Consistent Stream Processing

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

ABSTRACT

Stream Processors (SPs) continuously transform huge volumes of input streams with a computational model that is inherently distributed, scalable, and fault-tolerant. For these reasons they are used in application environments in which almost real-time computation is of paramount importance, such as stock option analysis, fraud detection systems, monitoring, and real-time data analytics for web applications. In many applicative domains, SPs are used in conjunction with data management systems such as transactional databases and data warehouses that store intermediate or final results produced by the SPs. However, SPs have no control on the consistency guarantees of the results produced on external components. We propose a novel approach that we name *consistent stream processing* that integrates the external state of databases within the SP and enforces consistency guarantees both on state updates and on external querying. We extend the computational model of SPs with transactions and we provide two possible strategies to enforce their transactional properties.

ACM Reference format:
Lorenzo Affetti. 2017. Consistent Stream Processing. In *Proceedings of ACM International Conference on Distributed Event-Based Systems, Barcelona, Spain, June 2017 (DEBS '17),* 4 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Stream processing applications range from fraud detection systems and stock option trend analysis to real-time system monitoring and big data analytics. The emerging need to capture and analyze information as it comes is leading to a blossom of Stream Processors (SPs) that cope with the speed and the volume of data. SPs like Storm [16], Spark [17] and Flink [9] offer a distributed and fault-tolerant runtime to process information. Distribution is fundamental to scale on huge amounts of data, while fault-tolerance is necessary in large-scale computing environments where processes and hardware can frequently fail.

SPs transform input streams into output streams and update the state of external components, such as distributed filesystems, databases, and data warehouses. However, they cannot enforce consistency guarantees on the side effects induced on this external state. Various architectures, pioneered by the famous Lambda architecture [13], propose the integration of stream processing and data management capabilities. However, to the best of our knowledge, they do not study in depth the problem of data consistency.

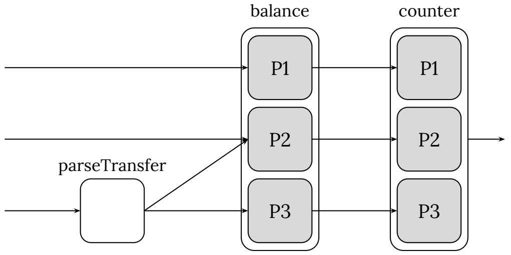

Figure 1: The graph of computation for the bank system

We contribute to this field of research by proposing a novel model that we name *consistent stream processing*, which embeds the concept of *state* within the SP, makes the state queryable, and enforces consistency guarantees on state updates and external queries.

The remainder of this paper is organized as follows: Section 2 overviews the SP model, highlighting its inconsistencies; Section 3 introduces our model for consistent stream processing; Section 4 presents the strategies to achieve consistency; Section 5 provides the preliminary results we obtained from an implementation of the proposed strategies; Section 6 provides further information about the related work; Section 7 concludes the paper providing the future directions of our work.

2 PROBLEM STATEMENT

SPs represent streaming computations as directed graphs where edges represent the streams in which elements flow while vertices represent the operators that, given one or more ingoing elements, produce zero or more ones. The graph of computation enables task-parallelism since each operator is independent from the others. It enables data-parallelism through partitioning, meaning that the SP runs many instances of the same operator, each of them responsible for a subset of the input elements. Input elements are mapped to partitions based on a *key* attribute, defined by the developer.

Take as an example a bank system. Figure 1 represents the graph that computes the balance of users. Users are the sources of the bank transactions and generate input elements that represent deposits and witdrawals in the form ($user$, $amount$) (be the amount positive or negative) or transfers in the form ($user1$, $user2$, $amount$). Deposits and withdrawals are directly sent to the `balance` operator, while tranfers pass through the `parseTransfer` operator that splits each transfer into a deposit and a withdrawal. The `balance` operator adds the amount of the bank transaction to the balance of the user. `balance` is a *stateful* operator since it keeps a state for the balances of the users.

Let us assume that withdrawals that make an account fall below zero are not allowed. The `balance` operator checks if the constraint is respected and updates the internal state accordingly. The `counter` operator keeps an internal state (partitioned by user) and counts

DEBS '17, Barcelona, Spain
2017. 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

the number of transactions that every user performs every minute. If this number exceeds 5, `counter` blocks further transactions to prevent possible frauds.

At a first sight the graph seems correct, but it hides some problems. First, once `counter` detects that a fraud is happening and aborts operations, the internal state of `balance` is not updated accordingly. Second, if the withdrawal associated to a transfer is invalidated by `balance`, also its deposit counterpart should be invalidated.

For the first problem, it could be possible to swap the order of the `balance` operator and the `counter` operator and prevent that fraudulent transactions update the state of `balance`. However the problem has only been overturned: if a transaction fails because of overdraft, it should not be counted by the `counter`. The second problem actually would not happen if `balance` was not partitioned. Without partitioning, the `balance` operator can access the balances of both users and check both of them. However this solution will lead to the absence of data-parallelism. We could think of introducing some sort of *inter-partition communication* to keep the `balance` operator partitioned while preserving consistent transfers. Actually, this is part of the solution that we describe in Section 3.

3 CONSISTENT STREAM PROCESSING

To overcome the problems presented in Section 2, we propose a novel *consistent stream processing* model that moves the state stored in external components into the stateful operators of a graph of computation; it exposes the internal state by making it *queryable*; and guarantees both consistent state updates and reads.

A *state operator* is a partitioned stateful operator that manages a subset of the resources of the overall state and optionally enforces an integrity constraint check on them. Every resource is identified by the same key attribute defined for partitioning over the input elements. Every instance of the state operator processes all of the relevant resource updates for its portion of state. External queries must specify the key of a resource to get its value. Every query can consistently read many keys from many different operators.

Our model brings the concept of database transactions in the context of a stream processing graph. A transaction is a set of operations that are executed atomically —as if they were one—: either the transaction *commits* (it succeeds) or it *aborts* (it fails). In general, every transaction execution can be interleaved with other ones producing different anomalies on their results [7]. Ensuring an isolation level among transactions prevents some or all of the anomalies that can happen during their execution. For the purpose of this paper we are interested in the *serializable* level of isolation defined by ANSI SQL-92 standard; i.e. the results of transactions are the same *as if* they were executed serially.

We build our model for transactions on the computational graph introduced in Section 2. Given a directed graph of computation $G = (V, E)$ with vertices V and edges E where each edge $e \in E$ is a pair (i, j) such that $i \in V$ and $j \in V$. A *transactional graph* $TG = (TV, TE)$ is a sub-graph of G where $TV \subseteq V$ and $TE \subseteq E$ such that $\forall(i, j) \in TE$ it holds that $i, j \in TV$. Every transactional graph must contain at least one state operator. In the case of Figure 1 the whole graph of computation is a transactional graph, while `balance` and `counter` are the state operators partitioned by *user*.

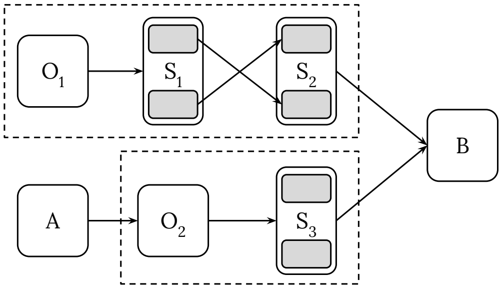

Figure 2: An example of a graph with computation with transactional graphs

The integrity constraint on `balance` is "no balance can drop below zero"; for `counter` it is "no user can perform more than 5 operations per minute".

We suppose that every transactional graph has a single entry point —denoted as o— that assigns a unique identifier t to the elements, and it has zero, one or more exit points denoted as the set E. Every element that enters operator o starts a new transaction in the transactional graph. We denote elements with the notation e^t_{vi}, meaning that e is the i-th element generated by operator $v \in TV$ in transaction t. Given a transaction t, element e^t_{ai} *generates* e^t_{bj}, if $\exists i, j, (a, b) \in TE$ such that e^t_{bj} is the j-th element produced by operator b while transforming element e^t_{ai}. The generation set O^t_b is composed of every elements e^t_{bi} generated by operator b in transaction t. The set T_t is the union of every $O^t_v, v \in TV$.

If any e^t_{vi}, violates the integrity constraint of at least one of the state operators in TV, transaction t aborts and every resource affected by the processing of every $e^t_{vi} \in T_t$ is rolled back to its previous value. Otherwise, t commits and the updates are applied.

Figure 2 shows a computation graph: transactional graphs are surrounded by dotted rectangles, partitioned state operators contain grey boxes that represent that the same operator runs on many instances. Note that the single instances of S_1 are directly connected with the ones of S_2 to show that partitions can generate elements independently.

Two transactions T_i and T_j share resource x if they both update x during their execution. In this case, T_i and T_j are in conflict. We denote the conflict set C of two transactions as the set of all shared resources between them. In order to provide the serializable level of isolation we must ensure that either T_i updates all of the resources $x \in C$ before T_j or after.

In Section 4 we propose two strategies to achieve atomicity and isolation. In Section 5 we discute the preliminary results we achieved with the implementation of these strategies on top of an open-source SP.

4 ALGORITHMS FOR CONSISTENCY

Concurrency control mechanisms achieve atomicity and isolation in the execution of transactions. Pessimistic strategies avoid conflicts by locking resources, while optimistic strategies let conflicts happen and abort and replay transactions if so [8]. Next we discuss a pessimistic and an optimistic strategy to enforce transactional guarantees in our consistent stream processing model.

4.1 The Pessimistic Strategy

In the pessimistic strategy, we ensure atomicity with a locking mechanism: every transaction must acquire a *lock* on a resource before reading or updating its value.

For what concerns the isolation of transactions, since they can update different partitions in different state operators within the same transactional graph in parallel, we cannot be sure that if T_i updates the state of operator S_1 *before* T_j, the same order will be maintained for the state of S_2, thus breaking isolation (see Figure 2). We solve this problem by enforcing an order between transactions.

We order transactions by their unique identifiers (integers) and we introduce a *scheduler* before every operator in TV: a scheduler is a non-partitioned operator that reorders the elements before feeding them to the subsequent operators, thus ensuring that they are processed in the desired order.

A partitioned state operator stores input elements in per-resource queues. Every enqueued element can access a resource only when the element that precedes it has released the lock. Every stream element generated by the state operator carries the result of the integrity check applied on the resource after its processing.

Downstream of the transactional graph, we introduce an operator that merges all the results of the integrity checks of a transaction and submits them to the state operators in TV. If at least one of the results is negative, the transaction aborts and the state operators roll back the affected resources to their previous value. Both in the case of commit and abort the state operator releases the resources related to the transaction and lets the next element in the queue access them.

We treat a query as a transaction that always commits, so it does not need to wait for the merge of integrity check results. Queries simply flow through state operators respecting the ordering of transactions specified above.

4.2 The Optimistic Strategy

The optimistic strategy does not avoid conflicts by locking, but lets them happen and replays the transactions in case.

Upon execution, the current transaction T_c is associated with a unique timestamp and the *watermark*, that is, the timestamp of the last completed transaction. Each state operator stores per-resource timestamps of the last transaction T_l that modified them. When a transaction is going to update a resource there can be two cases: (i) if T_c has a watermark that is greater than the resource timestamp, it means that it was executed *after* the termination of T_l; (ii) otherwise, it means that T_l is still running and T_c does not know if the current value of the resource will be committed or aborted. In the first case T_c can proceed in the update, in the second one, a conflict is detected and T_c is marked for replay. Replays, like aborts, require a rollback on every state operator in the transactional graph in order to restore resource values.

The entry point of the transactionl graph has four responsibilities: (i) it assigns a unique identifier to every transaction and a unique and incremental timestamp upon transaction execution; (ii) it receives the result of transactions and in case of abort or replay, peforms a rollback on the state operators; (iii) it replays transactions marked for replay; (iv) it keeps track of the current *watermark* by updating it accordingly on transactions completion.

	Avg latency	Throughput
VoltDB	5092 ms	589 tr/s
Pessimistic	8.2 ms	6235 tr/s
Optimistic	3.7 ms	21448 tr/s

Table 1: Average latency and maximum throughput for VoltDB, Pessimistic and Optimistic strategies

As in the pessimistic case, additional operators attached to the exit points of the transactional graph merge and feed the results of transactions back. However, the results are not fed to state operators, but to the entry point.

External queries to state operators use the watermark to gather consistent results. As for the pessimistic strategy, queries do not abort, however, they can be replayed in case their watermark is lower than at least one of the timestamp of the requested resources.

One of the drawbacks of the optimistic strategy is that the number of replayed transactions can explode if the number of conflicts increases. To mitigate this problem we elaborated a replay policy based on two considerations: (i) replaying a transaction with the same watermark twice will lead to a further replay; (ii) if transaction T_c needs to replay because it conflicts with T_l, then it should be replayed after T_l completes.

5 PRELIMINARY RESULTS

We implement our consistent stream processing approach on top of the Apache Flink [9] SP. We test our implementation with the bank system example proposed in Section 2, but without any fraud detection system (no `counter` operator was employed) and generating only transfers. The `balances` operator includes 100k different bank accounts split in 8 partitions. The origin and destination accounts for each bank transfer are selected randomly with a uniform distribution. We compare the results obtained from our system with VoltDB, a commercial in-memory database system. In VoltDB, we store the accounts in a table partitioned over 8 different machines 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 [1].

In the case of the pessimistic strategy and in VoltDB we deploy our system on a cluster of 20 Amazon EC2 t2 XL instances, each equipped with 4 CPU cores and 16 GB of RAM. In the case of the optimistic strategy we obtained preliminary results on a less performing deployment of two 12-core-processor Dell PowerEdge T320 machines.

Table 1 shows that VoltDB achieves a surprisingly low throughput of 589 elements/s with an high average latency of 5092ms. This result shows that inter-partition updates are very expensive for VoltDB: in another experiment with the same configuration with a load of read-only transactions, VoltDB achieves a maximum throughput of 236186 element/s with an average latency of 7.48s. The pessimistic strategy achieves a throughput of over 6k elements/s and a latency of 8.2 ms, while the optimistic strategy reaches an throughput of over 21k elements/s and a latency of

[1] https://www.voltdb.com/blog/programming-voltdb-easy-flexible-and-ultra-fast

3.7ms. These results show that our model is promising and competitive with a state-of-the-art representative of in-memory database systems.

6 RELATED WORK

Since the early 2000s, seminal works on stream processing define its requirements [5, 14] like handling imperfections (delayed, missing and out-of-order data); guaranteing data safety and availability; and provide approximate results in case of unbounded memory requirements by using sliding windows, or synopses, or other techniques. STREAM [3] is one of the first SPs that was able to process streaming data by the application of continuous queries that were expressed with CQL [4], a SQL-based declarative language that enables queries on streams by mapping streams to relations with the application of stream-to-relation operators (tipically windows), and back, using relation-to-stream operators. STREAM was developed as a single monolithic system. Borealis [1] is one of the first DSMS which was inherently distributed and that used a directed graph to express queries. The interested reader can deepen the topic on stream processing and its various fields of application in the survey of Cugola and Margara [11] about processing streams of information.

In the last year a large number of SPs was born, both as pay-per-use cloud and as open-source software. Among the most important open-source SP currently in development we can identify the top-level projects of the Apache Software Foundation Flink, Storm and Spark [9, 16, 17]. All of these SPs offer real-time, distributed and fault-tolerant stream processing capabilities. Akidau et al. [2] recently proposed a model to characterize stream processing computations that converged in the Apache BEAM project [2], an effort to provide a unified programming model that enables the users to implement and execute stream processing computations on any SP.

The model proposed in this paper adds database capabilities to SPs. We detail a brief state of the art for distributed databases. No-SQL databases often trade-off consystency for performance: MongoDB [6] provides transactions that involves a single document, DynamoDB [12] sacrifices strong consistency for eventual consistency under certain failure scenarios. H-Store [15] is the project from which VoltDB was born. It is an in-memory database that provides strong consistency on transactions that affects the same partitions by using a single thread of execution or it coordinates transaction execution in the worst case of a multi-partition transactions. S-Store [10] is a system that aims at providing the capabilities of a stream processor on top of H-Store. It provides the abstractions of streams as time-varying tables and operators as trigger functions.

This work was inspired by Martin Kleppmann's talk "Turning the Database Inside Out" and Jay Kreps' "Questioning the Lambda Architecture" [3] and the presentation held by Jamie Grier at Flink Forward 2016 "The Stream Processor as a Database" [4].

7 CONCLUSIONS

In this paper we address the problem of consistent operator state management in SPs. We define the model of transactions on the graph of processing and we propose two possible solutions, the pessimistic and optimistic strategies, to guarantee the atomicity and isolation of transactions. We implemented the solutions on top of the Apache Flink Stream Processor and tested the implementation both for latency and throughput and compared them with the in-memory database VoltDB. The results obtained are promising.

As future work we plan to relax the constraints for isolation between transaction and trade high isolation for performance. Moreover, we plan to integrate the snapshot algorithm for fault-tolerance already employed in Flink to extend its applicability to our transactional model.

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 *CIDR 2005*. 277–289.

[2] 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 2015* 8, 12 (2015), 1792–1803.

[3] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. 2003. STREAM: the stanford stream data manager (demonstration description). In *SIGMOD 2003*. ACM, 665–665.

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

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *PODS 2002*. ACM, 1–16.

[6] Kyle Banker. 2011. *MongoDB in Action*. Manning Publications Co.

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *SIGMOD 1995*, Vol. 24. ACM, 1–10.

[8] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[9] 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 Bullettin* 38, 4 (2015), 28–38.

[10] 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. *VLDB 2014* 7, 13 (2014), 1633–1636.

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

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review 2007* 41, 6 (2007), 205–220.

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

[14] Michael Stonebraker, UÇğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD 2005* 34, 4 (2005), 42–47.

[15] 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 *VLDB 2007*. VLDB Endowment, 1150–1160.

[16] 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 *SIGMOD 2014*. ACM, 147–156.

[17] 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, 10–10.

[2] https://beam.apache.org/

[3] https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/

[4] https://www.slideshare.net/FlinkForward/jamie-grier-the-stream-processor-as-a-database-building-online-applications-directly-on-streams