraints are respected. If not, abort the transaction. Otherwise, it can commit and continue the commitment protocol.

Isolation. Isolation (also called *concurrency control*) is a property that can have different *levels*. Many studies formally defined the various levels of isolation in literature [6], [1], [5]. In our analysis, the most relevant is *serializability*, which ensures that the effects of a set of concurrent transaction executions are the same as if all the transactions were executed in some sequential order. Here are presented some of the usually recurring implementation of isolation levels:

- *Scheduling*
  If the scheduler emits single-slot plans for concurrent transactions, and if they are forwarded to the *same* slot, then the result is serializable by definition.
- *Two-Phase Locking*
  Two-phase locking (2PL) is a lock-based approach, where locks can be *shared*, if the requesting transaction wants to read the element, or *exclusive*, if the requesting transaction wants also to write the element [7].
- *Timestamp ordering*
  The timestamp ordering (TS) technique attaches timestamps to transactions and based on them decides in which order execute the transactions. In particular, transactions are required to execute *conflicting operations* (read and write, or write and write, on the same element) in timestamp order. If the order is violated, the older transaction is aborted and re-executed with a new (bigger) timestamp.
- *Deterministic ordering*
  The basic idea of this concurrency control method is to first decide in which order execute a set of transaction invocation (agreement on the input) and then execute the transactions with a schedule that is serial equivalent to the agreed order. This algorithm, as well as its advantages and disadvantages, are described in further detail in [25].

Durability. The guarantee of durability ensures that the effects of a committed transaction execution will not be lost, also in case of failures. Essentially, the durability transactional guarantee means that there must be some sort of *fault tolerance* mechanism that is triggered by the transaction commit.

Fault tolerance. This guarantee ensures that the information contained in the system is stored into a *stable* storage and therefore will not be lost in the case of failures. To provide fault tolerance, the typical approaches involve logging. In particular, two different techniques of logging, widely used in the data-intensive tools' domain, are *Write-ahead log* (WAL) [23] and *Command log* (CL) [21]. A recovery strategy based only on logging is not feasible in practice, because eventually the log will explode. Therefore, usually it is adopted together with the *snapshot* technique [12]. A snapshot consists of saving a consistent state of the system on a stable storage, from which it is easier to start the command log recovery. Another way to provide fault tolerance is to use *replication*, that was already described before.

III. Systems

A. VoltDB

Functional concepts

VoltDB [28] is a database management system. It is the commercial product derived by the H-Store research prototype [27], [19]. VoltDB is relational, so the information that it stores is in the form of tables. The set of stored tables represents its shared state. Clients must use *stored procedures* to interact with the system. A stored procedure corresponds to the driver program concept of our model, and VoltDB requires it to be a mix of Java code and SQL queries. The client needs to send the compiled version of the procedure (in Jar form) to the driver receiver of the system, that in VoltDB is any node of the system. Once the system has received the stored procedure, it can parse it, create the physical plan through its optimizer, and deploy the tasks on the workers (static scheduling). After that, the client will be able to call it, following the pull approach. Iterations are supported by simply writing SQL queries inside loops. VoltDB does not provide support for task states.

State and data management

VoltDB supports both partitioning and replication. Replication can be active, single-leader or leaderless. The communication among the workers is done through the network, therefore the data bus is not persistent.

Correctness guarantees

VoltDB classifies transactions in three kinds:

1) *Single-partition* (i.e. single-slot, since VoltDB workers always have one slot)
2) *One-shot*
3) *General*

Based on this classification, it applies different optimizations and implementations to achieve guarantees. For instance, atomicity for a single-partition transaction is provided by the fact that every worker has a single slot, therefore the execution of that transaction must be sequential and until completion. For a one-shot *read-only* transaction, it does not need any commitment protocol. If the transaction can not be classified

	Invocations	Shared state	Task state	Job optimizer	Iterations
VoltDB	Pull/push	Yes	No	Yes	Yes
Calvin	Pull	Yes	No	Yes	No
StreamDB	Push	Yes	No	Yes	No
Spark	Pull	No	No	Yes	Yes
Flink	Push	No	Yes	Yes	Yes
Spark Streaming	Periodic	No	Yes	Yes	No
TSpoon	Push	No	Yes	Yes	Yes
S-Store	Pull/push	Yes	Yes	Yes	Yes

TABLE I
FUNCTIONAL CONCEPTS.

	Persistent data bus	Replication	Partitioning
VoltDB	No	Active, single-leader Leaderless	Content
Calvin	No	Active, leaderless	Content
StreamDB	No	Active, single-leader	Content
Spark	Yes	Active, leaderless	Content
Flink	No	No	Content
Spark Streaming	Yes	Active, leaderless	Content
TSpoon	No	No	Content
S-Store	No	Active, single-leader Leaderless	Content

TABLE II
STATE AND DATA MANAGEMENT.

	Atomicity		Consistency		Isolation		Durability		Fault tolerance	
	Impl.	**Precond.**	**Impl.**	**Precond.**	**Impl.**	**Precond.**	**Impl.**	**Precond.**	**Impl.**	**Precond.**
VoltDB	Scheduling	Single-slot/One-shot RO transaction	Check before commit	/	Deterministic	Det. transactions	FT before commit	/	CL+Snapshot	/
	2PC	/							Replication	/
Calvin	Deterministic	No system-induced aborts	Check before commit	/	Deterministic	Det. transactions	FT before commit	/	CL+Snapshot	/
									Replication	/
StreamDB	Scheduling	Single-slot transaction	No	/	TS	/	No	/	No	/
Spark	No	/	No	/	No	/	No	/	Replication	/
Flink	Scheduling	Single-slot transaction	No	/	Scheduling	Single-slot transactions	No	/	Snapshot	Client sends missing data
Spark Streaming	Scheduling	Single-slot transaction	No	/	Scheduling	Single-slot transactions	No	/	Snapshot	Client sends missing data
TSpoon	2PC	/	Check before commit	Check only on one task state	2PL TS	/ /	FT before commit	/	WAL	/
S-Store	Scheduling	Single-slot/One-shot RO transaction	Check before commit	/	Deterministic	Det. transactions	FT before commit	/	CL+Snapshot	/
	2PC	/							Replication	/

TABLE III
CORRECTNESS GUARANTEES.

into one of these two kinds, VoltDB uses standard 2PC. Consistency constraints can easily be implemented inserting a check inside the stored procedure. Regarding isolation, VoltDB uses the deterministic ordering approach to achieve serializability for multi-partition transactions. The order of execution of the transactions is decided by a worker that acts as a coordinator. Fault tolerance is achieved via command logging and periodic snapshotting. The durability of transactions is guaranteed using the fault tolerance mechanism defined before at the end of the commitment protocol.

B. Calvin

Functional concepts

Calvin is a distributed DBMS research prototype, that was originally designed to be a pre-processing layer for transactional guarantees and replication management [30]. Calvin stores a set of key-value pairs as its shared state. The driver program submitted by the client is a transaction. Jobs (i.e. queries) will be dynamically scheduled during the execution of a transaction. As in many database systems, the adopted approach is the pull one. Jobs are optimized via a query optimizer. A transaction can be submitted to any node. All transactions in Calvin must be deterministic. This is a fundamental assumption that Calvin will use to ensure multiple guarantees and optimizations. For this reason, iterations, which would make a transaction no more statically analyzable, are not supported. Task states are not supported.

State and data management

Calvin adopts replication and partitioning. Partitioning is based on the key of the elements. Replication is adopted at system-level, i.e. all the partitions are replicated, creating what Calvin calls different *regions*. A region is a set of nodes that contains the whole shared state in their partitions. The replication is leaderless, meaning that a client can contact every replica for reads and writes. The data bus is the TCP network. Hence, it is not persistent.

Correctness guarantees

Regarding atomicity, since transactions in Calvin do not have non-deterministic aborts, Calvin uses the deterministic protocol to ensure atomicity. Similarly, serializablity is provided with the deterministic ordering. For fault tolerance, Calvin uses CL and snapshotting. Durability is provided using fault tolerance mechanisms after the commit.

C. StreamDB

Functional concepts

StreamDB is a research prototype of a DBMS [13] that borrows design concepts from the stream processing area. Indeed, in StreamDB the processing follows the push model. Jobs are declared as dataflow graphs and are called *transaction graphs* because all jobs have transactional guarantees. Once the t-graphs are optimized and deployed, they are ready to process input data (static job scheduling). When a datum arrives at the system, it may trigger a transaction graph, that will interact with a shared relational state. No stateful tasks are provided. Iterations inside the t-graphs are not supported.

State and data management

Partitioning is supported. Replication is provided using strategy active, single-leader. The data bus is not persistent.

Correctness guarantees

Atomicity is provided only for single-slot transactions. Consistency constraints and durability are not provided. Fault tolerance is provided through replication. For isolation, it adopts the TS protocol.

D. Spark

Functional concepts

Apache Spark is a framework for distributed batch data processing [31]. It is inspired from the *MapReduce* framework developed by Google [16] and its open-source derivation, *Hadoop*. Spark offers APIs in several programming languages. The client can use these APIs to write the driver program. Iterations can be specified inside the program. Once the program is written, the client has to send it (in a jar format) to the master node, and the program will be executed (pull processing). Spark will dynamically schedule and optimize jobs during the execution. No support for state handling (neither shared nor task) is provided.

State and data management

Before starting the processing, the client must store the data into a distributed repository, usually a distributed file system. In Hadoop, the *Hadoop Distributed File System* (HDFS) [26] is used. The most popular option is HDFS also for Spark. Spark uses this distributed data repository as its distributed data bus. Hence, the data bus is persistent. HDFS supports partitioning, based on the content of the files, and replication, which is leaderless, since a client can add files in every node.

Correctness guarantees

Spark does not provide any ACID guarantees, due to the fact that it does not maintain any state. Fault tolerance is provided through HDFS replication.

E. Flink

Functional concepts

Apache Flink is a distributed stream processing platform, optimized for low latency [11]. It supports also batch processing as a special case of stream processing, reading one element of the batch at a time, as if it was a stream of data. In order to interact with the system, Flink provides different client APIs in several programming languages. A Flink program is mainly composed of functional primitives, that will be compiled by the client into a dataflow graph, that represents the desired stream processing pipeline, called *dataflow graph*. This pipeline is the logical job plan, that the client can send to the *job manager* process, once the program is compiled (in the jar format). Iterations are supported through the definition of special *iteration operators*. Within the program, the client can define also stateful operators, therefore task states are supported. On the contrary, there is no shared state in Flink. Once the dataflow graph is received, the job manager creates the physical job plan through the *scheduler* (which acts also as the optimizer). The scheduler also forwards the tasks to the other workers, that Flink calls *task managers*. The job scheduling therefore is static. Like many stream processing systems, Flink adopts a push approach, i.e., invocations are implicitly sent with the input data. Therefore a client, in order to start a computation, needs to send data to the source of the previously deployed pipeline.

State and data management

During execution, Flink tasks communicate among each other via TCP connections, that we represented as a non-persistent data bus. The *parallelism* setting is selected by the client, that can modify the "*parallelism.default*" property in the Flink configuration, or can set explicitly the parallelism for each operator in the program. This setting will enable partitioning, based on the key of the state element, over task states. Replication is not provided.

Correctness guarantees

Flink does not give any explicit transactional primitive, but it provides some guarantees as a consequence of its processing model. Indeed, due to the single-threaded execution of each task, we can say that, for such single-slot transactions, Flink provides atomicity and isolation, with regard to the task state accesses. Consistency constraints are not supported. For fault tolerance of the task states, Flink uses a periodic distributed snapshot. This means that, in the presence of a failure, Flink requires the client to resend all the previous data starting from the snapshot.

F. TSpoon

Functional concepts

TSpoon is a research prototype of a stream processing system, described in [2]. It is built on top of Apache Flink, from which it inherits most of its architectural and functional features. The main issue that led to TSpoon development was the missing of transactional guarantees in the field of stream processing. The main concept is that of the *transactional subgraph* (or *t-graph*), that is a subgraph of the dataflow graph (logical plan) typical of Flink pipelines. This subgraph can be defined from the client, with two constraints: t-graphs must have only one input edge and they can not share any operator. Once a t-graph is defined, obeying these two constraints, ACID guarantees will be provided for the executions of that particular subgraph. Other functional concepts are the same as in Flink.

State and data management

State and data management features are the same as in Flink.

Correctness guarantees

Guarantees are implemented using two new operators introduced by TSpoon: *open* and *close*. The open operator is created automatically at the beginning of a subgraph, and, for every input element, it creates a data structure wrapping the element and adding some metadata useful to ensure transactional guarantees. Additionally, the open operator stores pending transactions. The close operator will check the outcome of the transaction looking into the metadata and implement a 2PC protocol (with the role of the coordinator) with the tasks (that will be the participants). During the execution, tasks will check consistency constraints on their task state and attach metadata accordingly. Notice that consistency constraints are implemented and will abort the whole transaction execution if not satisfied, but they can be defined only within the context of a single operator state. Furthermore, during task executions, TSpoon offers isolation, using an algorithm of choice between 2PL and TS. For fault tolerance, the open operator maintains a write-ahead log, which is used also for durability.

G. Spark streaming

Functional concepts

Spark Streaming is a stream processing engine based on Apache Spark. The main difference with other stream processing engines, like Flink, is that Spark Streaming can dynamically schedule the tasks of a job during the execution, instead of deploying the tasks and fix them. That enables dynamic scalability and balancing, at the cost of an increased delay during the processing. The core programmatic abstraction is that of *discretized streams* (or *DStreams*), meaning that streaming input elements are accumulated periodically to create a batch, and, once the batch is collected, the classical Spark jobs are applied. From a functional point of view, the client always has to declare the program it wants to run, but in the streaming setting the program is not run as soon as it is received. Instead, a periodic approach is adopted. As usual in the stream processing world, Spark Streaming provides stateful operators.

State and data management

Data management features are the same as in Spark. The task state is not replicated.

Correctness guarantees

Similarly to Flink, some guarantees are provided as a consequence of its processing model. In particular, like it was for Flink, for single-slot transactions, atomicity and isolation are provided. For the fault tolerance of the task state, it adopts a distributed snapshot approach, and, in case of failures, it will ask the source to resend the lost data.

H. S-Store

Functional concepts

S-Store is a research prototype [22], developed with the purpose of integrating OLTP and stream processing into one coherent model. Its approach resembles the one of TSpoon, but in the opposite way. Indeed, as TSpoon was built on top of a stream processing system (Apache Flink) trying to provide transactional guarantees, S-Store starts from an ACID DBMS (H-Store) and tries to provide functionalities and guarantees proper of the streaming domain. The client using S-Store can interact with it by defining *stored procedures*, and later it will be able to invoke it. All the stored procedures are considered transactions, but S-Store distinguishes two types of transactions: *OLTP* and *streaming*. An OLTP transaction is a classical transaction, already provided in H-Store, that can be invoked using a pull approach. A streaming transaction instead is a transaction invoked by the arrival of input data (push approach). Streaming transactions in particular can be defined as a *dataflow graph* of transactions, with edges (called *streams*) that indicate the precedences among them. Each of these sub-transactions in the graph can maintain a *window*, a private state belonging to the transaction, that corresponds to our task state. S-Store maintains also a shared state, that is a set of relational tables, similarly to H-Store. Another difference between an OLTP transaction and a streaming one is that the former can access only the shared state, while the latter can possibly access windows (but only the ones of its tasks).

State and data management

The data bus is not persistent, because the communication is done through the network. Since also VoltDB is derived from H-Store, S-Store and VoltDB have very similar architectural features. Indeed, S-Store supports replication (active, single-leader or leaderless) and partitioning, as also VoltDB does.

Correctness guarantees

We will not describe in detail the isolation protocols implemented by S-Store because it is essentially the same of VoltDB. The same holds for atomicity and consistency constraints implementation. Regarding durability and fault tolerance instead, S-Store provides two different mechanisms:

1) Strong recovery
   Same as the VoltDB mechanism, it uses a command log and a periodic snapshot.
2) Weak recovery
   It is similar to Flink recovery, but it does not need the client to resend the data and the invocations, because all the invocations are logged in the command log. Looking in the log, the system will replay all the invocations, that will lead to a valid state, but potentially not the same as before the crash.

In both cases, CL and snapshot are used. However, in the weak recovery mode, only the input data and invocations are logged, and not the intermediate data exchange among tasks. Furthermore, also replication is used.

IV. Discussion

In this section, we will discuss the results of our analysis on the presented systems, summarized in the Tables I, II, III. The goal of this discussion is to show how systems, even if designed for different workloads, can share functionalities, deployment strategies, or guarantees implementations. It is interesting to notice how our unifying model can describe all the systems examined. We also notice that a lot of issues are recurring in all data-intensive settings, and the process to solve those issues is usually similar across workloads.

A. Functional concepts

First of all, we notice that in the database area, as well as in the batch processing one, the pull processing approach is the relevant one. The opposite happens in the stream processing world. The push processing feature is present in stream processing systems probably because, usually, they have the requirement of low latency processing. Hence, data must be processed as soon as it arrives, without waiting for an additional invocation, that could be harmful to the latency. However, there are some exceptions. StreamDB is a database, but it supports only the push approach since it is inspired by the stream processing architectures. VoltDB supports push processing thanks to the materialized views support. Indeed, maintaining a materialized view is very similar to a stream processing problem: the view is defined, and as soon as an event that modifies the view happens, it must be modified to keep it consistent. Spark Streaming adopts the periodic approach because it wants to exploit the Spark (batch) engine also in the case of stream processing. S-Store, a hybrid system, is built on top of H-Store, a relational DBMS. Therefore, it supports natively pull processing. Furthermore, S-Store implements a mechanism to support streaming transactions using H-Store's triggers, therefore implementing push processing. In this way, it can handle a mixed workload.

The most relevant difference between DB and SP tools is how they handle state information. Both have state information, but DB systems maintain it into a shared state, while SP systems use task states. Spark does not offer any state management, and this represents probably its main weakness. S-Store, instead, can provide support for both shared state and task state.

B. State and data management

We noticed that all the analyzed systems adopt replication and partitioning. Different techniques are used indifferently from system to system. Spark is the only system that has a persistent data bus, due to its usage of a distributed file system.

C. Correctness guarantees

Some systems give very little assistance for transactions. For instance, in the streaming domain, only TSpoon provides explicit primitives for transactional processing. TSpoon was created as a layer on top of Flink to add transactional guarantees to it. In other SP systems, some transactional guarantees are implicitly provided only as a consequence of their processing model. For example, in Flink, a task is assigned to only one slot. Hence, the processing of its state must be single-threaded. This implies that, in the case of a single slot transaction, the processing of the task state is atomic.

Concerning the database area, we notice that a similar strategy for single slot transaction is used by every system analyzed. Besides, some systems provide additional methods to ensure atomicity for every type of transaction.

Regarding isolation guarantees, we can notice how implementations in most of the stream processing systems are a consequence of the processing engine behavior. As described before, almost all of them offer serializability only for the transactions that involve the same single slot. This guarantee is the consequence of the fact that, in the case of multiple single slot transactions to the same slot, they must execute in some serial order; therefore, the isolation level is serializability. The only exception is TSpoon, which implemented on top of Flink a concurrency control protocol. In the database domain, there is an opposite trend, as the majority of them implements concurrency control algorithm to explicitly provide serializability.

Consistency constraints are implemented in every system that has the explicit "abort transaction" primitive. In this case, a simple check of the desired consistency constraints is made within the transaction before committing. If the constraints are not satisfied, a client can use the "abort" primitive.

Durability, when present, is always given relying on the fault tolerance mechanism.

Fault tolerance is provided by virtually every system considered in the analysis. The only exception is StreamDB, which is an academic prototype and may implement it in the future. Every system uses one of distributed snapshotting (with optional command logging), write-ahead logging, or replication, regardless of the category. Arguably, we can say that a WAL is equivalent to a snapshot made after every task execution and therefore reduce every system to two techniques. Some SP systems may give lower importance to fault-tolerance guarantees. For example, Spark Streaming and Flink, by default, use only a distributed snapshot. This means that fault tolerance is only provided if the client remembers the data sent after the snapshot. In other words, they expect the client to behave as a command log.

D. General remarks

First of all, we notice that a lot of workloads may require stateful computation, and the lack of state information in Spark can be limiting.

Also, StreamDB makes clear how the distinction between categories sometimes is blurred. It has assumptions and architecture similar to the ones of stream processing systems, but it also has a shared state, a property typical of database systems. Further evidence of this is that Spark and Spark Streaming exploit the same architecture to handle both batch and stream processing. We also notice how this approach could potentially be adopted by all the databases taken into consideration that adopt a pull approach. None of them already implemented this, but it could be future development in the database research area.

As previously said, the most relevant difference between stream processing systems and database systems is state management. In particular, DB systems maintain a shared state, while SP ones use task states. This has an impact on which functionalities a system provides to the client. For instance, if we consider a relational database, a client can potentially write complex queries that involve multiple tables, with filters and joins. This is not possible in the case of streaming systems, because task states are usually not readable from the outside.

Flink provides a queryable state, which is a task state that can be read from a client, but it does not offer any guarantees on the reads of such a state.

S-Store is the only to include both a shared state and task states. In this way, it can handle mixed workloads. Hybrids like S-Store could be a first step towards a new breed of architectures widely used for data-intensive applications, as opposed to complex architectures that try to connect multiple systems. However, S-Store came from a pre-existing system (H-Store). Instead, what we envision for the future is a new holistic approach for designing a new data-intensive system from scratch.

V. Related work

In this section, we will show some attempts to create a model related to data-intensive systems. Usually, those models focus on one specific research area (for instance, stream processing) or aspect of a system (e.g., transactions). No one adopts a unifying approach as our model.

In [9], the authors focused on stream processing tools. In particular, they noticed that no existing system decoupled the concepts of processing engine and storage engine. This tight coupling reduces flexibility and opportunities for optimizations. Hence, they decided to separate the storage manager from the stream engine, similarly to what is done with buffer managers in the DBMSs. To do so, they defined a formal model of a storage manager, with a set of parameters divided in architectural, functional, and performance-related. In particular, the paper focused on the performance-related ones and on how they could be tuned according to different workloads. Starting from this model, they described how they implemented *Storage Manager for Streams* (SMS) and how it could be used along with a stream processing engine.

The same authors later wrote another paper [10], where they focused on how to integrate transactional processing and static data sources with stream processing. First of all, they assumed to have a hybrid system, which had features of both DBMSs and stream processing systems. Upon this, they created a model to define guarantees in such a scenario. This model was called the *Unified Transactional Model* (UTM). In this model, first, they declared how data incoming to a streaming pipeline is equivalent to an invocation in the DB area, similarly to what we did in our functional model. Then, they defined transactional guarantees in the case of a hybrid system. In particular, they focused on the isolation guarantee. After the definition of UTM, they described how they implemented a transaction manager capable of implementing the transactional model on top of a hybrid system.

A similar model is defined in the S-Store paper [22], which we already discussed in the previous section. Before describing the actual implementation of S-Store, the authors defined a transaction model for hybrid scenarios such as the one of S-Store, i.e. when there are both shared state and task states. First, they divided the possible transactions into two categories: OLTP and streaming. Then, they defined serializability in case of such a hybrid workload with both OLTP and streaming transaction executions. Furthermore, they also defined delivery guarantees. This model was focused only on the guarantees and not on architectural or functional concepts, as ours.

A model that tried to include functional aspects was the *Dataflow model* by Google [3]. This was a conceptual model that wanted to unify functional characteristics proper of the stream processing domain. This necessity came from the fact that, in that domain, no system provided the opportunity for optimizations along all the dimensions of correctness, latency, and cost. This is a consequence of the high heterogeneity in the research area: fault-tolerance guarantees are not clearly stated in many stream processing systems, correctness guarantees usually are not ensured, and few systems have precise temporal primitives. In summary, the lack of a unifying model led to engines that dictated the semantics. Dataflow aimed at the opposite and defined a modeling framework composed of a windowing, a triggering, and an incremental processing model.

Another related research is [15]. In this paper, the authors claim how complex event processing and stream processing are two sides of the same coin and, therefore, could be unified under the same modeling framework. They defined a long list of models, including functional, processing, deployment, data, language, and time characteristics. Then, they presented a comprehensive classification of several systems present in the literature. This research work adopts an approach that is very similar to ours, but it is focused on a narrower research area, such as stream processing.

VI. Conclusion

The goal of this paper was to examine data-intensive systems design and to understand which are the functional, architectural, and correctness concepts that those systems share. This analysis was done through the definition of a formal unifying model, which was able to illustrate the similarities and differences of data-intensive systems that arose in recent years. To validate the model, we analyzed 8 different systems coming from various domains, using the defined model for comparison. We noticed that the model was able to unambiguously capture recurring concepts and issues of data-intensive systems.

Possible future work includes expanding the taxonomy with other relevant systems, both from industry and research. Also, the model can be extended (for example, taking into account the physical storage of state and data). Furthermore, we envision that in the future our model can be used as a starting point for a new approach for designing data-intensive systems, where the design is driven by a declarative specification of the desired semantics. Hypothetically, it can be included in a development platform that will help the creation and deployment of a data-intensive system. For instance, a developer may use the unambiguous definitions of the correctness guarantees given by our model to ask the platform to deploy a data-intensive system that will ensure the correctness level desired.

REFERENCES

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In David B. Lomet and Gerhard Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78. IEEE Computer Society, 2000.

[2] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. Tspoon: Transactions on a stream processor. *J. Parallel Distributed Comput.*, 140:65–79, 2020.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[4] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[5] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, 2013.

[6] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995.

[7] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Flexible and scalable storage management for data-intensive stream processing. In Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 934–945. ACM, 2009.

[10] Irina Botan, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Transactional stream processing. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 204–215. ACM, 2012.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[12] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[13] Huankai Chen and Matteo Migliavacca. Streamdb: A unified data management system for service-based cloud application. In *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 169–176. IEEE, 2018.

[14] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[15] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[17] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, 2017.

[18] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[19] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

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

[21] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory OLTP recovery. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 604–615. IEEE Computer Society, 2014.

[22] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. S-store: Streaming meets transaction processing. *Proc. VLDB Endow.*, 8(13):2134–2145, 2015.

[23] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[24] Andrew Pavlo and Matthew Aslett. What's really new with newsql? *SIGMOD Rec.*, 45(2):45–55, 2016.

[25] Kun Ren, Alexander Thomson, and Daniel J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 7(10):821–832, 2014.

[26] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[27] 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 Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160. ACM, 2007.

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

[29] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1):70–80, 2010.

[30] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM, 2012.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.