FlowDB: Integrating Stream Processing and Consistent State Management

Lorenzo Affetti
Politecnico di Milano, DEIB
lorenzo.affetti@polimi.it

Alessandro Margara
Politecnico di Milano, DEIB
alessandro.margara@polimi.it

Gianpaolo Cugola
Politecnico di Milano, DEIB
gianpaolo.cugola@polimi.it

ABSTRACT

Recent advances in stream processing technologies led to their adoption in many large companies, where they are becoming a core element in the data processing stack. In these settings, stream processors are often used in combination with various kinds of data management frameworks to build software architectures that combine data storage, processing, retrieval, and mining. However, the adoption of separate and heterogeneous subsystems makes these architectures overmuch complex, and this hinders the design, development, maintenance, and evolution of the overall system. We address this issue by proposing a new model that integrates data management within a distributed stream processor. The model enables individual stream processing operators to persist data and make it visible and queryable from external components. It offers flexible mechanisms to control the consistency of data, including transactional updates plus ordering and integrity constraints.

The paper contributes to the research on stream processing in various ways: we introduce a new model that has the potential to simplify complex data-intensive applications by integrating data management capabilities within a stream processing system; we define data consistency guarantees and show how they are enforced within this new model; we implement the model into the FlowDB prototype, and study its overhead with respect to a pure stream processing system using real world case studies and synthetic workloads. Finally, we further prove the benefits of the proposed model by showing that FlowDB can outperform a state-of-the-art, in-memory distributed database in data management tasks.

CCS CONCEPTS

• **Information systems → Parallel and distributed DBMSs**; **Stream management**;

ACM Reference format:
Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2017. FlowDB: Integrating Stream Processing and
Consistent State Management. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 12 pages.
DOI: 10.1145/3093742.3093929

DEBS '17, Barcelona, Spain

978-1-4503-5065-5/17/06…$15.00
DOI: 10.1145/3093742.3093929

1 INTRODUCTION

Many applicative domains require a near real-time processing of a large volume of data as it becomes available. Examples include fraud detection systems, distributed systems monitoring, social media notification services, and stock options analysis. In general, the ability to analyze *streams of data* is vital in any modern information system. Modern *Stream Processors* (SPs) address this need for high-throughput and low-latency data processing by leveraging large clusters of commodity machines to distribute the processing load, and offer fault tolerance mechanisms to quickly recover from machine failures. SPs describe data processing tasks as directed graphs, where edges represent streams of data and nodes represent 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 recent advances in stream processing technologies and their advantages in terms of performance, scalability, and fault tolerance led to their adoption in the data processing stack of many large companies. In these settings, SPs are often used in combination with data storage and analysis frameworks such as transactional and analytical databases, to build software architectures that combine data storage, processing, retrieval, and mining [25]. However, these architectures suffer from the complexity of managing two or more separate subsystems with the need of correctly merging their results, which makes it difficult to reason on the semantics of the overall system, thus hampering its design, implementation, and maintenance. Moreover, these architectures might waste resources due to unnecessary data redundancy across different subsystems.

We believe that the need to integrate SPs with other data management tools sheds light on some key limitations in the current model of SPs, and in particular on some —missing— data management capabilities. Moving from this premise, we present a novel model that augments SPs by adding classic data management concepts. The model enables the developers to make the state of operators externally visible and queryable as in traditional databases, introduces integrity constraints to validate the correctness of the data, and provides transactional semantics and optional ordering guarantees for state updates. In this way, the model avoids the need for external services for data query and retrieval, thus offering a unifying data management layer with precise semantics and consistency guarantees. More specifically, our model extends modern distributed SPs with three novel concepts: (i) state operators, which expose their internal state; (ii) transactional subgraphs, which define the portions of the processing task that require consistent state management; (iii) integrity constraints, which specify application specific correctness criteria.

We implement our model in the FlowDB prototype based on the Flink open-source SP [5] and we evaluate its performance using case studies and synthetic workloads. FlowDB does not introduce additional overhead to pure stream processing in absence of state management requirements and performs better than a state-of-the-art distributed in-memory database in some classic data management tasks.

To summarize, this paper contributes to the research on stream processing in several ways: (i) it proposes a novel model for SPs that integrates stream processing capabilities with classic data management concepts such as queryable state, integrity constraints, and transactional operations; (ii) it implements this model in the FlowDB prototype; (iii) it studies the benefits of our model using case studies and synthetic workloads, comparing FlowDB with a pure SP and a distributed in-memory database.

The remainder of this paper is organized as follows: Section 2 discusses the motivations behind our work; Section 3 describes the model we propose to enhance the state-of-the-art SPs, with particular emphasis on the consistency guarantees it proposes; Section 4 discusses the design and implementation of the prototype FlowDB system based on Flink; Section 5 empirically evaluates the performance of FlowDB in terms of maximum throughput and average latency; Section 6 reviews related work and Section 7 draws the conclusions and the future directions of this work.

2 BACKGROUND AND MOTIVATION

Modern SPs such as Storm [27], Spark [29], Flink [11], and Google Dataflow [4] organize the computation into a graph of operators. Depending on the specific system, the graph can be explicitly defined by the developer or generated from a higher level declarative or functional language. Each operator within the graph transforms elements of the input streams into elements of the output stream: for instance, a typical "count" operator receives a stream of words and continuously outputs the number of occurrences of each word. This simple example highlights how each operator can store some internal state required for the computation at hand —the current count for each word in the previous example—. In the SP programming model, the state of each operator is local, that is, it can be accessed and modified only by that operator. This approach avoids state access conflicts and enables a high degree of parallelism. In particular, SPs achieve *task parallelism* by allocating different operators on different nodes and *data parallelism* by spawning multiple instances of each operator, with each instance working on an independent *partition* of the input stream. For instance, in our word count example, words can be redirected to four different instances of the "count" operator depending on their starting letter, such that each instance stores its own local state —the number of occurrences of the words in its partition— with no need to access the state stored into the other instances.

This model strives performance —parallelism with no risk of data access conflicts— for generality and applicability. As an example, consider a simple application that processes bank transactions, as shown in Figure 1. The SP implements a single operator that stores the current balance of each bank account, and updates the accounts according to the received stream of input orders, which can be either withdrawals — `W`, deposits — `D`, or transfers — `T`. To

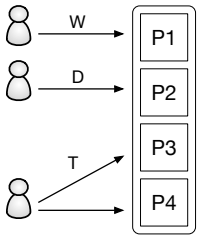


Figure 1: The graph of computation for the bank system

enable data parallelism, the SP could partition the bank accounts assigning them to four operator instances, `P1`, `P2`, `P3`, `P4`. This is fine with deposit and withdrawal operations, that involve an individual account and thus can be easily redirected to the partition containing that account, but does not match transfers, which might involve bank accounts belonging to different partitions. Moreover, depending on the application semantics, some consistency guarantees might be required to correctly implement the transfer logic: for instance, the money transfer needs to be atomic with respect to other operations and has to satisfy specific constraints, such as the availability of a minimum amount of money in the source account. Implementing these requirements represents an additional problem for SPs, since they do not allow to address state across operators or across partitions. Furthermore, SPs ensure the atomicity of execution of individual operators, but do not enable the specification of atomic operations that involve data in multiple operators or partitions. Finally, despite some initial proposals that we overview in Section 6, the internal state of operators cannot be queried and retrieved. In the bank scenario, this means that a user can only receive real-time updates from a bank account as a result of order processing, but cannot explicitly query the current balance of one or more accounts.

To overcome the limitations above, current architectures store state information —account balances— into dedicated database and data management systems, which are queryable and offer transactional updates with consistency guarantees. However, database technologies are not oriented towards distributed elaboration of large volumes of input data. This typically results in hybrid architectures that complement database systems and SP systems to exploit the best of the two worlds. However, the complexity of these architectures might lead to two types of problems. On the one hand, it may hinder the design, implementation, and maintenance of the whole system. On the other hand, it 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. To overcome these problems we propose a new model that augments the current SPs with data management features, such as transactional state updates, integrity constraints, and ordering guarantees.

3 INTEGRATING STREAM PROCESSING AND STATE MANAGEMENT

The model we propose strictly integrates the stream processing capabilities of modern distributed SPs with the data management

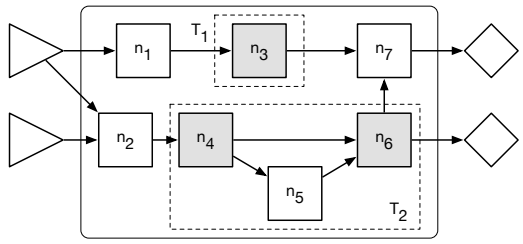


Figure 2: Model overview

and query capabilities of transactional databases. The goal is to elaborate large volumes of streaming data on the fly, while consistently updating data stores that are visible and queryable from external components. In a nutshell, we inherit the processing model of distributed SPs that organize the computation as a graph of stream transformation operators deployed on different threads and machines, and we enrich this model with an embedded data store organized into a set of `state` nodes —the analogous of database tables— that can be queried from external users and software components. State nodes can in turn be partitioned, with individual partitions possibly deployed on different physical nodes. As noted in the recent database literature, the steadily decreasing price of RAM is making disks obsolete as the primary data storage layer for operational systems [23]. Hence, our model assumes the data store embedded into the SP to reside in main memory and to be replicated to disk only for the purpose of durability.

The model simplifies the control of the order and consistency of updates to data stores, which would be difficult to obtain with separate stream processing and data management systems. We enable developers to specify integrity constraints on the content of data, and to declare transactional sub-graphs. The model ensures that the updates to state nodes that are part of a transactional subgraph occur under transactional semantics, and are invalidated in the case one or more integrity constraints are violated. By enforcing transactional semantics only within transactional sub-graphs the model strives to provide strong consistency when needed and high performance when possible.

In the remainder of the section, we present the model in detail, we introduce the concepts and terminology that we adopt in the rest of the paper, and we formalize the semantics of transactions and the order and consistency guarantees they offer.

3.1 Model overview

In line with modern distributed SPs, we define the computation as a directed graph $G = (N, E)$, where edges represent unbounded collections —*streams*— of elements, and nodes represent *operators*. Figure 2 depicts our model using squares for operators and arrows for streams. Streams originate from *sources* —triangles in Figure 2—, and terminate in sink —diamonds in Figure 2—. Both sources and sinks are outside the scope of the SP.

Operators consume and process elements from their input streams, and produce elements in their output stream as a result. For instance, a `map` operator applies a user-defined function to transform each element in the input stream into an element of the output stream. Similarly, a `filter` operator produces an output stream that retains only the elements of the input stream that satisfy a user-defined predicate. In general, operators can generate zero, one, or more output elements in response to each input element received, and might retain some internal state across the evaluation of multiple elements, as shown in Section 2.

Operators can also be partitioned, with each partition considering a portion of the input streams. We denote the k partitions of an operator n_i as $n_i^1, \ldots, n_i^k$. Each partition n_i^j processes one input element at a time, on a single processing thread. Two partitions n_i^j and n_i^l do not share state. Because of this, it is often necessary to control the mapping of input elements to partitions. For instance, if a partitioned operator is fed with words and needs to count the number of occurrences of each word, the developer needs to ensure that identical words are submitted to the same partition for the total count to be correct. To do so, developers can specify a *keyBy* function that takes an element and returns a key for that element. Elements with identical keys are guaranteed to be processed within the same partition. For instance, in the word count examples above, the *keyBy* function could partition worlds by hashing them.

We say that an element e_1 *causes* an element e_2 if e_2 is produced by a partition n_i^j as a result of processing e_1, and we write $e_1 \rightarrow e_2$. We denote as $e_1 \Rightarrow e_2$ the transitive closure of the causal relation.

We call *state* operators the subset $S \subseteq N$ of operators that embed some state and make it visible and queryable from external users and components. For example, in the bank account management system introduced in Section 2, a state operator might store the current balance of each account, which gets updated according to the streams of input operations — deposits, withdrawals, transfers. Similar to database tables, external software components and users can query the operator to access the current balance of some account. Figure 2 includes three state operators —n_3, n_4, n_6—, which are represented in grey. As any other operator, a state operator can be partitioned across multiple nodes. This enables to distribute the costs for storing, accessing, and updating data across multiple physical machines, thus making it possible to fit large datasets in main memory.

The possibility to access the state of operators opens room for consistency concerns. For instance, in our bank management system one might want to always see either a money transfer operation completed in both the source and the destination account, or in none of them. However, if the source and the destination accounts are stored in different partitions, they are updated independently, and queries might retrieve inconsistent states in which only one of the two partitions has been updated. Similar problems might appear across different state operators. For instance, consider two state operators: the first stores the current balance for each user, the second the last ten bank operations for the same users. Queries might want to see the effect of a bank operation in both states or none of them.

Our model enables developers to define *where* consistency needs to be enforced, and *which* consistency constraints are required. Specifically, we introduce *transactional subgraphs* to identify the portions of a processing graph that require consistency and *integrity constraints* to express application invariants that the update to state must preserve. A transactional subgraph $T = (N_t, E_t)$ is a connected subgraph of G such that the set of state operators

$S_t \subseteq N_t$ is updated with transactional semantics, which means that the updates satisfy atomicity, isolation, consistency, and order guarantees, as discussed in more details in Section 3.2. Figure 2 shows two transactional subgraphs, T_1, which includes node n_3 alone, i.e., all partitions of n_3; and T_2, which includes nodes n_4, n_5, and n_6. In our current implementation of the model, each transactional subgraph receives in input a single stream and produces zero, one, or more output streams. When an input element e enters the transactional subgraph T, all the changes that e determines on any state operator in T are performed atomically and in isolation —that is to say, without interleaving effects from any other element e_1—. In other words, a transactional subgraph T guarantees that all the state operators within T are updated *as if* only one element e at a time was processed within T until completion, that is, until all the elements caused by e exit T. Moreover, queries from external components cannot observe intermediate states in which the effects of an input element have been applied only to a subset of the state operators in T. In practice, as discussed in the following, we adopt finer grained concurrency control mechanisms that preserve a high degree of parallelism while offering the semantics presented above.

Our model complements transactional subgraphs, which group the state operators that need to be updated with transactional semantics, with integrity constraints, which express application specific constraints on the state that need not be violated by state updates. Integrity constraints can predicate on the state of any node within a transactional subgraph. When an element e enters a transactional subgraph T with integrity constraints C, the system tries to apply all the changes to state operators that are triggered by e or by any element caused by e. If these changes violate one of the integrity constraints in C, all those changes are discarded and the system produces a notification of integrity violation. As an example of how integrity constraints work, consider again our bank account management application. An integrity constraint might express the impossibility for an account balance to drop below zero. Any withdrawal or transfer operation that would lead to a negative balance must be discarded. In the case of a transfer operation, which involves two updates of two distinct accounts caused by the same tranfer request, *both* updates are discarded in the case of an integrity constraint violation, even if the two accounts are stored and managed in different physical nodes.

3.2 Consistency guarantees

We now discuss in more details the consistency guarantees that transactional subgraphs and integrity constraints offer. Let us consider a transactional subgraph $T = (N_t, E_t)$ including the set of state operators $S_t = (s_{t_1} \dots s_{t_n}) \subseteq N_t$ and the set of integrity constraints $C = (c_1 \dots c_m)$ that predicate on S_t. We denote as e a generic element that enters T.

$U(e, s_{t_i})$ is the set of update operations (changes) that the processing of e determines on state $s_{t_i} \in S_t$. More precisely $U(e, s_{t_i})$ is the result of processing element e or any element $e' : e \Rightarrow e'$ in the state node s_{t_i}. $U(e, T)$ is the set of all the update operations triggered by e in any state operator $s_{t_i} \in N_t$. We say that e *starts* a transaction τ in T when e enters T. We say that τ *terminates* when no more elements $e \cup e' : e \Rightarrow e'$ need to be processed in any node $n \in N_t$.

We denote as $O(T, e)$ the set of elements that are caused by e and that exit the transactional subgraph T. More formally $O(T, e) := e' : e \Rightarrow e', e' \in (i, j), i \in N_t, j \notin N_t$, meaning that the set includes any element e' caused by e and that belongs to a stream (edge of the graph) (i, j) that exits the transactional subgraph.

Finally, we represent a query Q as a set of read operations on a set of state operators S_Q.

3.2.1 Consistent state and atomicity. We say that the state of a transactional subgraph T is valid if it does not violate any integrity constraint $c_i \in C$. Given a transactional subgraph T with integrity constraints C, and an element e that starts a transaction τ on T, we say that τ is valid if the state of T is still valid after all the updates in $U(e, T)$ have been applied. Otherwise, we say that τ is not valid.

Our model ensures that only valid transactions are executed and that they are executed *atomically*, while non valid transactions are discarded. In particular, when an element e enters a transactional subgraph T, either *all* the updates $U(e, T)$ are applied if the transaction originated by e is valid, or *none* of them in the opposite case.

Atomicity extends to external queries, which cannot observe state configurations in which some updates in $U(e, T)$ have been applied and some not, but only configurations in which all the updates in $U(e, T)$ have been applied or none of them.

3.2.2 Isolation. Our model ensures isolation of transactions within a transactional subgraph T. More specifically, the model currently support *serializable* isolation [3], which ensures that the results of two transactions τ_1 and τ_2 take place as if τ_1 and τ_2 were executed in some sequential order, without any interleaving of their updates to state operations. More formally, given a transactional subgraph T and any two input elements e_1 and e_2, the result of processing e_1 and e_2 would be the same as if all the updates in $U(e_1, T)$ were applied before all the updates in $U(e_2, T)$ or all the updates in $U(e_2, T)$ were applied before all the updates in $U(e_1, T)$.

3.2.3 Order. Our model optionally enables developers to enforce that the effects of transactions are the same as if they were executed sequentially *in the same order* in which they started. More formally, given a transactional subgraph T and two input elements e_1 and e_2 such that e_1 enters T before e_2, our model ensures that the results of processing e_1 and e_2 are the same as if all the updates in $U(e_1, T)$ were applied before all the updates in $U(e_2, T)$, and that for any pair of output elements $e'_1 \in O(e_1, T)$, $e'_2 \in O(e_2, T)$ that are caused by e_1 and e_2, respectively, and that belong to the same output stream (i, j) —$e'_1 \in (i, j)$, $e'_2 \in (i, j)$— e'_1 appears on the output stream (i, j) before e'_2.

4 THE FLOWDB SYSTEM

We implemented the model presented in Section 3 in the FlowDB prototype. FlowDB is based on the open-source Flink SP [11], which it extends by introducing the concepts of state operators, transactional subgraphs, and integrity constraints, and by implementing the consistency guarantees discussed in Section 3. We first present the FlowDB API in Section 4.1 and then the FlowDB implementation in Section 4.2.

4.1 FlowDB API

FlowDB is written in Java and extends the Flink data stream API, which provides a functional interface to manipulate streams. The code in Listing 1 exemplifies the API by showing a simple topology that takes in input a stream of text lines `linesStream` and produces a stream of 2-tuples consisting of a word and the number of occurrences for that word observed so far. First, the topology splits each line into words using the `flatMap` operator that outputs multiple elements (words) for each input element (line). Then, a `map` operator converts each input word into a 2-tuple that contains the input word in the first position and the integer 1 in the second position. Finally, the `keyBy` operator groups the tuples by word and sums up the second value within each group.

Listing 1: Word count example in Flink

```
DataStream<String> linesStream = getInputStream(...);

DataStream<Tuple2<String, Integer>> counts =
    // Split line into words
    linesStream.flatMap(line -> line.split("\t"))
    // Convert each word w into a 2-tuple (w, 1)
    .map(w -> new Tuple2<>(w, 1))
    // Group by word (tuple field 0)
    .keyBy(0)
    // Sum up the count (tuple field 1)
    .sum(1);
```

Partitioning of operators is semi-automatic[1]: for instance, the `sum` operator can be partitioned and executed in parallel on different threads or nodes. In this case, the `keyBy` primitive ensures that tuples with the same word are delivered to the same partition, which retains the state necessary to make the count correct.

Listing 2: Bank transfer example in FlowDB

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

// Declare a state operator with String key and Float value
StateUtils.newStateOp("account", String.class, Float.class);

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

// Split a transfer into a withdrawal and a deposit
TransactionalDataStream<Withdrawal> wStream =
  t.map(tr -> tr.getWithdrawal());
TransactionalDataStream<Deposit> dStream =
  t.map(tr -> tr.getDeposit());

// Apply the withdrawal to the "account" state
// and close the transaction
wStream.keyBy(w -> w.getAccount())
 .state("account", (oldVal, w) -> oldVal - w.getAmount())
 .closeTransaction();

// Apply the deposit to the "account" state
// and close the transaction
dStream.keyBy(d -> d.getAccount())
 .state("account", (oldVal, d) -> oldVal + d.getAmount())
 .closeTransaction();
```

FlowDB augments the Flink API with a `state` operator and two `openTransaction` and `closeTransaction` primitives to define the boundaries of transactional subgraphs. Listing 2 shows how a developer can implement the bank transfer scenario discussed in Section 2 by taking advantage of a state operator and a transactional subgraph.

The `StateUtils.newStateOp` static method declares a new state operator named `account`. In FlowDB, each state operator is defined as a key-value store. This enables FlowDB to split the key space in partitions and to store state operators in main memory.

In Listing 2, FlowDB takes in input a stream of bank transfers `transferStream`. The `openTransaction` opens a transactional subgraph and transforms the input `DataStream` into a `TransactionalDataStream` `t`. Then, the code splits each transfer into the corresponding deposit and withdrawal, creating streams `Deposit` and `Withdrawal` from `t`, and uses the `state` operator to update the value of `account`. Let us consider the stream of deposits `dStream`: first, deposits are grouped by account —`keyBy` primitive— then the `state` operator is used to update the old value `oldVal` of the account based on the amount of the deposit `d.getAmount()`. The `state` operator returns the new value, which can be used for further downstream operations. Finally, the `closeTransaction()` operator closes the transaction. Although the example in Listing 2 defines a single state operator, the developers are free to include any number of state operators within a transactional subgraph, that is, between a `openTransaction` and a `closeTransaction` operators. The presence of a transactional subgraph ensures the consistency guarantees presented in Section 3, and in particular that the withdrawal and the deposit that generate from the same transfer are executed atomically, and that different transactions execute in isolation. Furthermore, developers can instruct FlowDB to enforce that transactions are processed in the same order in which they start by passing a parameter to the `openTransaction` operator.

Listing 3: Integrity constraints in FlowDB

```
...

// Apply the withdrawal to the "account" state
// and close the transaction
wStream.keyBy(w -> w.getAccount())
 .state("account", (oldVal, w) -> oldVal - w.getAmount(),
  (oldVal, w) -> oldVal - w.getAmount() >= 0)
 .closeTransaction();

...
```

In line with the model in Section 3, FlowDB enables developers to express integrity constraints on state[2]. Listing 3 modifies part of the transfer example in Listing 2 to include an integrity constraint to ensure that a withdrawal takes place only if it does not lead to a negative account balance. Specifically, FlowDB evaluates the predicate passed as third parameter to the `state` operator. If any of the predicates expressed within a transactional subgraph are not satisfied, then the transaction is considered invalid and all the updates to the state performed within that transaction are discarded. Developers may customize how FlowDB reacts to an invalid transaction, for instance by writing to a log or by sending a notification message through the network.

Finally, external components can submit read transactions — queries—. At the time of writing FlowDB enables the retrieval of values by key from one or more state operators. We plan to support range queries in the next releases.

4.2 FlowDB implementation

FlowDB aims to provide consistent state management with little overhead. To do so, it partitions state operators by key and

[1]The developers can provide hints for the desired degree of parallelism.

[2]The current implementation supports predicates over individual state operators, but we plan to support predicates that combine multiple state operators in the next releases.

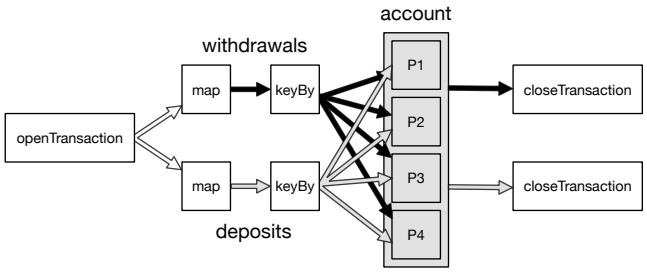


Figure 3: Topology of the bank transfer example

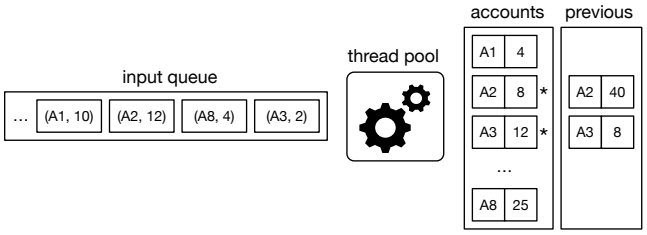


Figure 4: Management of state within a single partition

deploys each partition independently. For instance, the `account` state in Listing 2 can be partitioned by account, such that different accounts might be stored on different nodes. When `openTransaction` is invoked on a `DataStream`, FlowDB creates a `TransactionalDataStream` that wraps every element traversing the stream with metadata used to enforce transactional semantics. Specifically, each element entering the transactional subgraph is annotated with a `transactionID` that uniquely identifies a transaction and with a `validity` field that indicates whether any integrity constraint has been violated.

4.2.1 Single state operator. Let us consider again the bank account application. Figure 3 shows the topology that FlowDB instantiates from the code in Listing 2. First, the `TransactionalDataStream` that exits the `openTransaction` operator —white arrow in Figure 3— is used in parallel by two `map` operators to create the stream of withdrawals —black arrows in Figure 3— and the stream of deposits —grey arrows in Figure 3—. Each of these two streams enters a `keyBy` operator that delivers the elements to the correct partition of the `account` state operator based on the account they refer to. As an example, Figure 3 shows the `account` state split into four partitions `P1`, `P2`, `P3`, `P4`. Finally, both streams enter a `closeTransaction` operator.

Each element e in the `TransactionalDataStream` that exits the `openTransaction` operator receives a different `transactionID` and has `validity=true`. Each element e' produced by a non-state operator —e.g., the `map` operator— as a result of processing an element e is assigned with the same `transactionID` and `validity` as e. Upon receiving an element e, a state operator updates its state and produces an output element e' with the same `transactionID` as e and a `validity` that is `true` if no integrity constraints have been violated and `false` otherwise. The `closeTransaction` operator computes the overall validity for the transaction based on the `validity` value of all the elements with the same `transactionID` (those caused by the element starting the transaction), and communicates it back to the state operator, which discards or confirms the changes performed within the transaction depending on the validity value it receives. In case of `false` validity, the `closeTransaction` operator discards the elements produced during the transaction.

Notice that in some cases the elements part of the same transaction (those caused by the element starting the transaction) reach more than one `closeTransaction` operator. This is the case in Figure 3. When this happens, FlowDB automatically and transparently instantiates an additional `coordinator` operator that collects the (partial) validity results of transactions from every `closeTransaction` operator, combines them and becomes responsible for communicating the results back to the state operators part of the transaction.

We now discuss how each partition within a state operator is implemented. Figure 4 shows the internal components of a partition in the case of the `account` state operator used in our example. For each input stream (e.g., the stream of deposits in our example), the partition stores an `input queue` of elements (e.g., pairs of account id and deposit amount in our example). Several threads from a `thread pool` read in parallel from the `input queue` and update the state accordingly. To ensure isolation, FlowDB uses a locking mechanism that works at the granularity of the single $\langle key, value \rangle$ pair stored by the state operator (individual accounts and their balance in our example). A change to an account is not allowed until all previous changes have successfully completed or have been discarded. Figure 4 indicates locked elements with a star. Each locked account has an associated entry in table `previous`, which stores the value of that account prior to the execution of the last change, and is used to discard the change in the case of an invalid transaction.

Each thread from the `thread pool` reads from the `input queue` in FIFO order, postponing the processing of elements that refer to locked state until they get unlocked. For instance, in the situation depicted in Figure 4 a thread would skip the first element because it refers to the locked account `A3`, and would start processing the next element, which refers to the non-locked account `A8`.

Elements get unlocked upon receiving a notification that a transaction has terminated —either successfully or violating some integrity constraint— from a `closeTransaction` operator or from a `coordinator`. The communication of notifications is managed using sockets, with each partition of a state operator accepting connections from `closeTransaction` operators and `coordinators` on a defined port. To avoid the overhead of continuously opening new connections, FlowDB caches existing socket connections and reuses them for future communication.

4.2.2 Enforcing ordering guarantees. The mechanism discussed above ensures isolation of transactions through locking of resources. Furthermore, it ensures consistency and atomicity by collecting the validity of individual state updates and by accepting or discarding all of them. However, if two elements e_1 and e_2 enter a state operator and produce the output elements e'_1 and e'_2 respectively, it is not guaranteed that if e_1 precedes e_2 in the input stream e'_1 will precedes e'_2 in the output stream. Indeed, the two elements might

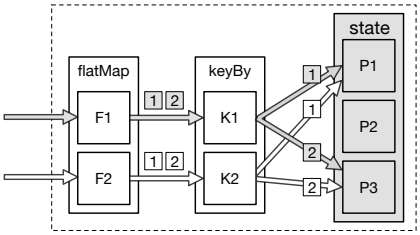


Figure 5: Chains of operators and isolation

be processed in parallel in different state partitions or in different threads within the same partition.

Developers may ask (by passing an ad-hoc parameter to the `openTransaction`) for the order of the results in the output stream to match the order of the elements in the input stream. In this case, FlowDB automatically and transparently annotates elements with sequence numbers and instantiates a `scheduler` component after the state operator. The `scheduler` collects elements from all the partitions and reorders them.

4.2.3 Chaining multiple operators. Chaining multiple operators within a transaction might break isolation. Consider the situation in Figure 5: two transactions τ_1 and τ_2 follow the flows represented by grey arrows and white arrows, respectively. The `flatMap` operator —partitioned across two nodes `F1` and `F2`— produces two output elements for each input element. We represent them with numbered squares. The subsequent `keyBy` operator —also partitioned across two nodes `K1` and `K2`— forwards these elements to the correct partitions within the following state operator `state`. Now suppose that the two elements in transaction τ_1 update element el_1 in partition `P1` and element el_3 in partition `P3` and that the same happens for the two elements in transaction τ_2. In this situation it may happen that partition `P1` processes the element coming from τ_1 before the element coming from transaction τ_2 while partition `P3` processes the element in τ_2 before the element in τ_1, thus breaking isolation. In general, this issue occurs every time the results generated by the elements in a single input stream are redistributed to different output streams, and in the presence of a `keyBy` operator.

FlowDB solves these isolation problems by (i) automatically detecting the points in a transactional subgraph where input elements are re-partitioned, and by (ii) automatically introducing a `scheduler` that reorders the elements based on their `transactionID`, as discussed in the previous section.

4.2.4 Processing queries. Queries that involve the state accumulated into a transactional subgraph T can be considered as special read-only transactions. FlowDB exploits this analogy and implements queries as special elements that traverse T and collect the values of interest from the state operators in T. These elements are submitted by a `query manager` component that is also responsible for collecting them once they have retrieved the values of interest. The `query manager` processes each query q and extracts the set of transactional subgraphs it needs to traverse. For each transactional subgraph T, it creates an element $el_{q,T}$ that it submits within the input flow of T. The element $el_{q,T}$ traverses T, collects the values of interest, and reaches a `closeTransaction` operator, which delivers it back to the `query manager`. Since queries are treated as any other transaction, they are ordered to avoid isolation problems in the case of multiple state operators in the transactional subgraph. As an example, let us consider again the bank account example in Figure 3 and assume that a user submits a query to retrieve the balance for a given account. The `query manager` receives the query, analyzes it, and discovers that the query is interested in a value within the `account` state operator. Thus, the `query manager` submits a query element to the `openTransaction` operator that opens the transactional subgraph including the `account` state. The query element retrieves the desired value and reaches the `closeTransaction` operator, that delivers the value back to the `query manager`.

Within a state operator, query elements are treated as any other element: they are stored in the `input queue` and wait until the resource they want to retrieve becomes available — not locked by an ongoing transaction. Differently from other elements, query elements access a state value with non-exclusive lock, thus ensuring that multiple queries for the same value can run in parallel, as far as no write transaction is currently trying to modify that value.

5 EVALUATION

The primary goal of FlowDB is to reduce the complexity of modern data processing architectures. To achieve this goal and be useful in practice, FlowDB has to provide an adequate level of performance in terms of the volume and velocity of data it can handle. The current FlowDB prototype builds on top of Flink version 1.1.2, a mature open-source platform well known for its performance [11]. In absence of state operators and transactional subgraph, FlowDB does not add any overhead to Flink and thus it offers the same level of performance for pure stream processing tasks. Hence, we are interested in assessing the behavior of FlowDB in the presence of state operators and transactions. To do so, we design an experimental campaign with two main goals: (i) we want to study the absolute performance of FlowDB and compare it with a state-of-the-art solution for data management in distributed environments; (ii) we want to investigate which parameters affect the performance of FlowDB.

5.1 Experiment setup

We deploy FlowDB on a cluster of 20 Amazon EC2 t2 XL instances, each equipped with 4 CPU cores and 16 GB of RAM. As a default case study, we consider the bank account management application presented in Section 4. In this application, FlowDB receives a stream of input bank transfers that it splits into two streams of deposits and withdrawals. Deposits and withdrawals that belong to the same original bank transfer are processed within a single transaction. Since the time to process an element depends on the number of other elements that try to access the same state resources concurrently, we decided to assess the performance of FlowDB in two situations: (i) we measure the overall time required to process 200k bank transfer transactions when they are all submitted in a single burst. In this case, the underlying Flink engine employs a *back-pressure* mechanism that automatically adjusts the input rate at the source based on the current load in the processing network. Thus, dividing the overall processing time by the number of input transactions processed gives us a precise estimate of the *maximum input throughput* that FlowDB can sustain; (ii) we measure the latency for

	Avg latency	Throughput
FlowDB	8.2 ms	6235 tr/s
Flink	3.1 ms	68705 tr/s
VoltDB	5092 ms	589 tr/s

Table 1: Default case study: average latency and maximum throughput for Flink, FlowDB, and VoltDB

processing individual bank transfer transactions when transactions are submitted at 80% of the maximum input throughput.

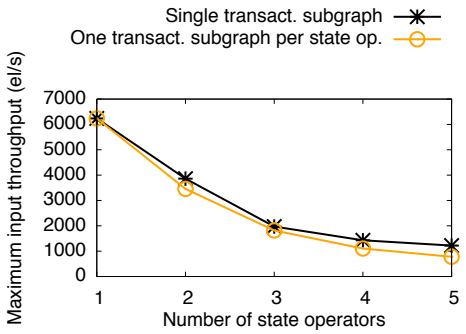
In our default scenario, the state operator includes 100k different bank accounts split in 8 partitions. The origin and destination accounts for each bank transfers are selected randomly with a uniform distribution. In all our experiments, we submit 20k input elements before start measuring, to make sure that the system is in a steady state. In the presence of a transactional subgraph, we configure Flink to buffer elements for only 1 ms before transmitting them to the next operator. Indeed, the default batching timeout of Flink of 100 ms would delay the validation of transactions from a `coordinator` and thus lock resources for longer than required.

5.2 Default case study

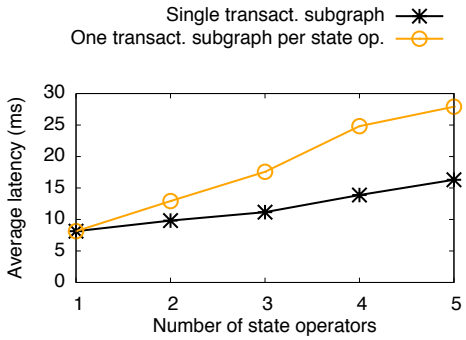
Using the default scenario described above, we measure the maximum throughput and the latency of FlowDB and we compare them with the Flink SP version 1.1.2 —the same we adopt to implement FlowDB— and the VoltDB in-memory distributed database version 7.0, which is well known for its excellent level of performance. In VoltDB, we store the accounts in a partitioned table that spans 8 different machines[3] and we implement the transfer transaction as a stored procedure that gets analyzed and precompiled at deployment time thus eliminating all of the query plan processing overhead during runtime execution[4].

Table 1 shows the results we measured. FlowDB achieves a maximum throughput of more than 6200 input elements/s with an average delay of 8.2 ms. By comparison Flink processes 68705 input elements/s with an average latency of 3.1 ms. However, Flink does *not* implement transactional semantics with the impossibility of checking the validity constraint we use for our example. Hence, Flink does not need to lock state elements and verify the validity of the update operations. Thus, this test measures the overhead of FlowDB in enforcing transactional semantics. The difference in throughput can be explained by remembering that Flink can process deposits and withdrawals in parallel, in any order, while the transactional semantics of FlowDB introduces the cost of isolation and atomicity — locking and validity check. Because of this, Flink also requires fewer operators to implement the same scenario —for example, it does not need a `coordinator` to compute the validity of transactions— that justifies the reduced processing latency.

VoltDB achieves a throughput of 589 input elements/s with an average latency of 5092 ms. This result is surprising and indicates that write transactions are very expensive for VoltDB. To prove this observation, we repeated the experiment by only submitting read transactions — queries to the value of individual accounts. In this scenario the performance of VoltDB increases significantly, leading to a maximum throughput of 236186 elements/s with an average latency of 7.48 ms. By comparison, in the case of only read transactions, FlowDB achieves a maximum throughput of 11800 elements/s and an average latency of 3.9 ms (with the same 8 nodes used by VoltDB). This result shows that our model is promising: it outperforms a state-of-the-art representative of in-memory database systems in terms of transactional updates, and it offers consistent levels of performance across read and write operations.



(a) Maximum throughput



(b) Average latency

Figure 6: Chain of dependent updates performed in series

5.3 Microbenchmarking

We now present a number of experiments that investigate which aspects have the highest impact on the performance of FlowDB. In doing so, we shed light on interesting characteristics of our model and of the underlying Flink platform.

5.3.1 Chain of dependent updates. As a first experiment, we study how FlowDB behaves when considering a chain of updates performed in series, one after the other, on different state operators. This scenario mimics the case in which the data produced by a state update is elaborated downstream and produces further updates to other state operators. We consider both the case in which all the updates occur within a single transactional subgraph and the case in which the updates occur in separate transactional subgraphs.

Figure 6a shows the maximum throughput we measure while increasing the number of state operators in the topology. With a

[3]This is equivalent to the number of state partitions we have in FlowDB. However, in this setting, VoltDB automatically creates 64 partitions.
[4]https://www.voltdb.com/blog/programming-voltdb-easy-flexible-and-ultra-fast

single state operator, FlowDB achieves a maximum input throughput of more than 6200 elements/s, which decreases with the number of state operators but remains close to 1000 elements/s even when considering 5 state operators.

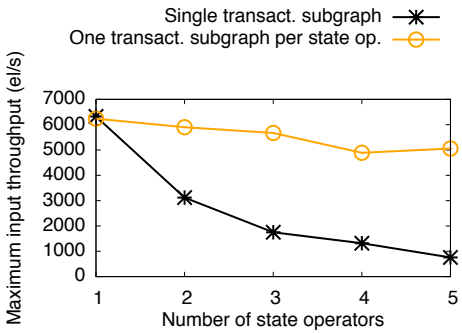
In the case of a single transactional subgraph, FlowDB needs to ensure that all the state operators receive and process the transactions in the same order. To do so, it introduces a `scheduler` between any two state operators, which reorders the incoming elements. Since we measure the maximum throughput by submitting input elements at the maximum accepted rate, all transactions become almost immediately available to the first state operator, which processes them in parallel. As a consequence, transactions complete out of order and accumulate in the subsequent `scheduler` before being submitted to the next state operator. Conversely, reordering of elements is not necessary in the case of different transactional subgraphs, since each transactional subgraph can safely process the elements in a different order. Given how similar are the performance we measure in the two cases we may deduce that implementing isolation by introducing total order through a `scheduler` does not represent a bottleneck, even if many transactions accumulate in this component. The use of one transactional subgraph for each state operator appears to be slightly more expensive, leading to a lower maximum throughput. This can be explained with the increased complexity of the topology required to open and close a higher number of transactions.

Figure 6b shows the average latency per transaction when the input rate is 80% of the maximum input throughput. With a single state operator, the average latency is well below 10 ms. Any additional operator slightly increases the latency due to the buffering performed by the Flink platform. In the case all the operators are in a single transactional subgraph, the overall latency remains below 20 ms even when considering 5 state operators. In the case each state operator is part of a different transactional subgraph, the latency increases up to more than 25 ms, due to the higher number of operators used to implement each transactional subgraph.

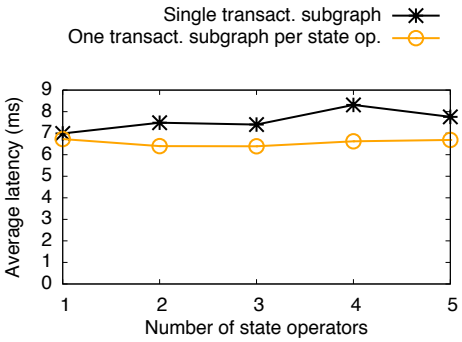
The results above let us conclude that in FlowDB the cost for ensuring isolation through a `scheduler` is negligible with respect to the cost of opening transactions, locking resources, and establishing the validity of a transaction. We will further study the cost of locking in the following sections.

5.3.2 Independent updates. As a second experiment, we study how the performance of FlowDB changes when considering independent state updates that occur in parallel. As in the previous section, we consider both the case in which all state updates occur within a single transactional subgraph and the case in which state operators belong to different transactional subgraphs. In the first case, we duplicate the input stream across all the state operators in the transactional subgraph. In the second case, we evenly distribute input elements across the parallel transactional subgraphs.

In the case of a single transactional subgraph, the maximum throughput —Figure 7a— decreases with the number of state operators. This is due to the increased volume of input data and to the need for collecting results from all the state operators to determine the overall validity of a transaction. Conversely, in the case of multiple transactional subgraphs, the maximum throughput remains almost constant, due to the capability of FlowDB to process transactions entirely in parallel. The limited decrement of throughput with 4 and 5 state operators can be explained with the higher number of operators in the topology, which overcomes the number of available cores in our testbed.



(a) Maximum throughput



(b) Average latency

Figure 7: Independent updates performed in parallel

The same reasons motivate the behavior of the latency — Figure 7b. With multiple transactional subgraphs the latency remains constant, while it slightly increases with a single transactional subgraph. Most significantly, in absolute terms the overall latency remains well below 10 ms even when increasing the number of state operators. These results further demonstrate the applicability of FlowDB to transactional tasks and prove that transactions do not significantly affect processing tasks that occur in parallel.

5.3.3 Number of unique state keys. We now study how the presence of conflicts during the update of state operators impacts on the performance of FlowDB. To do so, we consider again our default bank use case and we change the number of unique keys —accounts— within the state operator. Figure 8 shows the results we measured. As expected, the performance increases with the number of unique keys: indeed, with a uniform selection of the state elements to consider within each transaction, a larger set of elements reduces the possibility for conflicts. Interestingly, with only 1000 keys —a very small number that may introduce many state conflicts—, FlowDB manages around 6000 elements/s. Furthermore, even in the extreme case of only 10 keys, FlowDB processes close to 500 elements/s. We do not report any measure of latency

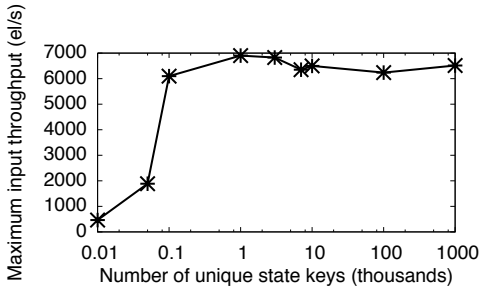


Figure 8: Number of unique state keys

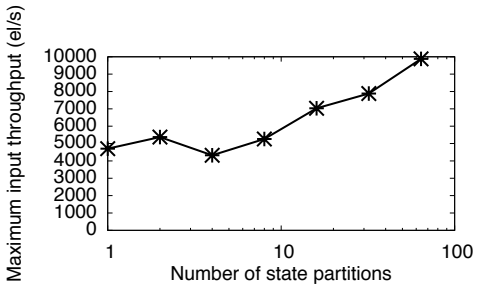


Figure 9: Scalability with the number of state partitioning

for this experiment, since it never changes. Indeed, as discussed in the previous sections, we measure the latency when submitting input elements at 80% of the maximum input throughput. Under these circumstances, the system is not overloaded and the latency only depends on the topology, which is fixed in this experiment.

5.3.4 Scalability. This section investigates how FlowDB scales when increasing the number of state partitions, which enable for a higher degree of parallelism when processing transactions. Figure 9 shows the results we measure in our default use case when changing the number of partitions from 1 to 64. As Figure 9 shows, the number of partitions has a small impact on the maximum throughput of the system when moving from 1 to 8 partitions. With more than 8 partitions, the throughput starts increasing linearly with the number of partitions, reaching almost 10k elements/s with 64 partitions. As in the previous experiment, the overall latency at 80% of the maximum input throughput remains constant —around 8 ms— and is not shown in figure.

5.3.5 Cost of queries. The possibility to access and query the state of operators from external components is a key feature in FlowDB. We now study how queries impact on the performance of FlowDB by measuring the latency for both state updates and query answering when changing the frequency of query arrival. We consider again our default bank management use case, and we reduce the number of distinct keys to 50 to better stress FlowDB when submitting queries. Indeed, as discussed in Section 4, queries lock state resources while reading and thus block incoming state updates of those resources. We submit bank transfers between two accounts as in all the previous experiment at a fixed rate of 4500

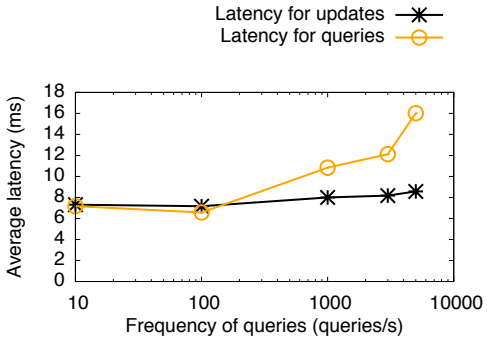


Figure 10: Cost of queries

transactions/s. Each query accesses the value of 5 different accounts. We change the rate of queries from 10 to 5000 queries/s and we measure the latency for processing bank transfer and queries.

Figure 10 shows the results we measure. We observe that the latency for completing an update remains almost constant and below 10 ms. Conversely, the latency for answering queries remains constant up to 100 queries/s and then slightly increases. Indeed, queries need to lock 5 accounts —10% of all the accounts available in the system— to be successfully processed, and they conflict with concurrent updates as well as with other queries. However, although increasing, the query latency remains well below 20 ms even in the case of 5000 queries/s. This demonstrates the applicability of our approach to transactional queries.

6 RELATED WORK

Our work addresses the requirements of modern data-intensive applications that need to integrate consistent state management and low-latency processing of large streams of information. Because of this inter-disciplinary nature, the work is related to several fields, including stream processing, database management, and consistency in distributed systems.

6.1 Processing streams of data

The last decade saw an increasing interest in technologies to process streams of data, and several systems have been proposed both from the academia and from the industry. We distinguish two generations of SPs. The first generation flourished in the mid 2000s and focuses on the definition of abstractions to query streams of data as in Data Stream Management Systems (DSMSs), or to detect situations of interest from streams of low-level information, as in Complex Event Processing (CEP) systems. The interested reader can refer to the detailed survey of these systems by Cugola and Margara [15].

DSMSs usually rely on declarative query languages derived from SQL, which specify how incoming data have to be selected, aggregated, joined together, and modified, to produce one or more output streams [7]. The reference model for DSMSs is defined in the seminal work on the Continuous Query Language (CQL) [6]. In CQL, the processing of streams is split into three steps: first, *stream-to-relation* operators —windows— select a portion of each stream to implicitly create static database tables. The actual computation takes place on these tables, using *relation-to-relation* (mostly

SQL) operators. Finally, *relation-to-stream* operators generate new streams from tables, after data manipulation. Several variants and extensions have been proposed, but DSMSs mostly rely on the general processing abstractions defined above. Thanks to the similarity of their languages and processing models to SQL, most DSMSs seamlessly interact with (relational) database systems to manage static data. However, to the best of our knowledge, DSMSs typically rely on the transactional mechanisms offered by the static database for data consistency, and none of them address the problem of distributed data management.

The Aurora/Borealis DSMS first introduced the idea of defining the processing in terms of a directed graph of operators [2] and to deploy the operators on different physical nodes [1]. This approach deeply influenced the second generation of SPs that we overview below and that are more closely related to the subject of this paper.

CEP systems were developed in parallel to DSMSs and represent a different approach towards the analysis of streaming data, which targets the detection of situations of interest from patterns of primitive events [18, 22]. CEP systems consider the elements of a stream as notifications of event occurrences and express patterns using constraints on the content and time of occurrence of events [10, 14].

The Event Processing Network (EPN) formalism introducted by Etzion and Niblett [18] represents event processing as a directed graph of operators, and thus it shares many similarities to the processing model considered in this paper.

The second generation of SPs has its roots in the research on Big Data and comprises systems designed 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 [16] to automate the distribution of processing. Subsequent proposals increased the expressivity of MapReduce, enabling the developers to specify arbitrarily complex directed graphs of operators [29]. 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 [28].

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 [30], provide streaming computations on top of batch processing abstractions by splitting each stream into small static chunks (micro-batches). However, this approach introduces some latency (typically, in the order of seconds), since the SP needs to accumulate data into micro-batches before starting the processing task.

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 [27], Heron [20], and Google DataFlow [4], to name a few. This is also the model used in Flink [11], that we adopt and extend in this paper.

Interestingly, a recent research proposal introduces the idea of making the state of operators explicit to simplify the implementation of iterative machine learning algorithms [19]. However, the proposed model only considers operations on state that are entirely performed on a single machine, either because the machine stores a single partition of a data structure or because it stores a replica of the entire data structure, which is required to fit in main memory.

Finally, reactive update of stateful variables have been widely studied by the programming languages community in the domain of *reactive programming* [8]. Reactive programming (RP) shares many similarities with stream processing, and grounds on abstractions to define dependencies between time changing values and automated propagation of changes. 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 [17, 24].

6.2 Distributed databases

Distributed relational database systems ensure ACID properties through locking or optimistic concurrency control [21]. NoSQL database often trade consistency for performance: for instance the document-based database MongoDB only supports transactions that involve a single document [9], and the Redis key-value store does not fully support arbitrary distributed transactions [12].

So called NewSQL database systems aim to reconcile the two worlds offering strong consistency and efficient distributed data management. H-Store [26] is an in-memory database that enforces atomicity of transactions on individual sites through single threaded computations, and schedules multi-site operations to ensure ACID properties. Furthermore, H-Store supports reactive behaviors through deterministic and parametric stored procedures. S-Store extends H-Store to deal with streaming workloads [13]. S-Store defines the stream processing capabilities on top of an OLTP system (H-Store), implementing streams as time-varying tables and stream processing as triggers. This approach later evolved in the VoltDB database system that we used in our evaluation [23].

6.3 Big Data architectures

To solve the dichotomy between consistent state management and low-latency stream processing, some data processing architectures have been proposed in the last few years and have been increasingly adopted in the data processing stack of several companies. Nathan Marz first proposed the Lambda Architecture to meet the need for low-latency results, while providing exact, reliable, yet "old" results in case of failures [25]. The Lambda architecture 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. The Lambda architecture couples this speed layer 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 decides whether to serve the results of the speed layer —recent, but possibly inaccurate— or those of the batch layer —accurate, but possibly outdated—.

More recent proposals are questioning this type of architecture because of its complexity and high maintenance costs, and foster the development of stream-only architectures, where the SP plays a more central role[5]. Some of these proposals —including Flink version 1.2[6]— also introduce the concept of queryable state, although

[5] http://milinda.pathirage.org/kappa-architecture.com/
[6] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/stream/queryable_state.html

focusing on the state of individual operators. Our proposal can be considered as an evolution of these architectures that embeds state within the SP and enables transactional updates that involve operations on multiple operators.

7 CONCLUSIONS

Modern companies often rely on data processing architectures that integrate stream processing and data management systems. These architectures are often complex and make it difficult to reason on the correctness and consistency of the data they produce.

To solve this problem, we propose an innovative model that tightly integrates state management within a distributed SP system. The model extends SPs with three key novel concepts: transactional sub-graphs, integrity constraints, and consistency guarantees.

The paper presents the model and its implementation in the FlowDB prototype system. We evaluate the performance of FlowDB using synthetic benchmarks and a realistic case study. The promising results we measure let us believe that our model has the potential to simplify the design, implementation, and validation of large-scale data-intensive information systems while providing an adequate level of performance.

As future work we plan to refine our implementation and test it with multiple workloads from real-world projects. Specifically, we plan to study fault tolerance in the presence of transactions, and to include additional levels of consistency, enabling developers to choose the trade-off between consistency and performance that better suite their needs. We will also investigate alternative protocols to ensure transactional semantics, including optimistic protocols for isolation.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The design of the borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '05)*. Asilomar, CA, 277–289.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (2003), 120–139.

[3] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of the International Conference on Data Engineering (ICDE '00)*. IEEE, 67–78.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *The VLDB Journal* 8, 12 (2015), 1792–1803.

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (2014), 939–964.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006), 121–142.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS '02)*. ACM, New York, NY, USA, 1–16.

[8] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *Comput. Surveys* 45, 4 (2013), 52:1–52:34.

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

[10] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. 2007. Cayuga: A High-performance Event Processing Engine. In *Proceedings of the International Conference on Management of Data (SIGMOD'07)*. ACM, New York, NY, USA, 1100–1102.

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

[12] Josiah L. Carlson. 2013. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA.

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

[14] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS'10)*. ACM, New York, NY, USA, 50–61.

[15] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *Comput. Surveys* 44, 3, Article 15 (2012), 15:1–15:62 pages.

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

[17] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 361–376.

[18] Opher Etzion and Peter Niblett. 2010. *Event Processing in Action*. Manning Publications, Greenwich, CT, USA.

[19] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 49–60.

[20] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 239–250.

[21] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226.

[22] David C. Luckham. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston, MA, USA.

[23] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory oltp recovery. In *Proceedings of the International Conference on Data Engineering (ICDE 2014)*. IEEE, 604–615.

[24] Alessandro Margara and Guido Salvaneschi. 2014. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 142–153.

[25] Nathan Marz and James Warren. 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., Greenwich, CT, USA.

[26] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era (It's time for a complete rewrite). In *Proceedings of VLDB (VLDB '07)*. VLDB Endowment, 1150–1160.

[27] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156.

[28] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2.

[29] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10.

[30] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438.