Journal of Parallel and Distributed Computing 140 (2020) 65–79

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

TSpoon: Transactions on a stream processor

Lorenzo Affetti, Alessandro Margara [*], Gianpaolo Cugola

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Italy

ARTICLE INFO

Article history:
Received 19 July 2018
Received in revised form 16 September 2019
Accepted 10 March 2020
Available online 26 March 2020

Keywords:
Distributed Stream processing
Data databases
Transactions
Queryable state
TSpoon

ABSTRACT

Stream processing systems are increasingly becoming a core element in the data processing stack of many large companies, where they complement data management frameworks to build comprehensive solutions for processing, storage, and query. The adoption of separate tools leads to complex architectures that leave developers with the difficult task of writing application-specific code that ensures integration correctness. This hinders design, implementation, maintenance, and evolution. We address this problem with a new model that seamlessly integrates data management capabilities within a distributed stream processor. The model makes the state of stream processing operators externally visible and queryable, providing transactional guarantees for state accesses and updates. It enables developers to configure transactions obtaining strong guarantees when needed and relaxing them for higher performance when possible. We introduce the new model and formalize the transactional guarantees it offers. We discuss the implementation of the model into the TSpoon tool and experiment different algorithms to enforce transactional behavior. We evaluate the performance of TSpoon with real world case studies and synthetic workloads, compare it with state-of-the-art tools for distributed in-memory stream processing and data management, and analyze in detail the cost to ensure various transactional semantics.

1. Introduction

Increasingly many applications require a near real-time processing of large volumes of data as they become available. Examples include distributed systems monitoring, social media notification services, and fraud detection systems. In general, the ability to analyze *streams of data* is vital in today's information systems. Modern *Stream Processors* (SPs) address this need for high throughput and low latency by leveraging clusters of commodity machines to distribute the processing load, and offer fault-tolerance mechanisms to quickly recover from machine failures. SPs adopt a dataflow model that represents data processing tasks as directed graphs, where edges are streams of data and nodes are operators that transform input streams into output streams. The model imposes that each operator only accesses its local state, thus avoiding state access conflicts and enabling a high degree of parallelism.

The benefits of SPs in terms of performance, scalability, and fault-tolerance led to their adoption in the data processing stack of many companies. In these settings, SPs often complement more traditional data management tools such as DBMSs to build comprehensive architectures for data processing, storage, and query [27]. For instance, monitoring systems often clean, transform, and aggregate raw data as it becomes available, before storing it and making it accessible to users; shopping websites validate incoming user requests before confirming and registering them; social media services analyze users' actions and store them on persistent datastores. All these scenarios exemplify the need for software architectures that analyze and transform large volumes of input data while storing some intermediate results of these computations and making them available for other services.

Unfortunately, these architectures suffer from the complexity of managing separate subsystems and force developers to coherently integrate them, a difficult task that requires a deep understanding of the semantics of individual systems and a careful design and implementation of the integration mechanisms, with the risk of introducing functional errors and performance problems. As a consequence, these architectures may hinder the design, implementation, and evolution of the overall system. Moreover, they might waste resources due to unnecessary data redundancy across different subsystems.

We address this problem by proposing a new model that extends that of today's SPs with *data management* capabilities – storage and query of the results of the computations – thus eliminating the need for external systems. The model allows developers to expose the local state of SP operators and make it queryable. It introduces transactional guarantees for state updates and queries, and enables developers to selectively configure

[*] Corresponding author.
E-mail addresses: lorenzo.affetti@polimi.it (L. Affetti), alessandro.margara@polimi.it (A. Margara), gianpaolo.cugola@polimi.it (G. Cugola).

https://doi.org/10.1016/j.jpdc.2020.03.003
0743-7315/© 2020 Elsevier Inc. All rights reserved.

the level of isolation and the durability of transactions to obtain stronger guarantees when needed and relaxing them for higher performance when possible.

We implement the model in the TSpoon (Transactions ON the Stream ProcessOr) system based on the Flink SP [7,14] and we propose different algorithms to ensure transactional guarantees at various levels of isolation and durability. We evaluate the performance of TSpoon using case studies and synthetic workloads, comparing it with state-of-the-art tools for distributed in-memory stream processing and data management. We confront the proposed algorithms for transactional updates and queries, and we analyze their benefits and costs. TSpoon does not introduce any overhead to pure stream processing in the absence of transactional requirements. While not being a full-fledged database system, TSpoon also performs better than a state-of-the-art distributed in-memory database in several data management tasks.

In summary, the paper makes the following contributions: (1) it introduces a new model that seamlessly integrates queryable state and transactions within a SP; (2) it formalizes the semantics of transactions on the state of a SP and proposes configurable levels of isolation and durability; (3) it presents the TSpoon system and explores different strategies to achieve the proposed guarantees; (4) it offers a detailed evaluation of the performance of TSpoon, focusing on the algorithms for transactional semantics and on the comparison with state-of-the-art tools for distributed stream processing and data management.

This paper extends our previous work on integrating stream processing and state management [5]. This research sheds light on some intrinsic limitations in the model of today's SPs, with the goal of pushing the boundaries of such model and provide better data management capabilities without sacrificing performance, scalability, and fault-tolerance. We are convinced that this work can open new research perspectives and help building the foundations of future SPs.

The paper is organized as follows: Section 2 overviews distributed SPs and motivates our work. Section 3 introduces our model. Section 4 presents the design and implementation of TSpoon and Section 5 evaluates its performance. Finally, Section 6 reviews related work and Section 7 draws conclusive remarks.

2. Background and motivations

This section motivates our work by overviewing the processing model of SPs and discussing its limitations in terms of state management capabilities.

2.1. The SP model

State-of-the-art SPs such as Apache Storm [32], Google DataFlow [6], and Apache Flink [14] enable high-throughput and low-latency distributed processing of data streams by adopting a dataflow model that organizes the computation into a directed graph of operators. The edges of the graph are the streams of data – unbounded sequences of data elements – that flow from operator to operator. Operators consume data from their input streams and append data to their output streams. For instance, a `map` operator transforms each input element into an output element according to the behavior specified by a user-defined function. Similarly, a `filter` operator propagates or discards input elements according to a user-defined predicate. Depending on the specific system, the graph can be explicitly defined by the developer or generated from a higher-level specification, for instance using a declarative API. Operators can be either stateless or stateful. For stateless operators, the processing of each input element only depends on the content of that element, while

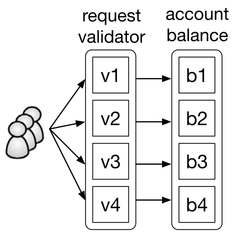


Fig. 1. A SP implementation of a bank management application.

stateful operators accumulate some local state, which can be accessed while processing input elements. For instance, a stateful `count` operator receives a stream of words and continuously stores and outputs the number of occurrences of each word received so far.

This model offers *task parallelism* by enabling different operators to run simultaneously on the same or on different machines. It also offers *data parallelism* by creating parallel instances of each operator, with each instance working on an independent partition of the input streams. Developers need only to specify the behavior of operators and how the input streams can be partitioned among parallel instances to guarantee a correct behavior. For example, in the case of the `count` operator above, the stream needs to be partitioned by word to ensure that all the occurrences of a given word are processed by the same operator instance, which retains the current count for that word. The SP runtime takes care of operator deployment, data communication, and fault-tolerance, which are arguably among the most complex and critical aspects in distributed applications. To make this possible, the dataflow model enforces some design rules that trade generality for performance and scalability. Most notably, the model requires that different operator instances do not share any state. These rules limit the data management capabilities of SPs, as we discuss in the remainder of this section.

2.2. Limitations in the SP model

Our work moves from two observations: (i) companies often need to integrate *data analytics* tasks – complex computations over the input data – with *data management* tasks — storage and retrieval of the results of such computations; (ii) the SP model is suitable for data analytics, but presents some severe limitations in data management. As a consequence, companies couple SPs with data management systems, building complex architectures that hinder the design, implementation, and maintenance of the overall solution. This work aims to offer a unifying solution that overcomes some of the limitations of today's SPs to accommodate data management side by side with analytics.

To better illustrate the limitations of the SP model in data management tasks, let us consider a simple bank application that includes typical aspects of data management. Fig. 1 shows part of the application: from the left, users produce a stream of bank requests, which can be either deposits, withdrawals, or transfers. Requests are first processed by a `request validator` operator that stores the amount of money that each user has transferred from or to her account in the last month and blocks further requests when this amount overcomes a given threshold. Requests are then forwarded to the `account balance` operator that retains the current account balance for each user. Both operators consist of four instances (`v1` .. `v4` and `b1` .. `b4`), each responsible for a subset of the accounts.

Although the example is kept simple for the sake of illustration, it highlights patterns that are common to many applications. First, input streams are analyzed on-the-fly to compute fresh statistics (in the example, the amount of money transferred in the last month) and to perform input validation (in the example, to check if the request can be accepted): data cleaning and validation is typical of virtually every system that accepts requests and data from the external environment, including websites that handle requests from users, and monitoring systems that receive data from other devices. Second, some computation results are used to update the state of the application (in the example, the account balance), which is made available to other software systems. Next, we use this example to illustrate the limitations of the SP model in terms of *transactional guarantees* and *queryable state*.

Transactional guarantees. The impossibility to share state between operator instances makes it hard to implement the bank management application in a SP while preserving correctness guarantees. Consider a transfer request and its effect on the state of `account balance`: the request should update the balance of both the provider and the recipient accounts; however, due to partitioning, the two accounts can be stored in different instances of the operator. In this situation, a developer can follow two paths. On the one hand, she can make sure that the state of `account balance` is not partitioned, so that a single instance can process transfer requests. However, this approach gives up on scalability, which is one of the main advantages of SPs. On the other hand, she can split each transfer request into a withdrawal from the provider account and a deposit to the recipient account. Also this second option opens several problems. (1) A deposit should succeed only if the corresponding withdrawal terminates successfully. For instance, if the provider account does not contain enough money, the entire transfer should be discarded. In other words, we would like the transfer to satisfy some *consistency* constraints – take place only if there is enough money in the source account – and to be *atomic* — if it succeeds, it must affect both the provider account and the recipient account, and if it fails, it must affect none of them. Atomicity should extend to multiple operators as well: if a request does not succeed in `account balance`, its effects should also be discarded from `request validator`. Unfortunately, SPs do not offer consistency constraints, nor they enable atomic execution of a group of operations in different instances. (2) Requests should not interfere with each other. Consider for instance the following situation: Bob owns 5$, receives 10$ from Alice, and transfers 10$ to Chuck. If the payment from Alice fails (that is, there is not enough money on Alice's account), Bob should not be able to complete the transfer to Chuck. In other words, we would like transfers to take place in *isolation* and not to access dirty state left by other not yet completed transfers. Again, since SPs do not offer mechanisms to group together the withdrawal and the deposit that are part of a transfer, we cannot ensure that both have been completed before performing further operations. (3) Once a transfer has been performed, it should be stored in the system indefinitely, even in the case of failures. In other words, transfers should be *durable*. While all today's SPs offer fault-tolerance mechanisms, they do not always guarantee that the operations are executed in the same order upon recovery, which might lead to different states after recovering from a failure.

In summary, SPs partition their internal state across operator instances. This enables task and data parallelism but prevents the correct implementation of application scenarios that require ACID (atomic, consistent, isolated, durable) transactional guarantees for state updates, as exemplified by the bank management application above.

Queryable state. Even if SPs retain state information during processing, this state is hidden into operators and cannot be queried and retrieved on demand from outside the SP. To access relevant state, developers must store it in external data management systems. In our banking example, the `account balance` should be modified to output the current state of each account and this information should be used to update an external DBMS. However, this approach would lead to data duplication, with potential waste of resources and additional effort to integrate multiple systems and keep them in a consistent state.

Despite some initial proposals to make the operator state visible [12], no SP supports queries that span multiple operator instances, or considers the consistency of the returned information. For instance, in the case of bank transfers, if we could access the state of multiple accounts from `account balance`, we should see a transfer completed both in the provider and in the recipient accounts, or in none of them. In addition, we should not be allowed to access any dirty state caused by the computation of failing transfers. Finally, once we observe the effects of a transfer, those effects should reflect in any subsequent state access.

In summary, application scenarios such as our bank management application would benefit from query capabilities that retain transactional guarantees.

2.3. Executive summary

To overcome the limitations of SPs in data management tasks, current architectures couple SPs with external data stores, where they duplicate state information. However, the complexity of these architectures forces developers to manually integrate the different sub-systems in a coherent way. They may prove inefficient or overmuch expensive due to the need of replicating data and processing tasks: the input streams of new data get duplicated and processed by a layer responsible for data storage, query, and retrieval, and by a layer responsible for (streaming) data analytics.

We tackle this problem by proposing a novel SP model that enables (i) query to the operator state, and (ii) transactional semantics for read queries and state updates. The model lets developers selectively apply transactional guarantees only to the operators that need them. Moreover, developers can configure the transactional semantics that the system offers by selecting different levels of isolation and durability, thus choosing the best trade-off between performance and consistency for the application at hand.

3. Transactions on a stream processor

We extend the dataflow model of distributed SPs by introducing the concept of *transactional subgraph* (*t-graph*), which identifies a portion of the graph of computation where the state of enclosed operators is accessed and updated with transactional semantics. Each streaming element that enters the t-graph initiates a *read-write transaction*: all its effects on the state of operators within the t-graph are processed as a single transaction with ACID guarantees. The state of operators within the t-graph is also externally queryable through *read-only transactions*. As we will clarify later, by limiting the scope of transactions to t-graphs, the model provides data consistency when needed and high performance when possible. Furthermore, developers can configure the level of isolation and the durability for t-graphs, selecting the best trade-off for the application at hand.

3.1. Stream processing model

Building on the dataflow model of distributed SPs, we represent a computation as a directed graph $G = (N, E)$, where

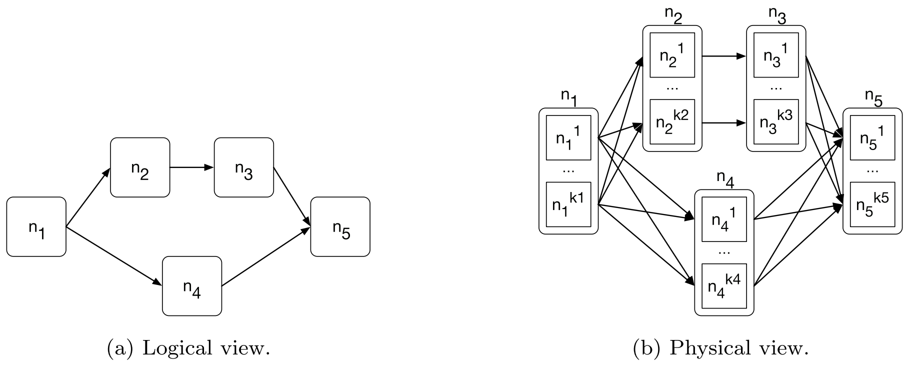


Fig. 2. The topology of a graph of computation: logical and physical view.

the nodes in N are the processing operators and the edges in E are the streams of data between operators. Streams are typed, meaning that all the elements in a stream share the same structure. Fig. 2a shows a graph of computation that includes five operators $n_1 \dots n_5$. An operator can receive input elements from one or more streams and append output elements to one or more streams. Streams originate from sources (such as operator n_1 in Fig. 2a) that receive data from the external environment, and terminate in sinks (such as operator n_5 in Fig. 2a) that return results to the external environment. We abstract the behavior of an operator $n_i \in N$ with a characteristic function f_n that determines how the operator processes input elements, updates its internal state (if any), and produces output elements.

Operators can be replicated in multiple *instances* for scalability, with each instance considering a portion of the input streams. We denote the k instances of an operator $n_i \in N$ as $n_i^1, \dots, n_i^k$. An instance n_i^j processes one input element at a time on a single processing thread, and appends zero, one, or more elements to each of its output streams, according to the characteristic function f_{n_i} of the operator. Fig. 2b shows the physical view of a graph of computation, with multiple instances of each operator.

As in the original SP model, the state of operators is local to each instance such that two instances of the same or different operators do not share any state. Developers control the partitioning strategy through a `keyBy` function, which computes a *key* for a given element. Elements with identical keys are guaranteed to be processed by the same operator instance, which retains any state for that key. The partitioning strategy is relevant in the case of stateful operators. For example, in the bank management application in Fig. 1, account balances are partitioned by account number. All the requests involving a given account need to be processed by the same operator instance, the one that stores that account. Developers can enforce this by indicating the account number as the key of the elements that enter `account balance`.

In general, each instance of an operator n_i can produce elements for any instance of a downstream operator n_j. For example, n_1^1 in Fig. 2b might produce elements for any instance of operators n_2 and n_4. However, if two operators have the same partitioning strategy, then the kth instance of operator n_i (n_i^k) is guaranteed to produce elements only for the kth instance of a downstream operator n_j (n_j^k). This is exemplified by the communication between n_2 and n_3 in Fig. 2b.

We assume that the communication channels between instances (the arrows in Fig. 2b) are FIFO ordered, meaning that the elements are received and processed by the downstream operator in the same order in which they are produced by the upstream operator.[1]

[1] To the best of our knowledge, this assumption holds in all distributed SPs, which usually adopt TCP communication channels.

We define a *causal* relation between stream elements as follows. Element e_1 *causes* element e_2 iff e_2 is produced by an operator instance n_i^j as a result of processing e_1, and we write $e_1 \rightarrow e_2$. We denote by $e_1 \xrightarrow{*} e_2$ the transitive closure of the causal relation.

3.2. Data management model

Our model introduces data management capabilities within *transactional subgraphs* (*t-graphs*). A t-graph $T = (N_T, E_T)$ is a connected subgraph of G that is constrained to have a single input edge in_T. Developers can introduce multiple t-graphs, provided that they do not share any operator. We denote S_T (the *state of* T) as the set of all the stateful operators that are part of the t-graph T. Each operator $s \in S_T$ has a name n_s to make it visible and queryable by name from outside the SP. As in the traditional SP model, an operator $s \in S_T$ processes elements partitioned by key: each operator instance stores the state for the partition it is responsible for in the form of key–value pairs (k, v), $k \in K_s$, $v \in V_s$, where K_s is the key domain and V_s is the value domain for operator s. Keys are unique, meaning that an operator s can store only one value for each key. An operator $s \in S_T$ can be associated with an *integrity constraint* that determines the set of valid values for a given key. In the bank application in Section 2, a developer can introduce an integrity constraint that requires the amount of money for each account to be non-negative.

Each streaming element e entering a t-graph T determines a *read–write transaction*: all the state changes that e induces on S_T take place with transactional semantics. External queries are *read (only) transactions* that retrieve part of the state in S_T with transactional semantics. More precisely, we model the interaction with S_T with two *operations*: *read* and *write*, which access and update the value for a key, respectively. Insert and delete operations are considered as special cases of write. We model a transaction as an ordered set of operations. Queries include only read operations. Read-write transactions include also write operations: they are initiated by an element e entering T and comprise all the operations performed by e or by any element e' caused by e ($e \xrightarrow{*} e'$) on any operator $s \in S_T$ during the execution of its characteristic function f_s. The assumption that a t-graph has a single input edge guarantees that elements entering the t-graph from different input edges cannot simultaneously start transactions that might interfere with each other, for instance if some of their elements are combined in a join operator. For the same reasons, t-graphs do not share operators.

We associate each transaction with a unique *identifier* and we denote t_i as the transaction with identifier i. Transactions can either succeed (*commit*) or fail (*abort*). We call read set R_i the set of keys that transaction t_i only reads and update set W_i the set of keys that transaction t_i also writes. We model the evolution of S_T by associating versions to key–value pairs. Versions can be *created*, *installed*, or *invalidated*. We denote $r_i(s_j^k)$ as a read operation in transaction t_i that reads version j for key k in the stateful operator s. We denote $w_i(s_j^k)$ as a write operation in transaction t_i that creates version j for key k in operator s. If a transaction t_i aborts, it instantaneously invalidates all the versions it created for any key in W_i. If a transaction t_i commits, it instantaneously installs the last version it created for any key in W_i.

Each element that exits in a t-graph T and that belongs to transaction t_i piggybacks the outcome of the transaction – commit or abort – and the set of installed versions, if any. This enables further analysis of the transaction effects downstream.

A *history H* over a set of transactions consists of a partial order among the read and write operations of those transactions. A history is always complete – contains the union of all the operations in all the transactions – and always preserves the order of operations within individual transactions.

3.3. Transactional guarantees

We now formalize the transactional guarantees that our model offers in terms of constraints on the presence and order of operations in the history.

Atomicity. Our model provides atomicity by ensuring that every transaction t_i either installs the last version it created for any key in W_i or invalidates all the versions it created for any key in W_i. This provides "all or nothing" semantics, ensuring that all the effects of a committed transaction are stored and none of the effects of an aborted transaction are stored. No intermediate states are allowed.

Consistency. We enable the developers to specify *integrity constraints* on the value of individual keys in t-graphs. Our model ensures that the state in a t-graph is always *consistent*: for every t-graph T, for every stateful operator $s \in S_T$ and for every key–value pair (k, v) stored in s, the installed version of k satisfies the integrity constraints for k. Since versions are installed by committed transactions, this means that successful transactions bring the t-graph from a consistent state to another consistent state. The versions of aborted transactions are instead invalidated.

Isolation. Isolation limits the interaction between concurrently executed transactions that read and write common keys. Our model allows developers to select different levels of isolation. More relaxed levels introduce fewer constraints and thus enable a higher degree of concurrency and higher performance. Conversely, stricter levels constrain the interaction between transactions more and thus offer higher guarantees but a lower degree of concurrency and performance. We inherit and extend standard isolation levels from the database literature [4] and we present them from the least to the most constraining. Each level subsumes the previous one.

Isolation level PL1 avoids write dependencies between concurrent transactions: the effects of transactions are the same as if their write operations were performed in some sequential order. Specifically, if transaction t_1 installs version v_1 for key k, and transaction t_2 over-writes k by installing version v_2, there should not be another key k' in which the reverse occurs, that is, all writes of t_1 must be ordered before or after all writes of t_2.

Isolation level PL2 additionally requires transactions to only *read* installed versions.[2] Specifically, under PL2, a valid history cannot contain a write operation $w_1(k_v^s)$ where transaction t_1 writes (creates) version v for key k in state s followed by a read operation $r_2(k_v^s)$ where transaction t_2 reads version v, unless t_1 commits and installs v.

Isolation level PL3 additionally prevents transactions from overwriting versions read by other transactions that have not yet completed. Specifically, under PL3, a valid history cannot contain a read operation $r_1(k_{v_1}^s)$ where transaction t_1 reads version v_1 for key k in state s followed by a write operation $w_2(k_{v_2}^s)$ where transaction t_2 writes version v_2 before transaction t_1 is committed or aborted. Isolation level PL3 is also known as (conflict) serializable isolation [11] and ensures that the state of a t-graph is the same as if all the transactions were executed in some sequential order, one after the other.

We further provide a stricter level of isolation that we denote PL4. It extends level PL3 by ensuring that the state of a t-graph is the same as if all the read–write transactions were executed sequentially in the *same* order in which they enter the t-graph. This level corresponds to strict serializability in classic database literature [29].

Durability. Given a t-graph T and its state S_T, durability ensures that the effects that processing an input element e has on S_T persist even in the case of failure. Our failure model considers both software failures in some operator instances and hardware failures in some components of the infrastructure. Our model ensures that the effects of transactions appear as if transactions were executed exactly once in the order expressed by the history, also in the presence of failures. In other words, it is not possible that two (read or read-write) transactions that take place before and after a failure, respectively, observe different orders in the history of operations.

3.4. The model in action

To further clarify our model, we show how it can be used to implement the bank management application in Fig. 1. Recall that `request validator` blocks requests based on the history of requests for a given account, and `account balance` stores the current value of each account. As discussed in Section 2, SPs presented two main limitations in this context: the impossibility to access the internal state of operators and the impossibility to process bank transfers correctly without sacrificing the distribution of data and processing. Fig. 3 shows a possible implementation of the bank application in our model. To guarantee transactional semantics, we include all the operators in a t-graph (dashed box in Fig. 3). We introduce a `split` operator (consisting of two instances in Fig. 3) that processes user requests and redirects them to the instances of the downstream `request validator` and `account balance` partitioned by account — that is, the `keyBy` function specifies the account number as the key for each request. A bank transfer request is split in a withdrawal from the source account and a deposit to the receiver account. Finally, `account balance` has an associated integrity constraint that requires the amount of each bank account to be non-negative.

Our model avoids the problems discussed in Section 2 and correctly processes bank transfers even if the information on bank accounts is partitioned across multiple instances of `request validator` and `account balance`. Indeed, the withdrawal and the deposit that compose a bank transfer originate from a single input request that enters the t-graph and are processed as part of a single read–write transaction with ACID guarantees. Atomicity and consistency ensure that if the withdrawal violates the integrity constraints on the `account balance`, then none of the effects of the request is registered in the state of the operators. Instead, the effects of successful requests reflect on the state of both `request validator` and `account balance`. Isolation

[2] Reading non-installed versions results in the read-uncommitted anomaly [11]. For this reason, PL2 is also referred to as *read-committed* in the ANSI standard [1].

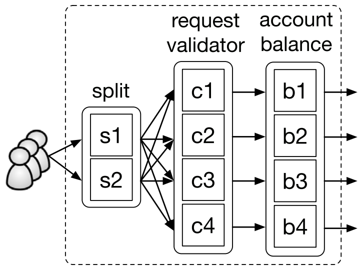


Fig. 3. Implementation of the bank management application in our model.

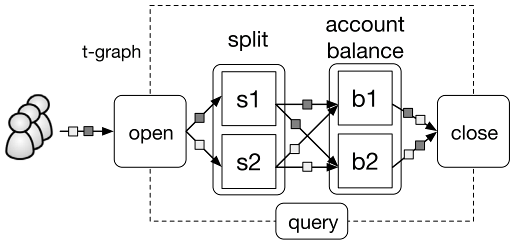


Fig. 4. Bank management application: architecture of the t-graph.

guarantees that two requests do not overlap. By selecting level PL3 or higher, developers ensure that requests behave as if they were executed sequentially. Durability ensures that the effect of successful requests persisted even in the case of failures. Finally, the state of `account balance` is exposed for queries, which are guaranteed to return all the effects of a bank transfer or none of them.

3.5. A comparison with the database model

So far, we discussed how our model enhances SPs with data management capabilities. To better capture the characteristics of our model, it is useful to directly compare it with the classic approach of database systems.

Our model inherits from databases the concept of queryable state through read-only transactions. Differently from databases, read–write transactions cannot be executed on-demand, but are instead statically deployed as t-graphs within the graph of computation of the SP, and are automatically executed whenever new data enters the t-graph. Accordingly, a database adopts a pull model, where on-demand transactions operate on (almost) stationary data, while our model adopts the push model that is typical of data streaming systems, where data is pushed and flows through the stationary operators that build t-graphs.

4. Implementation

We implemented our model in the TSpoon (Transactions ON the Stream ProcessOr) system,[3] which builds on the Apache Flink [14] open-source distributed SP.

4.1. TSpoon API

We illustrate the TSpoon API with an implementation of the bank application from Section 3.4, where we omit the `request validator` for simplicity. The application receives a stream of bank transfer requests, splits each of them into a deposit and a withdrawal, and executes them within a single transaction. Listing 1 shows the code of the application.

Listing 1: Bank transfer example in TSpoon

```
DataStream<Transfer> transferStream = getInputStream(...);

// Open a transactional subgraph
TransactionalDataStream<Transfer> t = transferStream.openTransaction();

// Split a transfer into a withdrawal and a deposit
TransactionalDataStream<BankOperation> opStream =
  t.flatMap(tr  > {
    collector.collect(tr.getDeposit());
    collector.collect(tr.getWithdrawal());
  });

// Apply the deposit/withdrawal to the "account balance" state
// and close the transaction
opStream.keyBy(op  > op.getAccountNumber())
 .map("account␣balance", String.class, Float.class,
         (oldVal, op)  > oldVal + op.getAmount(),
         value  > value >= 0,
         value)
 .closeTransaction();
```

TSpoon augments the Flink API with two `openTransaction` and `closeTransaction` operators to define the boundaries of a t-graph. In Listing 1, TSpoon takes in input a stream of bank transfers `transferStream`. The `openTransaction` opens a t-graph and transforms the input `DataStream` into a `TransactionalDataStream t`. A `flatMap` operator splits each transfer into the corresponding deposit and withdrawal, creating a stream of `BankOperation`. TSpoon offers an overload of several Flink operators that makes the internal state and its changes explicit. In Listing 1, the `account balance` operator is implemented as a stateful `map`: the first three arguments are the name of the operator and the types of the key and value. The fourth argument indicates how the value for a given key is updated when a new bank operation `op` is received. The fifth argument is the integrity constraint on the value. The last argument is the output of the operator. Finally, the `closeTransaction()` closes the t-graph.

External components can submit queries (read transactions) referring to stateful operators (for example, `account balance`) by name. TSpoon supports both the retrieval of individual values by key and predicate queries.

4.2. TSpoon architecture and transactional guarantees

Fig. 4 shows how TSpoon instantiates the t-graph defined in Listing 1. The t-graph contains the `account balance` stateful operator with the current balance of bank accounts partitioned across two instances; `split` is the `flatMap` function that receives transfer requests and redirects them to the instances of `account balance` that are responsible for the bank accounts in each request. As an example, Fig. 4 shows the stream elements involved in the processing of two transfer requests, represented as square boxes of different colors (light gray for one request and dark gray for the other one). Each request is managed by an instance of the `split` operator that transforms the transfer into a deposit and a withdrawal, each handled by an instance of `account balance`. The `close` operator propagates downstream all the results of `account balance` enriched with the outcome and set of changes of the transaction they belong to.

For each t-graph, TSpoon automatically and transparently instantiates the `open`, `query`, and `close` operators, also shown

[3] TSpoon is open-source and publicly available at https://github.com/affo/t-spoon.

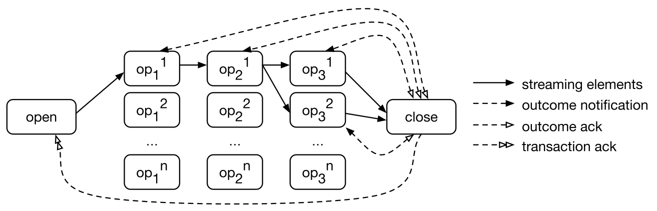


Fig. 5. Implementing transactions: protocol overview.

in Fig. 4, which implement the protocols to process queries and to enforce the required transactional guarantees. In a nutshell, the `open` operator wraps all incoming elements into data structures that carry metadata about transactions. Since individual transactions might be invalidated and re-executed multiple times to satisfy isolation constraints, the `open` operator also stores pending transactions. The `query` operator acts as a proxy for queries (read-only transactions) and ensures that they achieve the desired isolation level.

Fig. 5 overviews the communication that takes place to enforce transactional guarantees while updating stateful operators in a t-graph. Fig. 5 shows a t-graph with three stateful operators op_1, op_2, and op_3, each having n parallel instances. It shows the communication for a transaction that involves operators op_1^1, op_2^1, op_3^1, op_3^2. Stateful operators process the input elements (streaming elements in Fig. 5) and try to apply the requested changes to their local state. They propagate downstream the outcome of the changes, which might also be negative (abort) in the case of the violation of some integrity constraint. The `close` operator collects all outgoing elements to determine the global outcome of a transaction, and communicates it back to the stateful operators involved (outcome notification in Fig. 5). When all these stateful operators acknowledge the communication (outcome ack in Fig. 5), the `close` operator propagates the result of the transaction downstream, and acknowledges the end of the transaction to the `open operator (transaction ack in` Fig. 5). At this point, we say that the transaction is *complete*. Although not shown in Fig. 4, TSpoon can create multiple instances of the `open` and `close` operators to process different transactions in parallel.

4.2.1. Data structures

The `open` operator wraps each incoming element inside a data structure with metadata fields that are accessed and modified by the operators in the t-graph. TSpoon extends the standard Flink operators to provide the same processing semantics when used inside t-graphs, while also dealing with the management of metadata. The metadata for an element e that is part of a transaction t comprise the following fields:

- *id*. The unique identifier of the transaction t.
- ts_{exec}. A sequential timestamp associated to the current execution of t: a transaction may be aborted and re-executed multiple times due to isolation conflicts, in which case it preserves the same *id* but obtains different timestamps at each execution.
- ts_{compl}. The execution timestamp of the last transaction that the `open` operator knows to be complete.
- *fragment*. Tracks the number of element that are part of a transaction t. It enables the `close` operator to compute the number of elements it must receive for t.
- *update*. Changes (write operations) performed on the stateful operators.
- *outcome*. Outcome of the processing performed in the stateful operators: commit, abort (violation of integrity constraints), or retry (violation of isolation constraints).

The `open` operator assigns the fields *id*, ts_{exec}, and ts_{compl}. Each operator in the t-graph that processes an element e_1 producing element e_2 copies these fields from e_1 to e_2. The `open` operator ensures that timestamps are unique. The *fragment* tracks the number of elements that each operator in the t-graph produces. For instance, consider again the bank management application in Fig. 4: when the `split` operator processes a bank transfer request, it generates a deposit and a withdrawal, and uses the *fragment* field to notify downstream that it produced two elements. The `close` operator inspects the *fragment* field to determine when it received all the elements for a transaction. Stateful operators process incoming elements and propagate downstream the state changes they perform – *update* field – and the local *outcome* of the processing, which might indicate the occurrence of errors such as the violation of an integrity constraint.

4.2.2. Atomicity and consistency

TSpoon achieves atomicity and consistency by implementing a two phase commit (2PC) protocol [11], where stateful operators are participants and the `close` operator is the coordinator. The 2PC protocol exploits the sequence of communication steps shown in Fig. 5. Consider a transaction t_i with *id* i and ts_{exec} ts. Consider a stateful operator o that processes an element e that is part of t_i. While processing e, operator o can access its local state and create new versions for its local keys. Operator o decorates the output elements with the *outcome* metadata, which is propagated downstream to all the elements caused by e. The outcome is *commit* if the processing terminates successfully, *abort* if the processing violates some integrity constraints, or *retry* if the processing violates some isolation policy. A *retry* is semantically equivalent to an *abort*, but additionally causes TSpoon to attempt re-executing the transaction.

The `close` operator collects the outcomes from all state operators involved in t_i, using the *fragment* field to determine the number of elements to wait for (streaming elements in Fig. 5). The global outcome for t_i is *commit* if all the instances returned *commit*, *retry* if there is at least one retry, and *abort* otherwise. The `close` operator notifies the global decision to the stateful operators involved (outcome notifications in Fig. 5), which install – in the case of commit – or invalidate – in the case of abort/retry – the versions created for t_i. Every stateful operator involved acknowledges the `close` operator (outcome acks in Fig. 5), which propagates downstream the results of t_i, if the outcome is either *commit* or *abort*, or asks the `open` operator to schedule a new execution for t_i if the outcome is *retry*. The `close` also operator notifies the `open` operator that the transaction with execution timestamp ts is complete (transaction ack in Fig. 5). As we will see, this information is used to ensure isolation.

The protocol above ensures consistency by checking the integrity constraints when accessing stateful operators, and atomicity by either applying or invalidating *all* the changes triggered by a transaction.

4.2.3. Isolation

TSpoon implements isolation through concurrency control protocols. We implemented two alternative protocols. (i) Lock-based protocols (`LB`) lock keys to prevent concurrent access from other transactions. (ii) Timestamp-based protocols (`TB`) use timestamps to determine which version for a key to access within a given transaction. We implement isolation levels `PL2`, `PL3`, and `PL4`, since `PL2` can be implemented with no additional cost with respect to `PL1`. In addition, in the case of `LB` the implementations of levels `PL2` and `PL3` coincide, so we consider only the latter.

Lock-based protocols. In LB protocols, each stateful operator maintains a queue of elements for each key. When an element e belonging to transaction t is processed for key k, t acquires an exclusive lock for k, preventing concurrent accesses. The lock is released when the close operator notifies the global outcome of t (outcome notifications in Fig. 5), which results in installing or invalidating the state changes performed by t. Each subsequent element e' from a transaction t' that wants to access the same key k waits in the input queue. This ensures that e' accesses the version installed by e, if t commits, or the previously installed version, if t aborts.

This strategy serializes operations on individual keys, but still allows write operations from concurrent transactions to be installed in different orders in different instances of one or more operators, which violates the requirement of PL1. Consider for example the two bank transfer requests in Fig. 4 and assume that they involve the same two bank accounts a_1 and a_2. Since the requests are handled concurrently in the split operator, it is possible that account a_1 processes the light gray request first, while account a_2 processes the dark gray request first. TSpoon prevents this violation by forcing transactions to execute in order with respect to their execution timestamp ts_{exec}. In particular, it aborts transactions that attempt to execute operations out of timestamp order and schedules them for re-execution (with a higher timestamp). This strategy avoids deadlocks: if transaction t_2 is waiting for some operation of transaction t_1 to complete, then the execution timestamp of t_2 is larger than that of t_1. As a consequence, if some operator receives an element from t_1 after an element from t_2, it aborts t_1 preventing it from locking any resource. Finally, to further reduce the probability of re-executing transactions, the LB protocol reorders queued elements according to their execution timestamps while they wait to acquire some resource.

The above protocol ensures that a transaction can access a key only when the previous updates to that key have been installed, thus preventing the read-uncommitted anomaly. Moreover, all the (read and write) operations that involve the same keys are executed in timestamp order. This ensures that the results of transactions are the same as if they were executed sequentially in timestamp order. Thus, this algorithm guarantees isolation level PL3.

The protocol, however, does not guarantee isolation level PL4, since in the case of re-execution the order of execution timestamps may not reflect the order of transaction ids. If the developer requires level PL4, TSpoon introduces an additional sequencer component before *each* stateful operator in a t-graph, which consists of a single instance and reorders transactions according to their *id*. The sequencer adopts the same mechanism (based on *fragment*) of the close operator to determine the number of elements that compose a transaction.

Timestamp-based protocols. TB protocols do not lock resources, but rather use timestamps to ensure that transactions always read/update versions that are consistent with the desired isolation level. In the case this is not possible, they abort transactions and schedule them for re-execution. In the following, we define the timestamp of a version as the timestamp of the transaction that created that version.

PL2 Isolation level PL2 constrains the order between writes. To enforce it, we ensure that the write operations of two transactions are executed in timestamp order everywhere: we enable a transaction to update (write) a key only if there is no other version for the same key with a higher timestamp, otherwise we abort and retry the transaction. PL2 further prevents transactions from reading versions that have not been installed yet. To ensure this, we force a transaction t to always read the latest version of a key that is known to be installed when t starts executing. Recall that the open operator assigns to each transaction a ts_{compl} that is the timestamp of the last transaction that it knows to be complete. Thus we force a transaction t with ts_{compl} t_c that wants to read key k to access the latest version of k with timestamp lower or equal to t_c.

PL3 Isolation level PL3 requires transactions to execute as if they were performed sequentially. To ensure this, we force transactions to read installed versions as in PL2, and we enable them to update a key only if there is no version for the same key higher than ts_{compl}. Although similar, PL2 and PL3 differ with respect to the constraints on the versions that they can read and write, which influence the transactions that are re-executed due to the violation of such constraints.

PL4 Isolation level PL4 requires transactions to execute as if they were performed sequentially in *id* order. To ensure this property we adopt the same implementation as in PL3 but we add a component before the close operator, consisting of a single instance that collects all the elements from all the transactions, orders them by their *id*, and checks for violations in the order of execution. If this is the case, it aborts violating transactions and schedules them for re-execution.

4.2.4. Queries

TSpoon provides access to the state of a t-graph using the query operator as a proxy. It ensures that queries access a consistent snapshot of the t-graph by obtaining the timestamp of complete transactions from the open operator. When trying to access a key, a query (that is, a read-only transaction) is assigned the timestamp t_c of the last completed (committed or aborted) transaction, and it always accesses the latest version with timestamp lower or equal to t_c.

4.2.5. Durability

TSpoon provides durability by relying on and extending the fault-tolerance algorithm of Flink, based on distributed snapshotting [13]: special markers periodically flow through the network of operators from sources to sinks; upon receiving a marker, each stateful operator stores its state to some durable storage and propagates the marker downstream. Upon failure, operators restore their state from the last snapshot, and sources replay all the elements that were not part of the snapshot.

TSpoon cannot reuse this mechanism out of the box to save the state of t-graphs for two reasons. (i) In the t-graph some state changes are asynchronous with respect to the flow of stream elements in the network of operators, and thus the markers might not capture a consistent snapshot of the state. For example, the close operator communicates back to stateful operators the global outcome of a transaction, which determines if the versions created by that transaction are installed or invalidated. (ii) In the case of re-executions, elements could be processed in a different order with respect to the first execution. This could violate durability guarantees: for example, two queries that take place before and after a failure could observe the effects of transactions in different orders.

To overcome these problems, TSpoon integrates the Flink algorithm with a Write Ahead Log (WAL) that stores the operations of successful transactions. The close operator registers on the WAL all state changes performed by a transaction right before forwarding the results of that transaction downstream. For each key in a stateful operator, the WAL preserves only installed updates and the exact order in which they were installed. The WAL is made available to all the operators in the t-graph in the case of recovery through external storage services such as a distributed filesystem.

Upon recovery, we can identify three kinds of transactions. (i) Transactions whose installed versions are stored in the last snapshot of stateful operators. (ii) Transactions whose installed versions are stored in the WAL but not in the last snapshot of stateful operators. (iii) Transactions that are not stored in the WAL (not yet completed).

TSpoon restores the updates of the first type of transactions from the Flink snapshot. Since they are part of the snapshot, Flink does not try to replay them. Then, it restores the effects of the second type of transactions from the WAL, thus ensuring that they are applied in the same order as in the original execution. Since these transactions are not part of the snapshot, Flink attempts to replay them. The `open` operator discards these re-executions, since it knows that the replay is performed through the WAL. Finally, the `close` operator propagates the results of these transactions downstream, to enable the recovery of downstream operators. The third type of transactions were not yet completed at the time of failure. TSpoon restarts their execution when the `open` operator receives the corresponding input elements.

5. Evaluation

TSpoon aims to reduce the complexity of data processing and management architectures. To achieve this goal and be useful in practice, it has to provide an adequate level of performance in terms of the volume and velocity of data it can handle. Our TSpoon prototype builds on top of Flink 1.3.2 and offers exactly the same performance of Flink for pure stream processing tasks that do not use transactional capabilities. Hence, our evaluation assesses the behavior of TSpoon in the presence of transactions, with four main goals: (i) study the absolute performance of TSpoon against a state-of-the-art solution for data management in distributed environments; (ii) investigate the trade-off between performance and transactional guarantees with different levels of isolation and durability; (iii) compare `LB` and `TB` protocols; (iv) study how other workload parameters affect the performance of TSpoon.

5.1. Experiment setup

We deploy TSpoon on a cluster of 5 Amazon EC2 t2 xlarge instances (with 4 CPU cores and 16 GB of RAM) and 15 t2 large instances (with 2 CPU cores and 8 GB of RAM), for a total of 50 CPU cores and 184 GB of RAM. As a default scenario, we consider the bank application presented in Section 4: TSpoon receives a stream of input bank transfers, and splits each transfer in a deposit and a withdrawal that are processed within a single transaction. We store 100k bank accounts partitioned across 50 instances of the `account balance` stateful operator, one for each CPU core. The source and destination accounts for each bank transfer are selected randomly following a uniform distribution.

We assess the performance of TSpoon by measuring its throughput and latency. We measure the average latency when the system is unloaded, by submitting input requests sequentially, sending an element at a time. In terms of throughput, we want to determine the maximum value of input elements that TSpoon can sustain before becoming overloaded and losing responsiveness (we call this "the *sustainable throughput*"). In practice, for each experiment we increase the input rate stepwise until the latency overcomes a given threshold (20 times the latency of the unloaded system). We repeat all experiments at least 8 times. For each measure, we plot the average value and the standard deviation.

5.2. Testing correctness

In addition to the usual techniques for unit testing, which we put in place throughout the entire implementation phase of TSpoon, we also took advantage of the distributed deployment discussed in Section 5.1 to perform extensive system testing under various operating conditions. In particular, we tested the correctness of atomicity and isolation protocols by initiating many read–write transactions while concurrently reading the state with read-only transactions and checking its validity with respect to consistency constraints and to the desired level of isolation. During our testing, we adopted a high rate of input elements and a low number of keys for stateful operators, to increase the probability of conflicting accesses to state that could reveal implementation errors. Finally, we performed the same testing experiments in the presence of failures, by randomly crashing operators within t-graphs. This allowed us to also test our protocol for durability.

Table 1
Default scenario: comparison with Flink and VoltDB.

	Latency	Sust. throughput
TSpoon	3.57 ms	8580 el/s
Flink	0.6 ms	59 609 el/s
VoltDB	4.62 ms	344 tr/s

5.3. Default scenario

We use the default scenario described above to compare TSpoon against Flink 1.3.2 and the VoltDB in-memory distributed DBMS version 8.0,[4] which are state-of-the-art representatives of their categories, well known for their excellent level of performance. We configure both Flink and TSpoon to deploy 50 instances of each operator — one per CPU core, a typical Flink configuration. We set the level of isolation for TSpoon to `PL3` (the same adopted by VoltDB) and we use `TB` concurrency control. VoltDB only enables developers to configure the number of database partitions per machine. We configure 2 partitions per machine, for a total of 40 partitions, since 15 out of 20 nodes in the cluster have 2 CPU cores, and we do not want to overcommit the available resources. We implement the bank transfer transaction as a stored procedure that is analyzed and compiled at deployment time to eliminate the overhead of creating a query plan at runtime. We compute the throughput and latency of VoltDB using the provided benchmarking tools.[5] They measure the maximum throughput by submitting 200k bank transfer transactions in a single burst, and then computing the average latency with an input rate that is below the maximum throughput. The comparison is fair, since the maximum throughput is an *over*-estimation of the sustainable throughput.

Table 1 shows the results we measured. TSpoon achieves a sustainable throughput of more than 8500 input elements/s with 3.57 ms latency. By comparison Flink processes 59 609 input elements/s with an average latency of 0.6 ms. However, the application implemented in Flink differs from that implemented by TSpoon and VoltDB as Flink does not provide any transactional guarantee and the results it produces can violate the requirements of our bank transfer scenario. In practice, these tests measure the overhead of TSpoon in enforcing transactional guarantees. Flink can process deposits and withdrawals in parallel, in any order, while TSpoon introduces concurrency constraints to enforce isolation and atomicity.

As a more fair comparison, VoltDB offers the same transactional guarantees of TSpoon, but achieves a throughput of only 344 transactions/s with a latency of 4.62 ms. Indeed, VoltDB is optimized for transactions that involve a single partition and

4 https://github.com/voltdb/voltdb.
5 https://github.com/VoltDB/voltdb/blob/master/examples/voter/README.md.

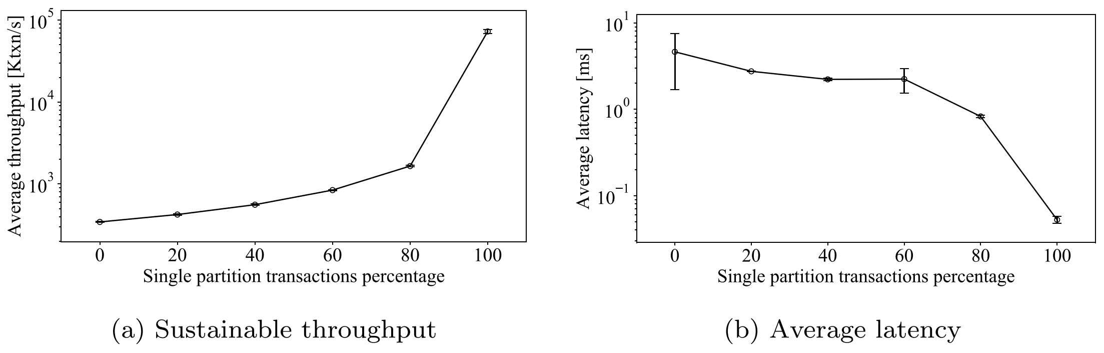


Fig. 6. VoltDB: throughput and latency when increasing the percentage of single-partition transactions.

this result indicates that multi-partition transactions are very expensive. We repeated the same experiment by submitting an increasing percentage of single-partition transactions (individual bank deposits and withdrawals). As Fig. 6 shows, we observed an increasing throughput and decreasing latency, up to more than 73k transactions/s and a latency of 0.05 ms when considering only single-partition transactions. In contrast, TSpoon delivers consistent performance independently on the type of transactions. We plan to implement optimizations for single-partition transactions inspired by VoltDB as future work.

In summary, the above results demonstrate that TSpoon is competitive with state-of-the-art data processing and management systems: it provides the same performance as Flink in pure stream processing tasks, reduces the throughput by less than 7× when providing strong (PL3) transactional guarantees, and significantly outperforms VoltDB in terms of multi-partition transactional updates.

5.4. *Isolation levels and concurrency control strategies*

Fig. 7 shows the performance of TSpoon in our default scenario with different isolation levels (PL2, PL3, PL4) and concurrency control strategies (LB and TB). Moving from PL2 to PL3 does not introduce a significant drop in throughput (Fig. 7a) or latency (Fig. 7b). Indeed, our default scenario includes a large number of bank accounts that lead to minimal state access conflicts. At level PL3, LB and TB protocols exhibit comparable behaviors, with a small advantage of TB in terms of throughput. PL4 is clearly more expensive, leading to a throughput of about 2280 elements/s for LB and about 3100 elements/s for TB. Indeed, the strong requirement of processing transactions in *id* order demands for a single-instance operator that enforces this order. LB imposes the order upfront, while TB checks the order before transactions complete, aborting and rescheduling those transactions that violate it. In our default scenario, the first strategy is more expensive and leads to an increase in latency (up to 7.5 ms).

5.5. *Sensitivity to parameters*

We now investigate how workload parameters influence TSpoon.

5.5.1. *Chain of updates*

We first consider a chain of updates performed one after the other, mimicking a scenario where the data produced by a state update is elaborated downstream and produces updates in other operators. We consider both the case in which all the involved stateful operators belong to the same t-graph and the case in which each stateful operator belongs to a different t-graph. Each stateful operator includes 100k different keys, as in our default scenario. Fig. 8 shows that the throughput decreases with the number of stateful operators, both in the case of a single t-graph and in the case of multiple t-graphs. In the case of a single t-graph (Fig. 8a), the elements of a transaction traverse the entire pipeline of operators before the transaction complete. The longer the pipeline, the higher the probability of conflicts between transactions. This problem does not occur in the case of different t-graphs (Fig. 8b), which instead introduce the overhead of opening and closing multiple transactions. The throughput is higher in the case of a single t-graph, meaning that opening and closing multiple t-graphs are more expensive than processing a single, longer transaction.

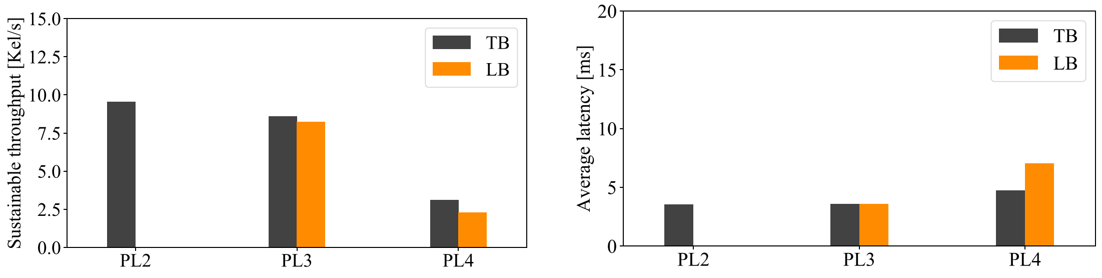
In the case of a single t-graph, the latency only slightly increases when moving from one to eight stateful operators due to the longer path from sources to sinks, and remains below 30 ms for all the configurations we tested (Fig. 8c). In the case of multiple t-graphs, the latency increases more (up to almost 80 ms in the case of PL4 with TB protocol), due to the presence of an additional `open` and `close` operators for each t-graph.

5.5.2. *Parallel updates*

Fig. 9 shows the performance of TSpoon when considering state updates that occur in parallel, in a single t-graph or in distinct t-graphs. In the case of a single t-graph (Fig. 9a), the throughput decreases with the number of stateful operators. Indeed, a higher number of stateful operators increases the probability of state access conflicts and also forces the `close` operator to wait for more outcomes. For protocols up to PL3, the throughput decreases from about 8000 elements/s to less than 2500 elements/s. The overhead of PL4 protocols dominates the costs associated to the increased number of stateful operators, leading to the same throughput from one to eight stateful operators. In the case of multiple t-graphs (Fig. 9b), the throughput remains almost constant, since TSpoon can process transactions entirely in parallel. The latency, not reported for space reasons, also remains almost constant in all the scenarios we tested.

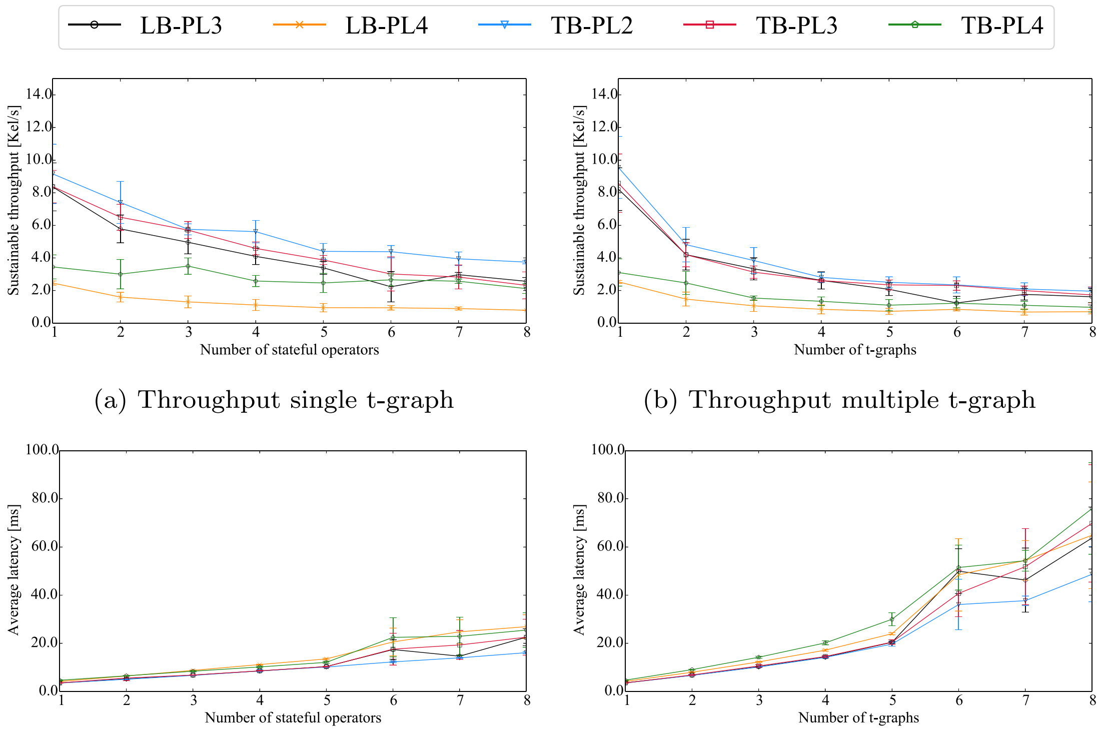
5.5.3. *Number of keys*

We now study how the probability of state access conflicts between transactions influences the performance of TSpoon. We consider again our default scenario and we change the number of keys (bank accounts) within the `account balance` operator. As Fig. 10 shows, LB tolerates access conflicts better, and its throughput does not significantly decrease even in the extreme case of only 100 keys. Instead, TB is more affected. The throughput decreases when reducing the number of keys for all isolation levels. In the case of PL2, the throughput remains stable from 10k to 1000 keys, and decreases with fewer keys from 8900 to less than 6200 elements/s. State access conflicts influence PL3 and PL4 the most, since these two levels of isolation introduce more constraints and increase the number of transactions that have to be re-executed. In both cases, the throughput decreases



(a) Sustainable throughput (b) Average latency

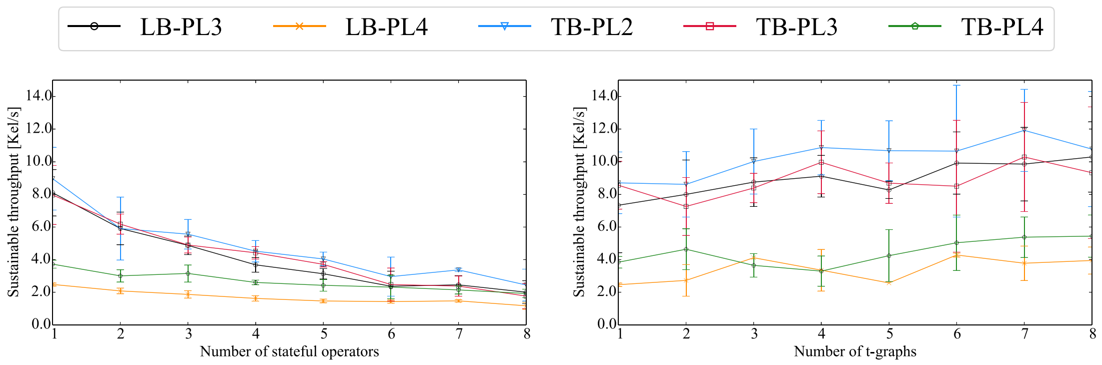
Fig. 7. Default scenario: comparison of isolation levels and concurrency control strategies.



(a) Throughput single t-graph (b) Throughput multiple t-graph

(c) Latency single t-graph (d) Latency multiple t-graph

Fig. 8. Chained updates.



(a) Throughput single t-graph (b) Throughput multiple t-graph

Fig. 9. Parallel updates.

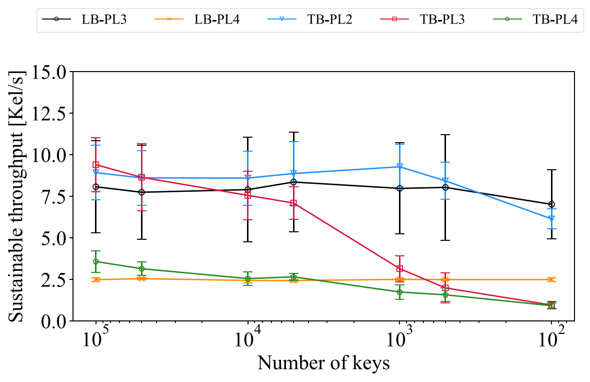


Fig. 10. Number of keys.

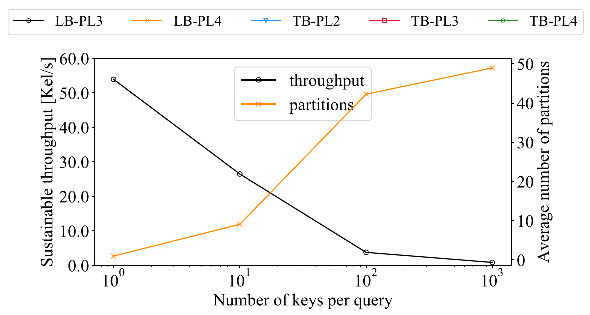


Fig. 11. Read queries.

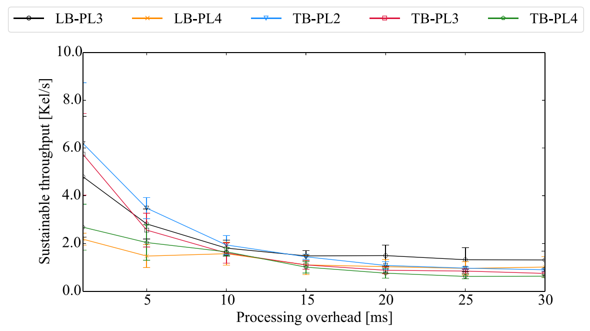


Fig. 12. Processing overhead.

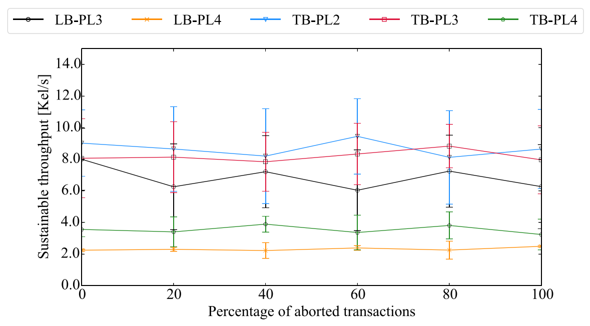


Fig. 13. Percentage of aborted transactions.

to less than 1000 elements/s (lower than PL4 with LB protocol). The latency, not reported for space sake, remains almost stable when changing the number of keys. Indeed, we measure latency when the system is not overloaded and there are no state access conflicts.

5.5.4. Processing overhead

We now investigate how TSpoon behaves when more expensive computations take place within a t-graph. We expect most application scenarios to include only simple state updates in t-graphs, leaving outside of t-graphs more complex computations such as data analytics tasks, which do not need transactional guarantees. Nevertheless, we are interested in seeing how robust is TSpoon to the presence of processing overhead within t-graphs.

We set up an experiment where we artificially add processing overhead (in the form of busy wait loops) to each stateful operator within a t-graph. We then repeat our default scenario while increasing the processing overhead from 1 ms to 30 ms. As Fig. 12 shows, an increased processing ovehead leads to a lower sustainable throughput for all the protocols we tested. The decrement is more visible for protocols up to PL3, since PL4 protocols start from a lower throughput due to the more expensive isolation algorithms. Nevertheless, the throughput curve flattens after a processing overhead of 15 ms, showing that TSpoon can successfully commit transactions at any isolation level even when they include some processing overhead. The latency, not reported for space sake, increases linearly with the processing overhead, and remains below 50 ms for all the protocols even in the presence of a processing overhead of 30 ms.

5.5.5. Number of aborted transactions

Fig. 13 shows how the performance of TSpoon changes when a given percentage of transactions abort due to a violation of consistency constraints. For all the protocols we implemented, there is no visible difference in throughput and latency when increasing the percentage of aborted transactions. Indeed, the behavior of the protocols does not depend on the outcome of transactions. This is different with respect to transactions aborted due to state access conflicts and rescheduled for execution, which have been discussed in the previous section.

5.5.6. Queries

We now study the throughput for external queries. Since the isolation level does not affect queries, we fix it to PL3. We use our default scenario with a fixed rate of updates of 1000 bank transfer requests/s and we change the selectivity of queries, that is, the number of accounts that each query selects. The yellow line in Fig. 11 shows the average number of `account balance` instances each query accesses. The black line shows the sustainable throughput for queries: TSpoon supports 53 900 queries/s when accessing a single instance of `account balance`. As the number of involved instances increases, the throughput decreases, reaching 890 queries/s in the extreme case in which a query accesses 1000 keys. Query latency, not reported for space sake, ranges from less than 1 ms when querying a single partition to 10 ms when querying all partitions.

5.5.7. Cost of durability

When durability is enabled, TSpoon persists the results of completed transactions in a Write Ahead Log (WAL) on disk before propagating them downstream. This introduces a runtime overhead due to disk access. We measure such overhead in our default scenario, with isolation level PL3 and LB protocol. When the system is unloaded, writing to the WAL introduces a negligible increase in latency. However, when the rate of input transactions increases, the need of continuous I/O operations influences the throughput that TSpoon can sustain, which in our default scenario decreases from 8350 to 5448 elements/s when durability is enabled.

Next, we measure the time to recover from a failure. Recall that we rely on the Flink snapshot algorithm for fault-tolerance,

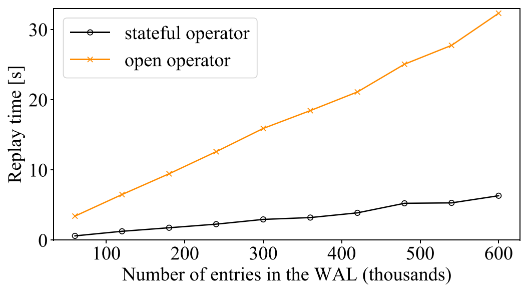


Fig. 14. Number of pending transactions.

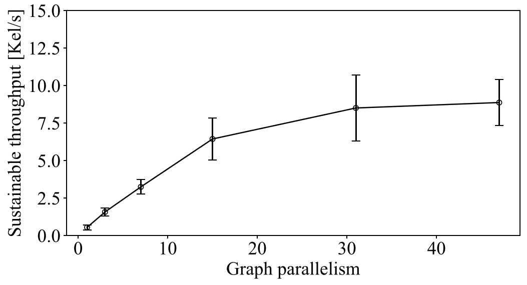


Fig. 15. Number of partitions.

which we augment with the WAL to ensure transactional semantics. The time to recover includes (1) the time to discover a failure; (2) the time to restore Flink to the last snapshot; (3) the time to restore the state of t-graphs from the WAL. The first two contributions only depend on Flink and its configuration. Thus, we measure the last contribution, which is specific to our model. Restoring the state of t-graphs involves the costs to restore: (i) the local state of each stateful operator; (ii) the state of the `open` operator. These two operations are performed in parallel, and so the overall cost of recovery is the maximum of the two contributions. Fig. 14 shows the cost of these two contributions in our default scenario, when changing the number of elements stored in the WAL (which depends on the input rate of transactions and on the frequency between two snapshots). In our scenario, the cost to restore the `open` operator dominates the cost to restore each stateful operator, resulting in a recovery time of 3.5 s with 60k transactions in the WAL and 32 s with 600k transactions in the WAL.

5.6. Scalability

Fig. 15 shows how TSpoon scales with the number of CPU cores (and, correspondingly, the number of partitions for the stateful operators). In our default scenario, the throughput increases from 544 to 6436 elements/s (11.8×) when moving from 1 core to 16 cores. After this threshold, adding new cores brings fewer benefits, and the throughout only increases to 8866 with 48 cores.

6. Related work

The inter-disciplinary nature of our work relates it to several fields, including stream processing, database systems, and data management architectures.

Processing streams of data. The last decade saw an increasing interest in technologies to process streams of data, with several systems being proposed by the academia and from the industry. A first generation of SPs flourished in the mid 2000s, focusing on the definition of abstractions to: (i) query streams of data, as in Data Stream Management Systems (DSMSs), or (ii) detect situations of interest from streams of low-level information, as in Complex Event Processing (CEP) systems [17]. Initially, SPs were developed as centralized components or libraries, despite some early proposals to model processing tasks as a directed graph of operators to be deployed in distributed infrastructures [2,3,21].

A second generation of SPs has its roots in the research on Big Data, aiming to process large volumes of streaming data in cluster environments. The research on Big Data initially focused on static data and batch processing and proposed functional abstractions such as MapReduce [18] to automate the distribution of processing. Subsequent proposals increased the expressiveness of MapReduce, enabling the developers to specify complex directed graphs of operators [34]. These systems assume long running computations and provide fault-tolerance mechanisms to resume intermediate results if they are lost due to the failure of one or more machines in a large cluster [33]. The second generation of SPs inherits the same processing model based on a graph of functional operators, but focuses on dynamic rather than static datasets. Some of them, for instance Spark Streaming [35], provide streaming computations on top of batch processing by splitting each stream into small static chunks (micro-batches). Other SPs provide native support for streaming computations, where stream elements move from an upstream operator to a downstream operator as soon as the former has completed its processing task. This is the case of Storm [32], Heron [23], Google DataFlow [6], and Flink [14], which we adopt and extend in this paper.

Related to processing dynamic data, the programming language community proposed reactive programming (RP) abstractions [8], which build on three pillars 1. time-changing variable and explicit definition of their dependencies; 2. automated propagation of changes. RP shares many similarities with stream processing, with the graph of dependencies between variables being analogous to the graph of computation in SP. Some recent proposals in the field study the trade-off between consistency and performance in distributed RP, which is closely related to the topic of this paper [20,25,26].

Distributed databases. Distributed relational databases provide ACID transactional guarantees through distributed commit protocols, concurrency control algorithms [11], and recovery mechanisms [28]. TSpoon builds on these concepts to integrate transactional semantics within the SP. The tension between strong consistency and guarantees in (distributed) data management systems has been widely investigated, leading to the formulation of various levels of isolation [1]. Indeed, with the increasing size of data-intensive applications [22], a number of systems trade consistency and strong transactional semantics for scalability. To the extreme of this approach, NoSQL databases limit or drop transactions: Dynamo [19] was born without transactional support; MongoDB only supports transactions that involve a single document [9], and Redis does not support arbitrary distributed transactions [15].

More recently, NewSQL database systems aim to reconcile strong transactional semantics and efficient distributed data management for some workloads. H-Store [31] is an in-memory database that enforces atomicity of transactions on single partitions of the state through single threaded computations, and schedules multi-site operations to ensure ACID properties. Furthermore, H-Store enables transaction optimization by enhancing the support for pre-compiled stored procedures. This approach later evolved in the VoltDB database system that we used in our evaluation [24]. S-Store defines stream processing capabilities on top of an OLTP system (H-Store), implementing streams as time-varying tables and stream processing as triggers [16]. In S-Store, a transaction is a directed acyclic graph of stored procedures calls that can access the whole underlying database. Instead, our approach limits the scope of individual operators to enable intra-transactional parallelism. Actor-Oriented Database Systems (AODB) target our same goal, and develop transactions as actor-oriented programs [10,30]. They provide lower-level primitives than TSpoon: parallelism is explicit and implemented by the developers using asynchronous messages between actors.

Big data architectures. Some data processing architectures aim to solve the dichotomy between consistent state management and low-latency stream processing. The Lambda Architecture provides low-latency results, but serves exact yet "old" results in case of failures [27]. It was conceived when SPs did not provide full support for distributed, fault-tolerant, and stateful computation and where used as a fast *speed layer* that could potentially provide wrong results in the case of failures. Thus the speed layer is coupled with a *batch layer* that runs periodic batch jobs to generate higher-latency but exact results. When the data is queried, the *serving layer* encapsulates the complex logic that integrates the results of the speed layer (recent, but possibly inaccurate) or those of the batch layer (accurate, but possibly outdated). More recent proposals criticize the complexity of this architecture and advocate stream-only solutions.[6] To enable these architectures, some SPs introduce the concept of queryable state [12], although focusing on individual operators without providing transactional guarantees as we do. Our proposal can be considered as an evolution of these architectures that moves transactional updates directly on the SP.

7. Conclusions

Data-intensive applications increasingly often combine consistent data management with analytics on large volumes of dynamic (streaming) data. Current architectures satisfy these needs by exploiting multiple subsystems, but this leaves developers with the daunting task of coherently integrating these subsystems. We propose a novel model that seamlessly integrates transactional state management within a distributed stream processor. The model introduces transactional regions within dataflow computation graphs: each element entering a transactional region initiates a read–write transaction, and the internal state of the region can be queried with read-only transactions. We implement the model in the TSpoon system, which offers different levels of isolation and durability to let developers choose the best trade-off between performance and consistency for the application at hand. We evaluate TSpoon thoroughly measuring its performance under different workloads, transactional semantics, and implementation strategies, and showing that it can outperform state-of-the-art data management tools in common scenarios. We are confident that this work has the potential to open a new line of research and innovation, and lead to architectures that are more efficient and easier to design, develop, and maintain.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

6 http://milinda.pathirage.org/kappa-architecture.com/.

References

[1] A. N. S. for Information Systems, Ansi x3.135-1992, database language sql, 1992.

[2] D.J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A.S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the borealis stream processing engine, in: Proceedings of the Conference on Innovative Data Systems Research, in: CIDR '05, Asilomar, CA, 2005, pp. 277–289.

[3] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: A new model and architecture for data stream management, VLDB J. 12 (2) (2003) 120–139.

[4] A. Adya, B. Liskov, P. O'Neil, Generalized isolation level definitions, in: Proceedings of the International Conference on Data Engineering, in: ICDE '00, IEEE, 2000, pp. 67–78.

[5] L. Affetti, A. Margara, G. Cugola, Flowdb: Integrating stream processing and consistent state management, in: Proceedings of the International Conference on Distributed and Event-Based Systems, in: DEBS '17, ACM, 2017, pp. 134–145.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R.J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, VLDB J. 8 (12) (2015) 1792–1803.

[7] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M.J. Sax, S. Schelter, M. Höger, K. Tzoumas, D. Warneke, The stratosphere platform for big data analytics, VLDB J. 23 (6) (2014) 939–964.

[8] E. Bainomugisha, A.L. Carreton, T.v. Cutsem, S. Mostinckx, W.d. Meuter, A survey on reactive programming, ACM Comput. Surv. 45 (4) (2013) 52:1–52:34.

[9] K. Banker, MongoDB in Action, Manning Publications Co., Greenwich, CT, USA, 2011.

[10] P.A. Bernstein, S. Bykov, A. Geller, G. Kliot, J. Thelin, Orleans: Distributed virtual actors for programmability and scalability, Tech. rep., Microsoft, mSR-TR-2014–41, 2014.

[11] P.A. Bernstein, N. Goodman, Concurrency control in distributed database systems, ACM Comput. Surv. 13 (2) (1981) 185–221.

[12] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, K. Tzoumas, State management in apache flink®: consistent stateful distributed stream processing, Proc. VLDB Endow. 10 (12) (2017) 1718–1729.

[13] P. Carbone, G. Fóra, S. Ewen, S. Haridi, K. Tzoumas, Lightweight asynchronous snapshots for distributed dataflows, 2015, arXiv preprint arXiv:1506.08603.

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.

[15] J.L. Carlson, Redis in Action, Manning Publications Co., Greenwich, CT, USA, 2013.

[16] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, et al., S-store: a streaming newsql system for big velocity applications, Proc. VLDB 7 (13) (2014) 1633–1636.

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

[18] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, in: Proceedings of the Symposium on Operating Systems Principles, in: SOSP '07, ACM, 2007, pp. 205–220.

[20] J. Drechsler, G. Salvaneschi, R. Mogk, M. Mezini, Distributed rescala: An update algorithm for distributed reactive programming, in: Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications, in: OOPSLA '14, ACM, New York, NY, USA, 2014, pp. 361–376.

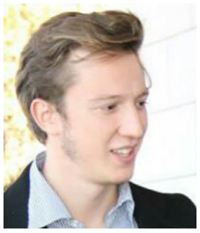
[21] O. Etzion, P. Niblett, Event Processing in Action, Manning Publications, Greenwich, CT, USA, 2010.

[22] M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, O'Reilly, 2017.

[23] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J.M. Patel, K. Ramasamy, S. Taneja, Twitter heron: Stream processing at scale, in: Proceedings of the International Conference on Management of Data, in: SIGMOD '15, ACM, New York, NY, USA, 2015, pp. 239–250.

[24] N. Malviya, A. Weisberg, S. Madden, M. Stonebraker, Rethinking main memory oltp recovery, in: Proceedings of the International Conference on Data Engineering, in: ICDE 2014, IEEE, 2014, pp. 604–615.

[25] A. Margara, G. Salvaneschi, We have a DREAM: Distributed reactive programming with consistency guarantees, in: Proceedings of the International Conference on Distributed Event-Based Systems, in: DEBS '14, ACM, New York, NY, USA, 2014, pp. 142–153.
[26] A. Margara, G. Salvaneschi, On the semantics of distributed reactive programming: the cost of consistency, IEEE Trans. Softw. Eng. (Preprint) 99 (0) (2018) 1–25.
[27] N. Marz, J. Warren, Big Data: Principles and Best Practices of Scalable Realtime Data Systems, Manning Publications Co., Greenwich, CT, USA, 2015.
[28] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, ACM Trans. Database Syst. 17 (1) (1992) 94–162.
[29] R. Sethi, Useless actions make a difference: Strict serializability of database updates, J. ACM 29 (2) (1982) 394–403.
[30] V. Shah, M.A. Vaz Salles, Reactors: A case for predictable, virtualized actor database systems, in: Proceedings of the International Conference on Management of Data, in: SIGMOD '18, ACM, 2018, pp. 259–274.
[31] M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, P. Helland, The end of an architectural era (it's time for a complete rewrite), in: Proceedings of VLDB, in: VLDB '07, VLDB Endowment, 2007, pp. 1150–1160.
[32] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, Storm@twitter, in: Proceedings of the International Conference on Management of Data, in: SIGMOD '14, ACM, New York, NY, USA, 2014, pp. 147–156.
[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the Conference on Networked Systems Design and Implementation, in: NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, p. 2.
[34] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: Proceedings of the Conference on Hot Topics in Cloud Computing, in: HotCloud'10, USENIX Association, Berkeley, CA, USA, 2010, p. 10.
[35] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: Fault-tolerant streaming computation at scale, in: Proceedings of the Symposium on Operating Systems Principles, in: SOSP '13, ACM, New York, NY, USA, 2013, pp. 423–438.



Lorenzo Affetti is Ph.D. Student in Computer Science and Engineering at Politecnico di Milano in distributed stream processing. His work, published in the ACM DEBS conference in 2017, focuses on stream processing as a general-purpose programming paradigm for modern, event-driven applications. As such, it includes a model for transactional processing on the stream processor, and its implementation on the open-source stream processor Apache Flink. Lorenzo obtained his Master Degree Cum Laude at Politecnico di Milano in 2015.

Alessandro Margara is an assistant professor at Politecnico di Milano. His research interests focus on middleware solutions to ease the design and development of complex distributed systems, with a special emphasis on event stream processing and reactive systems. His work includes the definition of languages for event and stream processing and the implementation of parallel and distributed algorithms to efficiently support such languages. Alessandro obtained his Ph.D. Cum Laude from Politecnico di Milano. He has been a postdoctoral researcher at Vrije Universiteit Amsterdam and at Università della Svizzera italiana, Lugano. Some of Alessandro's recent publications appear in ACM Computing Surveys, IEEE TSE, IEEE TPDS, ICSE, ICDCS, DEBS, EuroSys, Middleware.

Gianpaolo Cugola received his Dr. Eng. degree in Electronic Engineering and his Ph.D. in Computer Science from Politecnico di Milano, where he spent most of his professional life. He is currently Full Professor at Politecnico di Milano where he teaches several courses in the area of Computer Science. He is co-author of tens of scientific papers published in international journals and conference proceedings. His research interests are in the area of Software Engineering and Distributed Systems. In particular, his current research focuses on middleware technology for largely distributed, pervasive, and reconfigurable distributed applications, with a special attention to the issue of Publish/Subscribe, Complex Event Processing and Data Streaming, as the basic mechanism to develop advanced middleware services.