Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Doctoral Programme In Computer Science and Engineering

# New Horizons for Stream Processing

Doctoral Dissertation of:
**Lorenzo Affetti**

Supervisor:
**Prof. Gianpaolo Cugola**

Tutor:
**Prof. Luciano Baresi**

The Chair of the Doctoral Program:
**Prof. Barbara Pernici**

2019 – XXXI

*To my mother*
*and my father*

# Acknowledgements

I would like to thank my advisor, Gianpaolo Cugola, for his experienced and to-the-point advices, and Alessandro Margara for his continuous support during my studies.

Thanks to the reviewers of this thesis, who gave many suggestions in order to improve its quality.

There are many people that took part in my life and gave me support, love, and their friendship during this experience. I enjoyed most of the weekend nights with my great friends Alessandro and Mattia Piran. I found intellectual stimulation, sense of humor, great colleagues, and great friends in Riccardo Tommasini and Michele Guerriero, fellows in this great journey. Vanessa walked with me through the hardest parts of this path. My family always supported me, and, in particular, my mother and my father, who this thesis is dedicated to, provided me with all the unconditioned love they could. There are now words to express how grateful I am towards them, both for their support, and for what they taught to me to build the person I am now.

To all of you, thank you.

# Abstract

STREAM processing has gained tremendous attention over the last years and many Stream Processors (SPs) have been designed and engineered to cope with huge volumes of data coming at high velocity. Streams could contain stock options, user clicks in web applications, customer purchases in an e-commerce application, positions of robots in a warehouse, or temperature measurements from sensors. The common requirement for streaming applications is to process unbounded streams of elements and continuously compute queries like "what is the top purchased product?", or "what was the average temperature in the server room in the last second?" in order to take rapid compensating actions such as ordering a new stock of the top purchased product, or prevent fire in the server room. In order to continuously process huge amounts of elements and take real-time decisions, SPs exploit the computational power offered by multiple machines by distributing the computation and dividing data in shared-nothing partitions to avoid expensive data race management while processing. Stream processing is also a programming paradigm suited for designing novel event-driven applications with high throughput and low-latency requirements. Streams offer decoupling among the processing modules and, thus, enhance application modularity and composability. Indeed, SPs are playing a central role in the technology stacks of modern companies and they are covering more and more tasks that, in standard deployments, compete to other tools. The employment of one system instead of multiple ones reduces system integration complexity, communication latency, and facilitates application maintenance and modeling. Novel event-driven applications require a Database Management System (DBMS) for state management that is, indeed, embedded in the state of computation of the SP. However, due to its embedding, the DBMS suffers from some limitations such as the lack of multi-key transactions and consistent external querying. Eventually, their central role requires SPs to conform to a standardized execution semantics in order to improve their usability, interoperability, and interchangeability.

This thesis takes a step towards SPs standardization through highlighting the discrepancies between them, and a step towards their integration with DBMSs by extending their computational model to deal with transactional computation.

For SPs standardization, we use SECRET, a well recognized mathematical model to express their execution semantics, to model five distributed SPs that were developed after the introduction of SECRET itself and are today widely used in companies at the scale of Google, Twitter, and Netflix. We show that SECRET properly models a subset of the behavior of these systems and we shed light on the recent evolution of SPs by analyzing the elements that SECRET cannot fully capture.

In order to decrease system integration overhead and to overcome the limitations of the current approaches for DBMS over SP, we enhance the capabilities of the SP with DBMS's ones by extending the SP computational model with transactional semantics: we develop a unified approach for multi-key transactions on the internal state of the SP, consistent external querying with respect to transactional operations on the state, and streaming data analysis. We implement TSpoon, a prototypal implementation of our extended model, as an extension to the open-source SP Apache Flink$^{TM}$. We evaluate our prototype using synthetic workloads in various configurations to understand which metrics mostly impact its performance. Eventually, we evaluate a real use-case scenario and compare the results with the ones obtained from VoltDB, a commercial in-memory database known for its excellent level of performance: TSpoon outperforms VoltDB in the execution of multi-key transactions and proves to be a promising future direction for the integration of DBMSs and SPs.

# Sommario

Lo stream processing, ovvero, l'elaborazione di flussi di dati, ha richiamato grande attenzione negli ultimi anni. L'interesse crescente alla gestione di grandi volumi di dati prodotti ad altissime velocità ha dato vita alla progettazione ed allo sviluppo di molti Stream Processor (SP) —gli strumenti volti a processare tali flussi. I dati contenuti nei flussi possono essere di molteplice natura: fluttuazioni degli indici di borsa; click in applicazioni web; acquisti in un'applicazione e-commerce; posizioni dei robot nei magazzini; misurazioni di temperature da sensori o altro. I requisiti comuni per le applicazioni in questo dominio sono l'elaborazione continua di infiniti flussi di dati espressa per mezzo di *query* (i.e., interrogazioni sui dati) in modo da intraprendere azioni di compensazione in base ai risultati ottenuti. Per esempio, la query "qual è il prodotto più acquistato?" al fine di ordinarne una nuova provvista, oppure "qual era la temperatura media nella stanza server nell'ultimo secondo?" per prevenire un possibile incendio. Per processare grandi moli di dati senza soluzione di continuità e prendere decisioni in tempo reale, gli SP sfruttano la potenza computazionale offerta da più macchine distribuendo su di esse il calcolo e separando i dati in partizioni indipendenti, in modo da evitare costose operazioni di coordinazione. Lo stream processing è anche un paradigma di programmazione adatto alla progettazione di nuove applicazioni orientate agli eventi che richiedono alto throughput e bassa latenza. I flussi di dati, infatti, garantiscono disaccoppiamento tra i moduli applicativi e migliorano la componibilità e la modularità delle applicazioni stesse. Dato ciò, gli SP ricoprono un ruolo sempre più centrale nel portafoglio delle tecnologie utilizzate nelle aziende moderne e vengono utilizzati per svolgere compiti per cui, in passato, venivano impiegati altri strumenti. L'utilizzo di un solo sistema, infatti, evita la complessità di integrazione di diversi, riduce le latenze di comunicazione tra sistemi e facilita la manutenibilità e il design delle applicazioni stesse. Le nuove applicazioni orientate agli eventi, infatti, richiedono un Database Management System (DBMS) (i.e., un sistema per la gestione dei dati) per gestire lo stato della computazione, il quale viene incluso direttamente all'interno dello SP. Tuttavia, il DBMS così progettato soffre di limitazioni, come l'assenza di transazioni su più chiavi e l'impossibilità di garantire la consistenza delle interrogazioni esterne allo stato della computazione. Infine, dato il loro ruolo centrale, gli SP devono

conformarsi ad una semantica di esecuzione standardizzata per migliorarne l'usabilità, l'interoperabilità e per poter essere intercambiabili.

Questa tesi si muove verso la standardizzazione degli SP delineandone le discrepanze, e verso la loro integrazione con i DBMS estendendone il modello computazionale.

Per quanto riguarda la standardizzazione degli SP, viene utilizzato SECRET, un modello matematico per la semantica di esecuzione degli SP riconosciuto dal mondo accademico, per modellare cinque diversi SP distribuiti sviluppati dopo l'introduzione di SECRET stesso e che sono oggi ampiamente utilizzati in aziende come Google, Twitter e Netflix. Nella tesi, si mostra che SECRET modella propriamente un sottoinsieme dei comportamenti di questi sistemi e si pone evidenza su alcuni aspetti della recente evoluzione degli SP per mezzo dell'analisi di quegli elementi che SECRET stesso non riesce a catturare.

Per l'attenuazione delle problematiche e delle limitazioni degli attuali approcci di integrazione tra DBMS ed SP, viene adottato un approccio che porta le capacità di gestione dei dati degli SP ad essere più vicine a quelle dei DBMS tradizionali. In primo luogo, viene infatti esteso il modello computazionale degli SP con semantiche transazionali, dando vita ad un approccio unificato per: transazioni su chiavi multiple sullo stato interno allo SP; query esterne consistenti rispetto ad esse; ed analisi dei flussi di dati. In secondo luogo, viene implementato il sistema TSpoon: l'implementazione prototipale del nostro modello come estensione dello SP open-source Apache Flink™. Per capire quali siano le metriche che più impattano le performance del prototipo, viene dettagliata la sua valutazione per mezzo di workload sintetici con varie configurazioni. Infine, viene fornito un caso d'uso reale, i cui risultati vengono paragonati a quelli di VoltDB, un database commerciale in-memory noto per le sue elevate performance: TSpoon ottiene risultati migliori di VoltDB nell'esecuzione di transazioni su chiavi multiple, provando di costituire una promettente direzione futura per l'integrazione tra DBMS ed SP.

# Contents

**1 Introduction** **1**
1.1 Modeling SPs Execution Semantics . . . . . . . . . . . . . . . . . . . . 3
1.2 Transactions on the Stream Processor . . . . . . . . . . . . . . . . . . . 5
1.3 Thesis Contributions and Outline . . . . . . . . . . . . . . . . . . . . . . 8

**2 Related Work** **9**
2.1 Processing streams of data . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Modeling stream processing . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3 Distributed databases . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

**3 SPs Execution Semantics** **19**
3.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.1.1 The SPs Computational Model . . . . . . . . . . . . . . . . . . . 19
3.1.2 SECRET . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.2 Analysis of Stream Processing Engines . . . . . . . . . . . . . . . . . . 23
3.2.1 Experimental methodology . . . . . . . . . . . . . . . . . . . . . 24
3.2.2 Flink . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.2.3 Storm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2.4 Spark . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.2.5 Google Cloud Dataflow . . . . . . . . . . . . . . . . . . . . . . . 29
3.2.6 Azure Stream Analytics . . . . . . . . . . . . . . . . . . . . . . . 29
3.3 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.3.1 Time model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.3.2 Windowing approaches . . . . . . . . . . . . . . . . . . . . . . . 32
3.3.3 Management of out-of-order elements . . . . . . . . . . . . . . . 33
3.3.4 Graph of operators . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.3.5 Fault tolerance . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.3.6 Summary and open challenges . . . . . . . . . . . . . . . . . . . 34

**4 Transactions on the Stream Processor** **37**
4.1 State Management Capabilities . . . . . . . . . . . . . . . . . . . . . . 37
4.1.1 Database Management Systems . . . . . . . . . . . . . . . . . . . 37
4.1.2 Stream Processors . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.2 Limitations in the SP model . . . . . . . . . . . . . . . . . . . . . . . 40
4.2.1 Transactional guarantees . . . . . . . . . . . . . . . . . . . . . . 41
4.2.2 Queryable state . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.2.3 Executive summary . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.3 Transactions on a Stream Processor . . . . . . . . . . . . . . . . . . . 42
4.3.1 Stream processing model . . . . . . . . . . . . . . . . . . . . . . 43
4.3.2 State management model . . . . . . . . . . . . . . . . . . . . . . 44
4.3.3 Transactional guarantees . . . . . . . . . . . . . . . . . . . . . . 45
4.3.4 The model in action . . . . . . . . . . . . . . . . . . . . . . . . 46
4.3.5 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.4 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.4.1 TSpoon API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.4.2 TSpoon architecture and transactional guarantees . . . . . . . . 49
4.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.5.1 Experiment setup . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.5.2 Default scenario . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.5.3 Isolation levels and concurrency control strategies . . . . . . . . 55
4.5.4 Sensitivity to parameters . . . . . . . . . . . . . . . . . . . . . . 56
4.5.5 Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

**5 Conclusions and Future Work** **63**

**Bibliography** **65**

CHAPTER 1

# Introduction

From stock options to user clicks in web applications, from customer purchases in an e-commerce application to positions of robots in a warehouse or temperature measurements from sensors, data is produced and processed as *unbounded streams of elements* in a wide variety of contexts. Streaming data is huge in volume, continuously produced, and used to take rapid business decisions. Stream Processors (SPs) is the generic name of those systems specifically tailored for processing large volumes of streaming data as it is produced; as such, they exploit the computational power offered by multiple machines as much as possible by distributing the computation and dividing the data in share-nothing partitions to avoid expensive data race management while processing.

Thanks to their versatility, their ability to scale over multiple machines to cope with huge volumes of continuously produced data, and their native design for time-dependent computation, SPs address the most interesting novel use cases for data-intensive applications. We forecast that SPs will play a central role in the stack of modern companies and that they will cover more and more tasks that, at the moment, compete to other tools[1]. This process must pass through a standardization of their execution semantics and the analysis of their suitability to use cases different from pure data analysis. In line with this assumption, this thesis takes a step towards a standardized execution semantics through modeling the discrepancies between distributed SPs and a parallel step towards their generality by extending SPs computational model to deal with transactional computation.

In particular, stream processing is often employed for analytical queries on streaming data; e.g., the most purchased item in the last hour, the average temperature in a room, the top ten trending topics in a social platform. This kind of queries are often

[1] Proposals for all-streaming architectures are flourishing in non-academic context: http://milinda.pathirage.org/kappa-architecture.com/, https://data-artisans.com/blog/drivetribe-cqrs-apache-flink

time-dependent, indeed, SPs offer windowing —dividing elements in slices based on their timestamp— out-of-the box. However, as a newborn processing paradigm, stream processing lacks of a standardized semantics for time-dependent computation and existing SPs adopt different processing models [25]. This severely hampers the usability, interoperability, and interchangeability of SPs, since a user needs to understand system-specific aspects to confront various alternatives and select the ones that better suit her needs. The need for modeling SPs execution semantics led in 2010 to the definition of a formal model called SECRET [25]. However, SECRET targets first-generation, single-node SPs that answer fixed-shape queries expressed with some declarative language [12]; modern SPs are *distributed* over clusters of machines to cope with huge volumes of streaming elements and express the most various streaming computations through a graph of user-defined operators often programmed using full-fledged, standard programming languages (e.g. Java).

In order to provide users with a high-level tool that relieves them from understanding the system-specific intricacies of distributed SPs execution semantics, we asses if SECRET can still capture the differences between this new breed of SPs. We discover that this is not the case and we propose where future modeling efforts should be directed to [5].

Besides this effort in formalizing the semantics of existing systems, we address the problem of the maintenance and the management of heterogenous systems that employ SPs together with traditional Database Management Systems (DBMSs). Indeed, modern data-intensive applications include more components to achieve their goals: SPs, stateless application servers, and DBMSs [53, 63] for their *state management* capabilities —i.e., consistently execute parallel updates and queries on the application's state, and handle recovery from failure. Integrating different systems may hinder the design, implementation, and maintenance of the system, forcing developers to adopt different languages and processing paradigms, and to manually integrate the different sub-systems in a coherent way.

In order to overcome system integration complexity and to facilitate data-intensive application maintenance and modeling, we develop a unified approach for both data analysis and state management on the SP by extending the stream processing paradigm with the state management capabilities of databases. We therefore enhance the SP capabilities with queryable state and transactional behavior for state updates. In doing so, we both extend SPs computational model with transactional semantics and we prototype and evaluate our solution extending the open-source SP Apache Flink™ [4].

The research objectives of this thesis are:

**O1** highlighting the current discrepancies in distributed SPs execution semantics;

**O2** proposing future directions for proper modeling of distributed SPs execution semantics;

**O3** designing a unified stream processing model for transactional and analytical data processing;

In the remainder of this chapter we motivate our research objectives. Section 1.1 treats research objectives **O1** and **O2**, and motivates the need for new models for distributed SPs. In Section 1.2, we motivate objective **O3** by showcasing the problems of

integrating heterogenous systems and providing an overview of our solution for their integration. Finally, in Section 1.3, we highlight the various contributions of this thesis and its outline.

## 1.1 Modeling SPs Execution Semantics

In this section, we detail some of the issues that come along with different execution semantics of SPs through a simple example.

The main factors that differentiate the behavior of SPs are the models of *time* and *windows* they adopt [14]. Windowing consists in splitting the stream into finite blocks of contiguous elements —called windows— and performing the computation within the bounds of each window. Several types of windows exist: for instance, windows can be defined either based on the number of elements they contain or based on time boundaries, and they can partition a stream into non-overlapping chunks or contain common elements [14]. Windows are useful for two different reasons. First of all, they enable computation that takes into account the *recency* of the elements. For instance, queries like "...in the last 5 minutes" or "...for the last 100 measurements" would be otherwise impossible. Secondly, windows enable computations that would be otherwise unfeasible on unbounded datasets such as streams. For instance, counting the number of elements is not possible in general, since streams never terminate. The common solution to those use cases consists in windowing the stream and performing the computation within the bounds of each window. In the case of *time windows* (windows defined by the progressing of time) their semantics strictly depend on the related concept of time, which determines how the incoming elements are timestamped and associated to different windows. In the case of *event time*, time is seen as meta-data associated to each element in a stream either by the source that produces that element or by the SP itself. In the case of *processing time*, time refers to the system clock of the physical machine that runs the computation.

In order to better explain how different processing semantics could hamper SPs interoperability, we examine the case of a system for monitoring temperature in a building. The system computes the average temperature every 2 minutes and displays the last processed values on a web UI. It is composed of the sensors that collect measurements, the SP that computes the average, the DBMS that stores the values, and the application server that queries the database and displays the results obtained.

Figure 1.1 shows the input stream of couples $(timestamp, measurement)$ and the output for three different SPs.

- With $SP_1$ the user cannot know what happened to window $w_2$ and she should account for that. For instance, if the UI periodically queries the database for window results, it should support the absence of values and handle that case.

- With $SP_2$, while implementing the UI, the user should take into account that the result for $w_1$ has been produced twice and act accordingly; for instance, she should display only the last result available.

- With $SP_3$ the user should account for null values to avoid software failures; for instance, if the schema for temperature measurements in the database does not

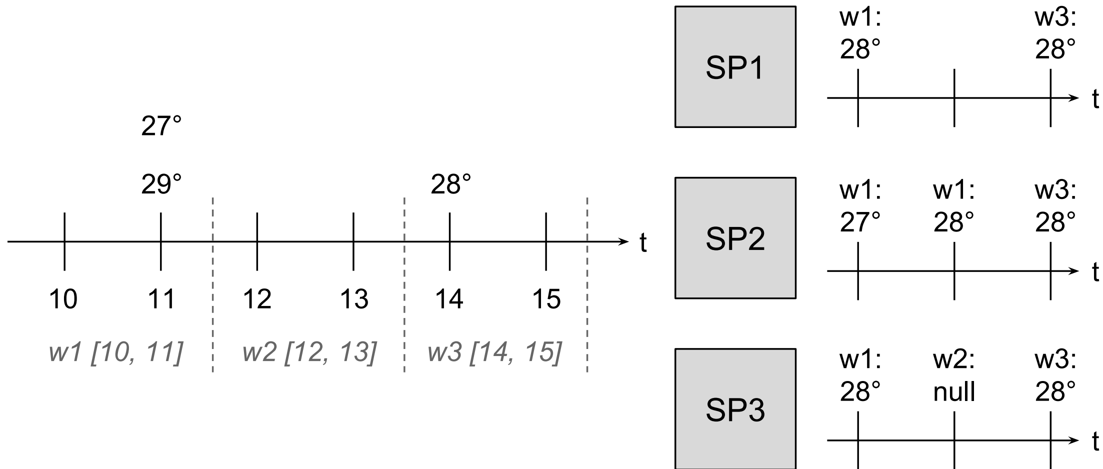



**Figure 1.1:** *Three SPs with different execution semantics and the output produced.*

allow null values, the insertion of the result for window $w_2$ would fail and cause the system to crash.

Indeed, with the three SPs listed above, we are exemplifying existing SPs semantics (see Chapter 3 for further details):

- $SP_1$ waits that every element for a time window has come and outputs the results of the average. If the windows is empty, it emits no output;
- $SP_2$ does not wait and outputs the result of the computation as soon as a new element comes;
- $SP_3$ is like $SP_1$, but it outputs empty windows results as null values.

Without a formal model of the SP behavior, the user should reverse engineer the implementation details of the SP from its output and adequate downstream systems accordingly. This could be overmuch expensive and not feasible in production phase and, thus, it severely hampers the usability and interoperability of SPs with other systems. A model that provides a high-level specification of SPs execution semantics and highlights their differences (i) relieves the user from the complex task of deducing the SP internals from its output, (ii) lowers the risk of problems and bugs in the system, (iii) and allows for a grater flexibility in swapping SPs and interoperate them.

In 2010, the problems in handling discrepancies between SPs motivated the definition of SECRET, a model that captures the behavior of the SPs available at that time [25]. SECRET was used to analyze both academic and industrial SPs. The semantics of the former were typically well and precisely defined —yet different from system to system— while the semantics of the latter were most often dependent on hidden implementation details. SECRET could effectively capture and confront the behavior of all the systems it was applied to.

After the definition of SECRET, the increasing number and the growing complexity of real-time data analytics applications led to a bloom of new *distributed SPs* that target cluster platforms to scale with the volume and velocity of input data. In order to asses if the semantics of time and windowing has changed in these new systems and to model

the discrepancies in their execution semantics, we used SECRET to model the behavior of five distributed SPs. We discovered that SECRET is not able to capture some aspects of their behavior and some windowing strategies that they employ. However, for the aspects that SECRET is able to capture, we discovered relevant discrepancies among their execution semantics [5].

In Chapter 3 we tackle research objectives **O1** and **O2**. We explain how we used SECRET to model Flink, Storm, Spark Streaming, Google Dataflow, and Azure Stream Analytics —distributed SPs that were developed after the introduction of SECRET itself and are today widely used in companies at the scale of Google, Twitter, and Netflix. We show that SECRET models well a subset of the behaviors of these systems and we shed light on the recent evolution of SPs by analyzing the elements that SECRET cannot fully capture.

## 1.2 Transactions on the Stream Processor

Companies often need to integrate *data analytics* tasks —complex computations over the input data— with *state management* tasks —transactional updates and queries to the application state. In this section we analyze two different system architectures that combine SPs for data analytics and DMBSs for state management and we explain their limitations. We, therefore, motivate the need for unification of SPs and DBMSs under a unique system.

As a first example, consider a system where the DBMS applies its transactional logic and provides the outcome of executed transactions to a downstream SP that extract useful analytics by processing them. For instance, take a banking system that accepts deposits, withdrawals, and money transfers from an account to another; applies them to the users' accounts; and computes analytics on the transactions executed; e.g., it calculates the most active geographical regions in the last hour, and/or the top ten biggest transactions executed in the last month, updating the results every time a new transaction arrives. The resulting system is composed of a database that applies the transactional logic for bank operations and a SP that computes the low-latency analytics and stores the results on the DBMS for retrieval (Figure 1.2a).

As a complementary example, consider a system where the SP processes the raw requests from users and operates transactions on a downstream DBMS. For instance, take a voting system in which voters send votes about participants in a talent show. The system wants to reward the singers that have a constant rate of votes over time and so, every second, it gathers the 10 most voted singers of the last 5 seconds and adds a point to them and it does the opposite for the 10 least voted singers. The resulting system is composed of a SP that computes the low-latency analytics and stores, within a single database transaction, the points for singers on the downstream DBMS (Figure 1.2b).

In both examples, administrators must deploy both a database and a SP and manage their interaction.

In the bank use case, the SP must extract the outcomes of transactions from the database: a first solution is to push the outcomes of transactions to the SP directly from the database. However this approach could not always be feasible because it degrades the performance of the database when updating its state.

Another solution is to poll the database at some fixed interval and process the ob-

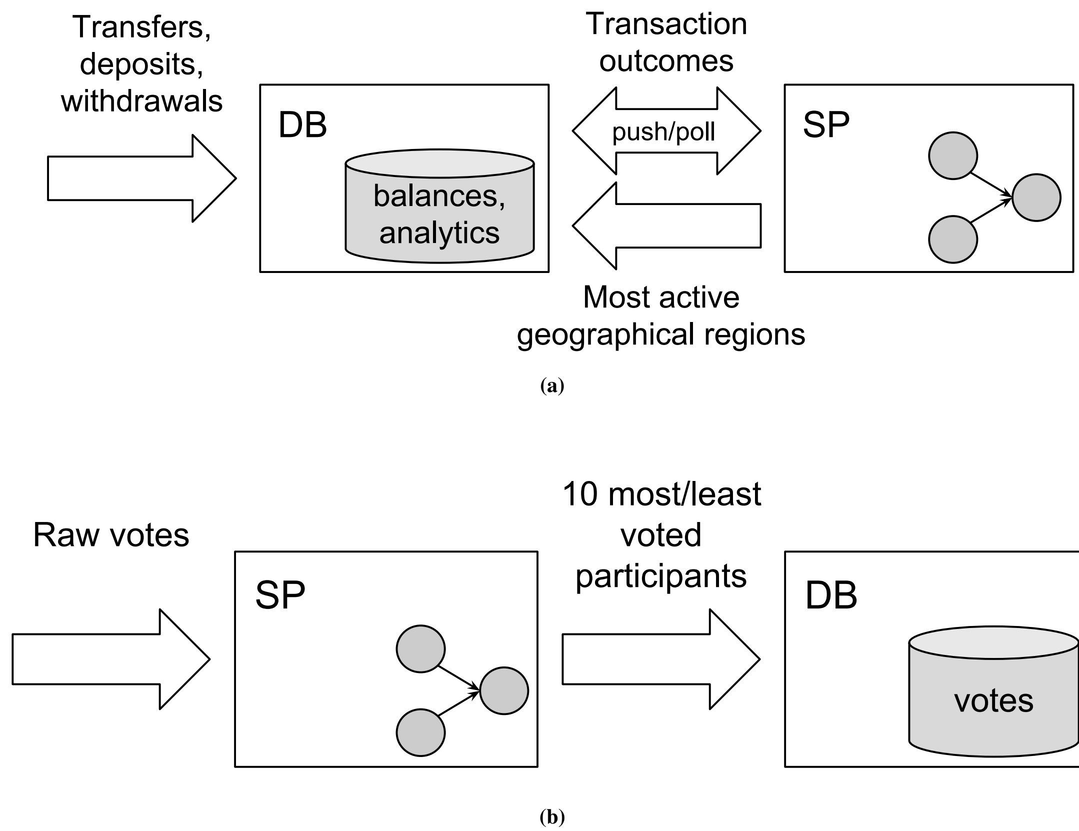



**Figure 1.2:** *Two systems that integrate a SP and a DBMS. (a) The DBMS executes transactions and the SP computes analytics on their outcome. (b) The SP computes analytics and saves the results on the downstream DBMS.*

tained results; however, this approach could be overwhelming for the database if the SP polls it at a high rate, or produce out-of-date results if employing a low rate. In general, system administrators should fine tune the polling interval to achieve a proper recency-performance trade off basing on the workload of the database and the computational power it is granted at the moment. This task requires a deep knowledge of the system architecture and leads to further reconfiguration steps when the physical deploy changes.

In the voting use case, on the contrary, the SP should be fine tuned in order to send transactions at a reasonable rate to the database based on its maximum sustainable workload. Again, administrators should fine-tune the systems and take care of reconfiguring them every time that the physical deploy changes.

Both applications fall into some major drawbacks due to the need of integrating multiple systems:

- *maintenance* of multiple systems: deploying, updating, and connecting many systems together requires configuration effort and careful design by dedicated personnel with a deep understanding of the semantics of individual systems, with the risk of introducing functional errors and performance problems;
- *variety* of programming paradigms: different systems have their own lexicon and abstractions, this introduces overhead in the communication among different developing teams that must find a common ground to discuss the capabilities of the specific subset of the application and continuously maintain their interaction and integration;
- decreased *responsiveness*: when requests cross the boundaries of multiple components of the system, they incur in additional network latency.

We, therefore, propose to unify the state management capabilities of DBMSs and the computational capabilities of SPs under the same system and programming abstraction. By unifying the two components, we address the problems listed above by providing:

- *maintenance* of a single system: with a unified approach, the administrators only need to understand and deploy a single system for the application to work;
- *unified* programming paradigm: developing teams can focus on the functionalities that the application must provide as a whole. A single team, indeed, can develop all features of the system from state management to data analysis without switching to a different knowledge domain;
- *responsiveness*: there is no cross boundary communication between the database and the SP, and so, a request is processed with no additional latency imposed by architectural overhead.

In order to integrate state management and computational capabilities, we extend the dataflow model of computation of stream processing to account for transactional computations.

In the dataflow programming paradigm, computation is expressed as a graph of operators connected by streams of elements. Each operator applies some functional logic to input elements and produce results to downstream operators. The operators

can be either *stateless* (e.g., filtering the odd elements in a stream of integer elements) or *stateful* (e.g., count the occurrences of words in a stream of tweets) [2, 7, 9, 29]. We extend this model with *transactional subgraphs*: well-defined areas in the graph of computation in which the operators both expose their internal state to external queries and apply consistent updates to such state by guaranteeing transactional behavior of operations [4].

In Chapter 4, we tackle research objective **O3**. We describe in detail the limitations that prevents SPs from expressing transactional behavior and what transactional subgraphs guarantee to the user. We also present TSpoon, a prototypal implementation of our extended model of computation that extends the open-source SP Apache Flink™ with transactional subgraphs; we describe its implementation; and we evaluate its performance.

## 1.3 Thesis Contributions and Outline

Firstly, in tackling the problem of assessing the adequacy of SECRET in modeling modern distributed SPs, this thesis contributes to the research on stream processing in several ways: (i) it provides a precise modeling of five modern SPs using SECRET; (ii) it compares systems and highlights their similarities and differences, thus helping the users to identify the systems that better satisfy their requirements; (iii) it identifies some aspects of modern SPs that SECRET cannot fully capture; (iv) based on these aspects, it discusses the evolution of SPs since the definition of SECRET and suggests promising directions for future modeling efforts.

Secondly, with our effort in system integration this thesis contributes to the research on stream processing and state management in various ways: (i) it introduces a novel model that seamlessly integrates queryable state and transactional semantics within a SP; (ii) it formalizes the semantics of transactions on the state of a SP and proposes configurable levels of isolation and durability for these transactions; (iii) it presents the TSpoon system, that implements the new and extended streaming model, and it explores different strategies to achieve the proposed guarantees; (iv) it offers a detailed evaluation of the performance of TSpoon, focusing on the benefits and costs of the different strategies for transactional semantics and on the comparison with state-of-the-art tools for distributed stream processing and state management.

In particular, Chapter 3 analyzes the SECRET model, describes its limitations by using it to model five modern distributed SPs. Chapter 4 introduces our model for transactions on the SP and presents and evaluates TSpoon, our implementation on the Flink SP. Finally, Chapter 2 presents the related work and Chapter 5 concludes this thesis and forecasts future work.

# CHAPTER 2

# Related Work

The inter-disciplinary nature of this thesis relates it to several fields like stream processing, database systems, and data management architectures. This chapter is divided into sections that overview them. Section 2.1 overviews the main approaches and systems to process streams of data which is the main field touched by this thesis. Section 2.2 overviews the work related to the definition of the execution semantics of modern distributed SPs. Section 2.3 overviews the relevant work for the integration of distributed DBMS and SPs, such as distributed databases state management capabilities and modern big data architectures that unify low-latency computation and consistent state management.

## 2.1 Processing streams of data

Stream processing applies transformations to unbounded sequences of elements to produce relevant results. The type of transformations are manyfold and originate from various use cases: continuous query answering, pattern matching, and continuous event processing.

Continuous query answering is similar to classic DBMS query answering with the difference that the query is not "one-shot", but it is "installed" in the SP and it is continuously executed as new elements come and update its result. Consider the query "count the number of taxi drivers that earned more than 2000$". If it is executed as a standard, one-shot query (it can be expressed in SQL, and executed by a DBMS for example), the result is a list of taxi drivers that never changes through time. If the query is executed by a SP, its result is a *stream* of taxi drivers that evolves through time according to the fares paid by passengers. Typically, the SPs that are tailored for query answering provide also abstractions to deal with *time*. It is common that the continuous query

contains some time reference, such as "count the number of taxi drivers that earned more than $2000\$$ *in the last month*". The SPs that provide *time windows* can slice input streams within time boundaries and perform the query on the content of the window itself.

Pattern matching is different from continuous query answering in that, given a *rule* and stream sources, it extracts patterns of primitive events to generate new streams of complex events. To better understand the difference from continuous query answering, consider a scenario in which a forest is equipped with smoke, humidity, and temperature sensors that produce streams of measurements. A system for continuous query answering analyzes the measurements of temperature and humidity and it can calculate, for instance, "the average temperature and humidity every second"; while a system for pattern matching, given a rule, is able to detect fire in the forest. The rule could be "if there is $Smoke$ and, in the last $5$ minutes, the $Temperature$ is above $45$ degrees, and there was not $Rain$, then signal $Fire$". $Fire$ is the complex event identified by the composition of $Temperature$, $Rain$, and $Smoke$ by means of the rule described.

Continuous event processing is different from the two above: there is no such a concept of "query" or "rule", but an *event-driven streaming application* that embeds its business logic and the state of computation —that is exposed to external systems for querying. The focus is not only on the transformation of the input streams into output streams, but also on the side effects that the events generate on the state of computation and on the external actions that they cause once processed.

The three approaches can be combined together. For example, an SP capable of continuous event processing, continuous query answering, and pattern matching can implement a complete social network. User posts and comments are stream of events ingested by the *streaming application*. Operators process events, apply their specific application logic, and store the effect in their state: they ingest posts, validate them, and store them for querying —for example, for the presentation layer. The new posts, that passed through the application logic, trigger a *continuous query* that computes the top cited users over all posts. Eventually, events are *pattern-matched* against a rule for detecting anomalous behavior of users (e.g., fake post creation to cite a user programmatically); bad users are used by the application logic to strengthen the validation step, in turn.

This thesis is relevant for the first and the third type of systems. Indeed, Chapter 3 treats the execution semantics of distributed SPs for continuous query answering and Chapter 4 extends the model of computation of modern SPs to accommodate event-driven streaming applications.

Several SPs have been proposed both from the academia and from the industry in the last decade, thanks to an increasing need for continuously processing unbounded sources of information and for producing low-latency results. As anticipated in Chapter 1, many of these systems differ in their execution semantics and none of them provides a unified approach to distributed transactional and analytical processing. In the remainder of this section, we provide an historical overview of the SPs related to the three areas of continuous query answering, pattern matching, and continuous event processing.

We distinguish two generations of SPs. The first generation flourished in the mid 2000s and focuses on the definition of abstractions for continuous query answering

–Data Stream Management Systems (DSMSs)– or on pattern matching to detect situations of interest from streams of low-level information —Complex Event Processing (CEP) systems. The interested reader can refer to the detailed survey of these systems by Cugola and Margara [37].

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 [14]. The reference model of DSMSs has been defined in the seminal work on the Continuous Query Language (CQL) [12]. In CQL, the processing of streams is split in three steps: first, *stream-to-relation* operators –windows– select a portion of each stream to implicitly create static database table. 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. An example of CQL query in natural language is “calculate the average speed of cars in the last 5 minutes for every segments of this road”. The windowing operator (“the last 5 minutes”) is used to extract a relation from the stream of speed measurements, which is used to compute the query that extracts the average on the speed field. Eventually, the results of the aggregation generate the output stream. Several variants and extensions of CQL have been proposed, but they all rely on the same general processing abstractions defined above. The declarative language of Azure Stream Analytics that we analyze in Section 3.2.6 also derives from this processing model. The SECRET model we adopt in Chapter 3 was originally designed to capture the processing semantics of this kind of systems [25, 42]. 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 the subject of this thesis.

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 [44, 58]. CEP systems typically consider the elements of a stream as notifications of event occurrences and express patterns in form or *rules* using constraints on the content and time of occurrence of events [26, 35]. Their internal behavior is different from DSMSs, that slice streams in time windows and compute aggregates on them. For instance, the T-Rex CEP [36] system translates rules into *event detection automata*. Every time a new event enters the system, T-Rex checks wether new automata instances must be created and if the event activates a transition that leads the automata to the next state. If an automata terminates (reaches the final state), the composite event must be generated. Interestingly, some CEP systems use interval timestamps. In this model, each data element is associated with two points in time that define the first and the last moment when it is valid [72, 82]. This model is never used in the modern SPs that we consider in this thesis and is not captured by SECRET.

More recently, the artificial intelligence and knowledge representation communities also started investigating streaming data posing an accent on integration and automated reasoning. This emerging field of research was named Stream Reasoning [40, 62]. In less then a decade, it has extended the Semantic Web stack with the RDF Stream data model, several continuous extensions to the SPARQL query language, and reasoning

algorithms optimized for RDF streams [11, 19]. Systems in this area are often referred to as RDF Stream Processing (RSP) engines. The interested readers can refer to the working drafts of the W3C RSP Community Group[1].

First-generation systems were mainly adopted for continuous querying, and, thus, they do not support internal state queryability, nor transactional execution of groups of operations.

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 for continuous query answering. The research on Big Data initially focused on static data and batch processing and proposed functional abstractions such as MapReduce [38] to automate the distribution of processing. Subsequent research increased the expressivity of MapReduce, enabling developers to specify arbitrarily complex directed graphs of operators [85]. 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 [84].

The second generation of SPs inherits the same processing model based on a graph of functional operators. Initially, the engines did not provide any built-in construct to express windows, and developers had to implement windowing manually if required by the application at hand. This is for example the case of Storm prior to version 1.0 [78]. Given the complexity of time and window management, windows have become first class objects in all modern SPs, as well as primitives to express event time and manage out-of-order records to produce correct results [9, 29, 54], and this motivates the need for a precise modeling and analysis of their behavior. Although being similar from their computational model and their time-related primitives, modern SPs differ in their processing techniques. Some of them, for instance Spark Streaming [86], provide streaming computations on top of batch processing by splitting each stream into small static chunks (micro-batches). Remarkably, Naiad [66] follows a similar approach. In order to achieve streaming computation, the developer has to divide the input streams into epochs that are fed iteratively to the graph of computation. *Timely Dataflow* –Naiad's programming model– timestamps input records with their epoch and uses it to track the completeness of each step of computation. Thanks to this approach, Naiad achieves fault tolerance by synchronously checkpointing operators' state among epochs. Using the same model, Naiad supports more use cases, such as iterative batch processing (e.g., calculate the PageRank algorithm [67] on a graph of millions of nodes) by feeding the results of each epoch to the next one until convergence. 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 [78], Heron [55], Google DataFlow [9], and Flink [29], which we adopt and extend in Chapter 4. The distribution across machines of SPs posed tremendous attention on their fault tolerance. The second generation, not only embeds time-related primitives, but also it aims at providing fault-tolerant stateful computation by preserving a consistent state across machines and guaranteeing exactly-once processing semantics —i.e, the processing of each record affects the state of computation only once. The paper about the Asynchronous Barrier Snapshotting algorithm by Paris Carbone [28] shed light on the evidence that consistently snapshotting the operators'

[1] `http://www.w3.org/community/rsp/`

state of a distributed graph of computation is possible with a low overhead while processing new records; the algorithm, indeed, was later implemented and used in many production setups by Apache Flink™ [27]. Remarkably, modern distributed SPs converged to produce correct (with respect to time) and consistent (with respect to failure) results at high throughput and low latency. In these systems, shared-nothing state is of paramount importance to achieve high performance; indeed, none of them consider the case of distributed transactions over their state for DBMS integration as TSpoon does.

The second generation of SPs not only performs low-latency streaming data analysis, but it can also support *event-driven streaming applications*. As anticipated at the beginning of this section, they are stateful applications that ingest events from one or more event streams and reacts to incoming events by triggering computations, state updates, or external actions. In contrast with traditional applications with separated compute and data storage tiers, they are based on stateful stream processing applications, where data and computation are co-located for better performance, both in terms of throughput and latency. Fault-tolerance and persistence is achieved by using the native mechanisms of the SP. Flink provides integrated state management capabilities, event time management, and an API for scheduling tasks according to the flow of event time to implement complex business logic[2]. Apache Kafka is a distributed streaming platform that allows to store streams, publish and subscribe to them, and process their records to produce derived ones[3], by leveraging a robust, fault-tolerant, distributed logging system at its core [79]. It provides both a library for stateful, distributed stream processing and low-level APIs for implementing producers and consumers of messages, thus enabling more flexible patterns of production/consumption from streams. Moreover, stored streams enable the addition of operators to the graph of computation at runtime by requesting the records from one or more streams as needed. Kafka is more flexible in application composition with respect to Flink –that does not allow to add operators to the graph of computation at runtime– however, its state management capabilities are limited to the stateful stream processing library.

Even if these modern features enable more complex stream processing applications, they still cannot fully integrate DBMSs capabilities because of shared-nothing state limitations. We tackle these limitations in Chapter 4 by providing a model for stream processing transactions, and the working prototype TSpoon, a SP that can express and execute both transactional and analytical graphs of computation.

Related to processing dynamic data, the programming language community proposed Reactive Programming (RP) abstractions [16], which build on three pillars 1. time-changing variables and explicit definition of their dependencies; 2. automated propagation of changes. RP shares many similarities with stream processing, with the graph of dependencies between variables being analogous to the graph of computation in the SP. Some recent proposals in the field study the trade-off between consistency and performance in distributed RP, which is closely related to the topic of this thesis [43, 60, 61].

As a final clarifying remark, the term "transaction" is used by some SPs, although with a different meaning that in the context of this thesis (see Chapter 4). Trident[4] enables exactly-once semantics on top of Storm [78] by dividing the input streams

[2] `https://flink.apache.org/usecases.html#eventDrivenApps`.
[3] `https://kafka.apache.org/`.
[4] `http://storm.apache.org/releases/1.0.6/Trident-tutorial.html`.

into batches and transactionally updating the state of the graph of computation upon batch processing completion. Similarly, Kafka [79] provides exactly-once semantics by transactionally placing markers on its output streams once a batch of input records has been consumed and produced its results[5]. Both approaches are an alternative to Flink's asynchronous checkpointing algorithm [27] and have to be considered as such. They are orthogonal to our approach to model multi-partition, multi-state updates on the internal state of the SP with ACID transactions.

## 2.2 Modeling stream processing

Several formalisms have been proposed to specify the execution semantics of individual stream and event processing engines, ranging from automata [6, 26] to temporal logic [35], to event algebras and calculi [13, 52]. While these formalisms often capture the semantics of time-based operators in general and window operators in particular, they are tailored to the specific features offered by the language they formalize and thus are not suitable to study the similarities and differences between heterogeneous stream processing systems.

Only few models have been proposed in the past to describe and compare some of the SPs discussed above. The survey by Cugola and Margara introduces a framework of models that capture the key aspects of DSMSs and CEP systems [37], such as the data and processing models, the processing language and operators, and the semantics of time. Etzion and Niblett propose the Event Processing Network (EPN) formalism to discuss CEP systems and event processing architectures [44]. An EPN is a directed graph where nodes represent operators and edges represent data flows between operators. These proposals, however, are descriptive and do not provide a formal ground to assess the commonalities and differences of the various SPs. To the best of our knowledge, SECRET represents the first attempt to formally capture the time and window semantics of SPs.

Interestingly, Dell'Aglio et al. define a model built on SECRET to capture the execution semantics of RSP engines [41]. The W3C RSP community group adopts it, together with LARS [21], as a reference model to define the semantics of RSP query language.

Given the relevance of performance —throughput and response time— for SPs, several proposals aim to model performance characteristics of SPs with the goal of predicting or improving some quality of service metrics or the allocation of resources [20, 31, 68]. These works are complementary to the proposal of Chapter 3, since they focus on predicting the performance of SPs rather than modeling their execution semantics.

## 2.3 Distributed databases

Distributed relational databases provide ACID transactional guarantees through distributed commit protocols, concurrency control algorithms [24], and recovery mechanisms [64]. TSpoon builds on these concepts to integrate transactional semantics within the SP.

[5] https://www.confluent.io/blog/transactions-apache-kafka/.

The tension between strong consistency and guarantees in data management systems has been widely investigated, leading to the formulation of various levels of isolation [3, 22, 46]. Indeed, with the increasing size of data-intensive applications [53], a number of systems trade consistency (for instance, in the case of replication, failure, or high contention) and strong transactional semantics for scalability and/or availability. Indeed, most commercial DBMSs used in production environments such as MySQL, Postgres, SAP HANA, or MS SQL Server do not offer *serializable* as their default isolation level, but *read committed* [15]. Serializability of transactions, indeed, comes at a high cost both in throughput and latency. Moreover, it severely hampers the availability of the distributed DBMS. High Available Transactions (HATs) offer a family of isolation levels and data replication semantics still guaranteeing an "always on" distributed DBMS upon failure. In its work, Peter Bailis proves that HATs cannot achieve serializability [15]. To the extreme of this approach, in order to provide maximum availability and scalability, NoSQL databases provide a limited version of transactions or directly drop their support. For instance, Amazon's key-value store Dynamo [39] was born without transactional support; the document-based database MongoDB only supports transactions that involve a single document [17], and the Redis key-value store does not support arbitrary distributed transactions [32].

More recently, NewSQL database systems aim to reconcile strong transactional semantics and efficient distributed state management for some workloads. According to Michael Stonebraker (Turing Award, 2015), this new wave of DBMSs must embody the following characteristics in order to address the new requirements of high transactional throughput and real-time query answering: (i) SQL as the primary mechanism for application interaction; (ii) ACID support for transactions; (iii) a non-locking concurrency control mechanism so real-time reads will not conflict with writes, and thereby cause them to stall; (iv) an architecture providing high per-node performance; (v) a horizontally scalable, shared-nothing architecture, capable of running on a large number of nodes [75]. H-Store [76] is an in-memory database that enforces atomicity of transactions on single partitions of the state through single threaded computations, and schedules multi-site operations to ensure ACID properties. Furthermore, H-Store enables transaction optimization by enhancing the support for pre-compiled stored procedures. This approach later evolved in the VoltDB database system that we used in our evaluation [59]. Calvin is a *deterministic* DBMS that gathers transactions into batches and applies an agreement protocol to make every site of the distributed database agree on the execution order of transactions. Once reached the agreement, transactions can be executed (and re-executed in case of failure) deterministically and without employing any distributed commit protocol. Indeed, Calvin does not allow programmatic abort of transactions; i.e., transactions can only abort because of non-deterministic hardware/-software failure [69, 77]. Spanner [34] is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is able to provide, at global scale, strong consistent reads and writes, and globally-consistent reads across the database at some timestamp in the past. These features are enabled by the fact that it assigns globally-meaningful timestamps to distributed transaction that reflects the serialization order by using its *TrueTime* API. TrueTime exposes clock uncertainty, so that Spanner can wait out that uncertainty upon commit in order to preserve transaction execution order across nodes and serve consistent data to reads. Google's implementation of the

TrueTime API keeps uncertainty small (generally less than 10ms) by using GPS and atomic clocks. Spanner shards data across data-centers into *directories*. Developers are responsible for describing one or more primary keys for each table and to assume join criteria between them. For example, suppose that every user has a set of photographic albums: if the developer specifies that the table $Albums$ will be joined with table $Users$ on $UserID$, then Spanner will gather users and albums in the same directory (sharding it by $UserID$) and interleave their rows in order to execute the query "select every album of a particular user" on a single directory, thus exploiting data locality. Thanks to data sharding, Spanner scales with the number of nodes. Spanner uses two-phase commit (over Paxos to mitigate availability issues) to agree on the results of distributed transactions. A closely related system, even if not distributed, is S-Store. It defines stream processing capabilities on top of a DBMS (H-Store), implementing streams as time-varying tables and stream processing as triggers [33]. In S-Store, a transaction is a directed acyclic graph of stored procedures calls that can access the whole underlying database. Instead, our approach limits the scope of transactions to individual operators (and, thus, fractions of the global application state) to exploit data locality and enable intra-transactional parallelism.

TSpoon can be considered a NewSQL system with a strong inclination for real-time queries. As a final remark, note that NewSQL systems require the developer to provide them with hints on how to organize data, in order to exploit data locality at its maximum degree and fully use the power of the cluster. As a consequence, these systems loose performance when executing transactions that are disrespectful of how data is organized. For example, if some transaction spans multiple nodes, it forces the system coordination in order to agree on its result, thus causing overhead in communication and increasing latency and, in most cases, decreasing throughput. Lastly, NewSQL systems focus on data storage and querying; while they can, in principle, perform stream processing at each node in the cluster, they lack both the flexibility in expressing generally complex graphs of computation, and the specific tailoring for parallel computation of SPs.

Related, although orthogonal, to our work in DBMSs and SPs integration are systems that unify OnLine Transaction Processing (OLTP) and OnLine Analytic Processing (OLAP). OLTP systems answer to external user requests in a reasonable amount of time (i.e., traditional DBMSs); while OLAP systems analyze the historical updates gathered from OLTP systems for business intelligence. "What was the total revenue of each of stores in January 2002?", or "how many more cars than usual did the company sell during that period?", or "which brand of phone covers is most often purchased together with that smartphone?" are examples of common OLAP queries. While SPs focus on real-time computation and transformation of dynamic streams of data, OLAP systems focus on interactive queries on big chunks of static, historical data. The different requirements of OLTP systems – random-access, low-latency writes from user input, and fast reads on a small amount of keys– and OLAP ones –computing aggregates over a large number of records– led to different architectures of the underlying storage layer –row-oriented for OLTP and columnar for OLAP– in order to achieve better data compression, indexing, and performance. L-Store [70] provides an integrated solution for OLTP and OLAP by employing a native multi-version, columnar storage model in order to lazily and independently stage stable data from a write-optimized layout into

a read-optimized columnar layout. SAP [45] provides a unified solution by gluing together different pre-existing OLTP and OLAP engines. IBM's Wildfire [18] handles hybrid transactional (only at *snapshot isolation* level of isolation [22][6]) and analytical workloads by enhancing Spark with OLTP capabilities, thus making it possible to exploit Spark streaming [86] to process transactions results in real time. However, both SAP and Wildfire do not support distributed transactions. To the best of our knowledge, TSpoon is the first distributed system that supports multi-partition ACID transactions at serializable isolation level and real-time stream processing.

TSpoon exploits the organization of operators in the graph of computation to achieve inter- and intra-transaction parallelism for higher performance. Related to transaction parallelism, the seminal works on nested and multi-level transactions [65, 80, 81] provide an extended transaction model that decompose transactions into *subtransactions* for concurrent execution, and prove isolation and recovery properties for this new model. Interestingly, new trends in DBMS employ this concept and provide the developer with well known primitives to express transaction composition and nesting: Actor-Oriented Database Systems (AODB) model the database and its transactions as actor-oriented programs where each actor manages a portion of the global state and embeds subtasks of the transaction logic [23, 74]. They provide low-level primitives to express intra-transaction parallelism explicitly using a mixture of SQL statements and asynchronous messaging between actors. We achieve a similar goal by translating transactional operations onto the graph of computation, although we do not provide the same flexibility as actors in the communication between the operators in the graph. We believe that the mixture of SQL and asynchronous messaging hinders the code maintainability and demands for multiple specific skills to database administrators. We do believe that our approach of modeling transactions on top of the graph of computation with simple data structures as internal state could be explored as a valid alternative in the future.

Some data processing architectures aim to solve the dichotomy between consistent state management and low-latency stream processing. The Lambda Architecture provides low-latency results, but serves exact yet "old" results in case of failures [63]. It was conceived when SPs did not provide full support for distributed, fault-tolerant, and stateful computation and where used as a fast *speed layer* that could potentially provide wrong results in the case of failures. Thus the speed layer is coupled with a *batch layer* that runs periodic batch jobs to generate higher-latency but exact results. When the data is queried, the *serving layer* encapsulates the complex logic that integrates the results of the speed layer (recent, but possibly inaccurate) or those of the batch layer (accurate, but possibly outdated).

More recent proposals criticize the complexity of this architecture and advocate stream-only solutions[7], where every event in the application is stored in streams, and its services consume events, apply their business logic, expose their internal state to external queries, and produce events. The strength of this approach lies in its opportunities for modularity and composability: one can add a new service and start processing events without affecting the overall architecture. The key enabler for this architecture is a fault-tolerant, scalable, distributed stream storage system like Kafka [79]. Persisted

[6]Snapshot isolation is a lower isolation level than serializable.

[7]`http://milinda.pathirage.org/kappa-architecture.com/`

streams, indeed, offer decoupling between services and can be replayed at will from the past. For instance, a new module of the application can be added to a system under execution and consume historical events to keep up with the current state of computation, or the output of a new version of a module can be compared to existing ones for A/B testing. However, flexibility comes with the downside of lack of coordination among services, unmanaged state, and absence of unified time-dependent operations. Pure SPs offer these guarantees, but lack of flexibility to perform general-purpose application logic and queryable state. Recently, some SPs introduce this feature to provide fast and dirty reads for rapid insights on the progress of computation [27]. Remarkably, they focus on individual operators without providing transactional guarantees as we do with TSpoon. Our proposal moves transactional updates directly on the SP and it can be considered as an evolution towards stream-only architectures.

CHAPTER 3

# SPs Execution Semantics

This chapter describes our approach in assessing SECRET's [25] adequacy in modeling modern distributed SPs. Section 3.1 provides an overview of the SPs computational model and describes the SECRET model. In Section 3.2 we model five modern distributed SPs by using SECRET and describe our methodology. Based on the results obtained from the modeling, Section 3.3 highlights the aspects of modern SPs that SECRET does not capture and proposes promising future directions for new models.

The reference paper for this chapter is [5].

## 3.1 Background

### 3.1.1 The SPs Computational Model

SPs provide abstractions to operate on dynamic datasets and produce new results on the fly as new input data becomes available. SPs take in input one or more *streams*, which are append-only unbounded sequences of data, and produce output streams as a result of their computation.

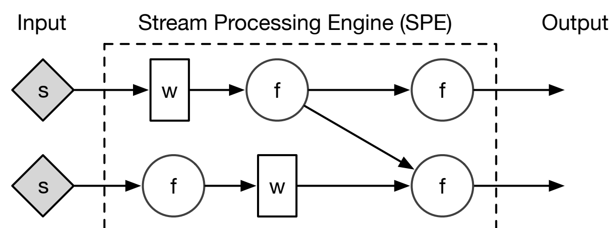



**Figure 3.1:** *The architecture of an SP*

The abstract architecture of an SP is presented in Figure 3.1. SPs receive input streams from one or more sources —grey diamonds in Figure 3.1— and organize the computation into a directed graph of operators —white circles and boxes in Figure 3.1— either explicitly or implicitly. In the latter case, developers are provided with high-level languages that are automatically translated by the SP into the operator graph. Some systems adopt functional languages and provide composable functions —such as `map`, `filter`, `reduce`, etc.— that transform input streams into output streams. Other systems adopt declarative, SQL-like languages that represent the processing as queries that get repeatedly and continuously evaluated as new data becomes available. Modern SPs take advantage of cluster architectures and deploy the operators in the graph on different cluster nodes, possibly replicating them. Organizing the computation into separate operators enables for task parallelism —different operators run on different threads on the same machine, or on different machines— while replication enables for data parallelism —different portions of an input stream are processed in parallel on different instances of the same operator.

In this chapter we separate operators in two classes: processing operators and windows. *Processing operators* apply a function to each and every element of their input streams. For instance, a filter operator selects or discards input elements based on a user-defined predicate, and a map operator converts each element of the input stream into an element of the output stream based on a user-defined function. We represent them in Figure 3.1 as white circles.

*Windows* enable computations that would be otherwise impossible due to the unbounded nature of streams. For instance, the average value of a stream of integers could only be computed after reading the *entire* stream, which is impossible since the stream never terminates. Windows obviate this problem by isolating the portions of the input streams upon which the function embedded within an operator should be applied [14]. We represent them in Figure 3.1 as white boxes.

Windows capture contiguous stream elements and are defined in terms of two parameters: a *size* that indicates the length of each window, and a *slide* that determines the interval between two consecutive windows. *Count windows* define the size and the slide based on number of elements. *Time windows* define the size and the slide based on some notion of order —*time*— between the elements in their input stream.

The behavior of time windows also depends on the notion of time, and different SPs adopt different time semantics. Using the terminology of Akidau et al. [9], we say that an SP adopts *processing time* semantics when each operator considers the clock time of the physical machine it is running on. In this case, elements do not have an attached timestamp and their mutual order is implicitly defined by the order in which they enter the operator. Processing time semantics leads to non-determinism, since the behavior of the system depends on the speed at which elements enter the engine and on the speed at which the elements flow between operators inside the engine, which is not under direct control of the developer. For these reasons, processing time has been often subject to critics, especially when considering distributed deployments of operators [9]. Nevertheless, some modern SPs adopt processing time, under the implicit assumption that the clock skew between the physical machines where the SP is running is negligible for the application at hand, and that the operators do not introduce significant delays.

We say that an SP adopts *ingestion time* semantics when each input element is times-

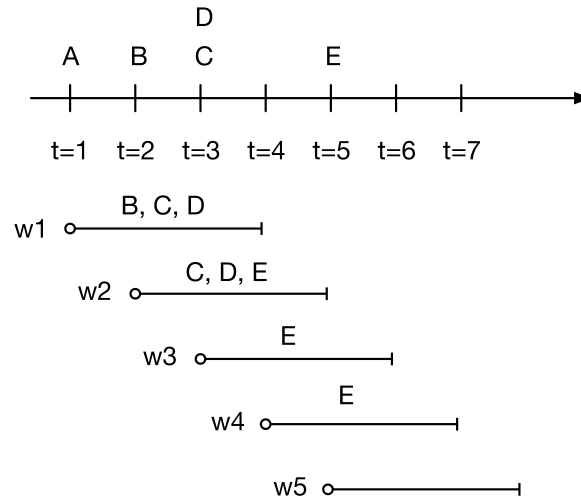



**Figure 3.2:** *Example of time window with size 3 and slide 1*

tamped when it first enters the engine, assuming that a single machine receives and timestamps all the input elements. Each time-dependent operator refers to such timestamp during evaluation. Given a specific timing of arrival of input elements, ingestion time ensures that the engine produces deterministic results.

We say that an SP adopts *event time* semantics when each element is timestamped by the source that produces that element, outside the SP. In the presence of event time, operators might receive elements out of order, due to clock skew between sources or due to network latency. In this case, the operators temporarily buffer the input elements and reorder them before processing. If the engine waits for enough time to receive all out of order elements —that is, if no element is lost— then event time ensures determinism.

To better explain the notion of windows, Figure 3.2 shows the effect of a time window of size 3 and slide 1 over an input stream. Stream elements are identified by upper case letters and ordered based on their timestamp —event time semantics. Notice that event time defines a partial order: for instance, elements C and D both have timestamp $t = 3$. Let us consider windows starting from $t \geq 1$. Window w1 includes all the elements from $t = 1$ excluded to $t = 4$ included, that is B, C, and D. Window w2 includes C, D, and E. Time windows can contain a heterogeneous number of elements, as in the case of w1 and w3. Furthermore, they might be empty, as in the case of w5.

In general, the results produced by an SP depend on (i) the topology of the operator graph; (ii) the functions implemented in each operator; (iii) the semantics of windows and time.

The first two elements are application-specific: they are defined by developers and are not system dependent. Instead, the semantics of windows and time varies from system to system and thus is the key element to capture the differences between the available SPs. The SECRET model we adopt in this work focuses on the semantics of windows in the case of event time.

### 3.1.2 SECRET

SECRET [25] models both time windows and count windows and builds on the following assumptions: (i) stream elements have an associated timestamp —event time semantics— that defines a partial order between elements; (ii) each operator reorders the input elements and processes them in timestamp order.

SECRET defines the semantics of windows by introducing the concepts (functions) of $\mathcal{S}$cope, $\mathcal{C}$ontent, $\mathcal{R}$eport, and $\mathcal{T}$ick. For ease of explanation, we present the SECRET formalism with reference to time windows, and we briefly discuss the differences in the case of count windows at the end of the section. The interested reader can refer to complete formalization of SECRET for further details [42].

As discussed in Section 3.1.1, each window operator is characterized by a size and a slide, and splits the input stream into windows all having the same size. SECRET defines windows as an interval $(t_o, t_c]$ where $t_o$ is the start time (excluded from the window) and $t_c$ is the end time (included in the window). A window is *open* at time $t$ if $t_o < t \leq t_c$. A window is *closed* at time $t$ if $t > t_c$. A window $w$ contains an element $e$ of an input stream if the timestamp of $e$ is within the boundaries of $w$.

$\mathcal{S}$cope is a function that maps each point in time $t$ to its *active window*, which is the open window $w$ with the lowest $t_o$[1]. $\mathcal{S}$cope only depends on the size and slide of the window operator, and on the start time of very first window —$t_0$—, which is system specific.

$\mathcal{C}$ontent is a function that maps each point in time $t$ to the stream elements that are in the active window at $t$.

$\mathcal{R}$eport is a function that defines the strategies that a window adopts to make its $\mathcal{C}$ontent visible for downstream processing. A window can adopt any combination —conjunction— of the following four strategies: (i) *content change*: $w$ reports only if the window content changes (with respect to the previous report); (ii) *window close*: $w$ reports only when the active window closes; (iii) *non-empty content*: $w$ reports only if the active window is not empty; (iv) *periodic*: $w$ reports only at regular time intervals.

With reference to Figure 3.2, an SP that reports on *content change* would not report the window `w4`, since its content is identical to the content of the previous window `w3`. Similarly, an SP that reports on *non-empty content* would not report window `w5`, since it does not contain any element.

$\mathcal{T}$ick is a function that describes the conditions that trigger a possible $\mathcal{R}$eport. SECRET identifies the following alternative $\mathcal{T}$ick strategies: (i) *tuple-driven*: each incoming element triggers an evaluation; (ii) *time-driven*: each time progress triggers an evaluation.

With reference to Figure 3.2, an SP that adopts a *tuple-driven* $\mathcal{T}$ick would evaluate the $\mathcal{R}$eport conditions twice for time $t = 3$, since two elements have timestamp $t = 3$. Conversely, a *time-driven* $\mathcal{T}$ick would trigger a single evaluation.

The case of count windows is analogous, with the difference that the size and slide of windows are defined in terms of number of elements rather than time. SECRET identifies each element of the input stream with a unique *id* and defines a global order between the stream elements based on their *id*. In the case of count windows, the $\mathcal{S}$cope

[1]Since a window operator is defined in terms of a slide that is greater than zero, at any given point in time the active window for that operator is unique.

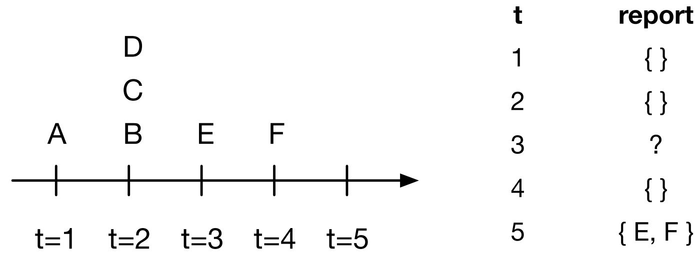



**Figure 3.3:** *Example of* time-driven $\mathcal{T}$*ick: count window with size 2 and slide 2*

function is defined in terms of the parameter $i_0$ —instead of $t_0$— which identifies the first *id* of the very first count window in the engine.[2]

Notice that a corner case can occur in the case of *time-driven* $\mathcal{T}$ick associated to count windows. Indeed, multiple windows might close at the same point in time, leaving to the SP the choice of which of them to report. Figure 3.3 exemplifies this situation for a count window with a size of two and a slide of two. Figure 3.3 denotes stream elements using upper case letters and orders them based on their event time. Element $\mathtt{A}$ has timestamp 1, and elements $\mathtt{B}$, $\mathtt{C}$, and $\mathtt{D}$ all have timestamp 2.

At time $t = 1$, no window closes and so the engine does not report. The same occurs at time $t = 2$. When the engine receives element $\mathtt{E}$ at time $t = 3$, all the elements for time $t < 3$ have been received —SECRET assumes that elements are processed in order. Thus, the system can process the elements $\mathtt{A}$, $\mathtt{B}$, $\mathtt{C}$, and $\mathtt{D}$, which fill two different windows of size two. Since the choice of the windows to report is engine specific, SECRET models this case by introducing a new $\mathcal{P}$ick function that encodes the selection.

As a final note, we observe that modern SPs consider windows as special operators in their processing graph and can use multiple window operators in the same graph — see Figure 3.1. On the other hand, SECRET focuses on windows and does not consider processing operators. Thus we model each window in isolation. We believe that this is not a severe limitation, since the overall semantics of a processing graph can be defined through the composition of its individual operators, and window operators are the most critical as they define the data slices upon which other operators get executed. At the same time, a complete modeling of the execution semantics of modern SPs should also take into account the topology of the processing graph. We defer a detailed discussion of this and other possible limitations of the SECRET model to Section 3.3.

## 3.2 Analysis of Stream Processing Engines

This section models the execution semantics of five widely adopted modern SPs using SECRET. We conduct an empirical analysis to determine the value of the parameters of SECRET —$\mathcal{S}$cope, $\mathcal{C}$ontent, $\mathcal{R}$eport, $\mathcal{T}$ick, $\mathcal{P}$ick— for each of the systems under analysis. The remainder of this section first presents the experimental methodology and then the systems under analysis and the results of our experiments for each of them.

[2]In the remainder of the thesis, we assume that the *id* associated to the very first element that enters the SP after it starts has value 0. The *id* is increased by one for each incoming element.

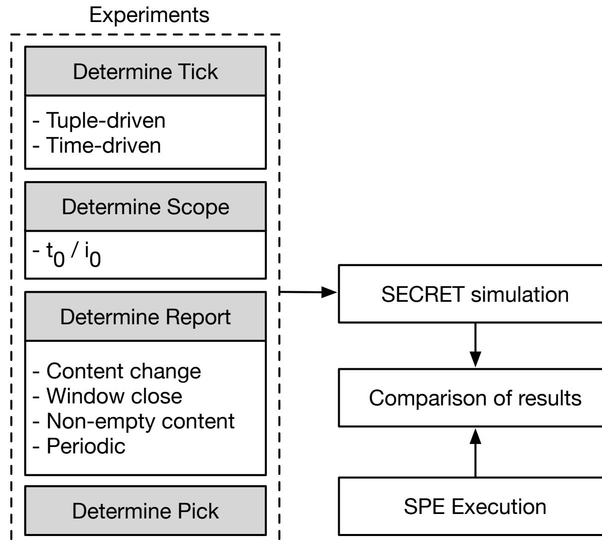



**Figure 3.4:** *Workflow of the experiments*

### 3.2.1 Experimental methodology

We devise an experimental methodology to determine the parameters of the SECRET model for the systems under analysis. We build minimal processing graphs that include a single source and a single window operator $w$ that outputs the entire content of $w$. We observe the time and content of each report from $w$.

Figure 3.4 presents the workflow defined in our methodology: the dashed block on the left of the figure represents the empirical experiments we perform to determine the functions $\mathcal{R}$eport, $\mathcal{T}$ick, $\mathcal{S}$cope, and $\mathcal{P}$ick. For each function, we list the concrete parameters that we need to identify, if any.

The behavior of $\mathcal{R}$eport depends on four independent flags: *content change*, *window close*, *non-empty content*, and *periodic*, which can be either active or inactive. $\mathcal{T}$ick can be either *tuple-driven* or *time-driven*. $\mathcal{S}$cope only depends on the parameter $t_0$ —in the case of time windows— or $i_0$ —in the case of count windows. $\mathcal{P}$ick can be any function that takes a set $s$ of windows and returns a subset of $s$.

To validate the correctness of our modeling, we rely on a simulator implemented by the authors of SECRET that we extend to capture the SPs under analysis. For a given input and for given $\mathcal{R}$eport, $\mathcal{T}$ick, $\mathcal{S}$cope, and $\mathcal{P}$ick functions, the simulator generates the expected output. We compare the results of the simulator with the real results produced by each SP using the same topologies and input data defined in the SECRET paper [25], which exercise the corner cases of the modeled behavior. We made the code for the deployment and execution of all the experiments publicly available[3]. The entire codebase consists of about 3000 lines of code, mostly written in Java and including the original SECRET simulator. Plugging a new SP only requires adapting the available examples to the API of that specific SP. We hope this will promote future extensions to other SPs.

We now proceed to describe the experiments we performed in detail. Unless otherwise specified in the description of the specific SP, we deploy the SP under analysis

[3] `https://github.com/deib-polimi/spes`

and the source of input data on the same machine. We configure the SP as a stand alone component and we use network communication to interact with it. We submit input elements in timestamp order, always using FIFO channels to avoid reordering. SECRET does not capture problems related to clock skews and out-of-order elements that might emerge in distributed settings. Thus, these issues are outside the scope of the analysis and will be better discussed in Section 3.3.

We adopt input elements in the form $\langle val, ts \rangle$, where $val$ is a string value and $ts$ is the timestamp of the element expressed in seconds. We design the experiments in such a way that timestamps well approximate the physical time measured on the wall clock time at the source. To do so, we pause the source between the submission of two consecutive elements $e_1 = \langle val_1, ts_1 \rangle$ and $e_2 = \langle val_2, ts_2 \rangle$ for $ts_2 - ts_1$ seconds. We keep the input rate low enough to guarantee that the SP is never overloaded. Finally, we assume the variance of the latency between the source and the SP to be negligible. Under these conditions, event time approximates well the processing time, thus making it possible to use SECRET also to analyze systems that adopt processing time semantics.

We discuss how we determine the $\mathcal{T}$ick, $\mathcal{R}$eport, and $\mathcal{S}$cope functions in the case of time windows, being the procedure for count windows analogous.

**Determine $\mathcal{T}$ick.** To determine the $\mathcal{T}$ick, we define a time window of size $\omega > 1$ and we submit more than $\omega$ elements all having the same timestamp. If the SP reports more than once, then the $\mathcal{T}$ick is tuple-driven, meaning that the window advances even with elements that have the same timestamp, otherwise the $\mathcal{T}$ick is time-driven.

**Determine $\mathcal{R}$eport.** The semantics of report depends on four flags, and the SP reports if (the constraints expressed in) *all* the flags are satisfied. We determine whether a flag $f$ is satisfied by conducting experiments in which the results only depend on the satisfiability of $f$.

First, we build an experiment where *content change* and *non-empty content* do not influence the results. To do so, we submit elements with increasing timestamps, where the distance between the timestamp of two consecutive elements is a fixed value $\tau$. We set a window of size $\tau$ and slide $\tau$ and we observe whether the system reports at every new tuple or only periodically, thus determining the value of the *periodic* flag. Next, we increase the size of the window to understand if the SP reports only on *window close*. Second, we build an experiment in which the content of the window does not change across subsequent ticks, to determine the value of *content change*. As we discuss in the following, none of the systems under analysis exhibits periodic behaviors and none of them reports only on *content change*. Finally, we consider empty windows and analyze the reports of the SP to determine the value of the *non-empty content* flag.

**Determine $\mathcal{S}$cope.** The $\mathcal{S}$cope function depends on the parameter $t_0$, which is the start time of the very first window considered by the SP. Let us call $ts_i$ the timestamp of the first element submitted to the SP, $\omega$ the window size, and $\beta$ the window slide. To understand the value of $t_0$, we submit elements to the SP with a predefined frequency and we consider different combinations of $\omega$, $\beta$, and $ts_i$. We use the results we obtain to infer the value of $t_0$ as a function of $\omega$, $\beta$, and $ts_i$. In the case of count windows, we consider the parameter $id_i$ instead of $ts_i$, which is the id of the very first element that enters the engine.

**Determine $\mathcal{P}$ick.** The $\mathcal{P}$ick function is only relevant in the case of count windows

**Time windows**

| | $\mathcal{T}$**ick** | **$\mathcal{R}$eport** | | | | $\mathcal{S}$**cope** |
|---|---|---|---|---|---|---|
| | | **CC** | **WC** | **NE** | **P** | |
| **Flink** | time | | ✓ | ✓ | | $t_0 = ts_i + \beta - \omega - 1$ |
| **Storm** | time | | ✓ | ✓ | | $t_0 = ts_i + \beta - \omega$ |
| **Spark** | time | | ✓ | | | $t_0 = \beta - \omega - 1$ |
| **DataFlow** | time | | ✓ | ✓ | | $t_0 = ts_i + \beta - \omega - 1$ |
| **Azure S.A.** | time | | ✓ | ✓ | | ? |

**Count windows**

| | $\mathcal{T}$**ick** | **$\mathcal{R}$eport** | | | | $\mathcal{S}$**cope** | $\mathcal{P}$**ick** |
|---|---|---|---|---|---|---|---|
| | | **CC** | **WC** | **NE** | **P** | | |
| **Flink** | tuple | | ✓ | ✓ | | $i_0 = id_i + \beta - \omega - 1$ | – |
| **Storm** | tuple | | ✓ | ✓ | | $i_0 = id_i + \beta - \omega - 1$ | – |
| **Spark 1.6** | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. |
| **DataFlow** | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. |
| **Azure S.A.** | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. |

**Table 3.1:** *Parameters identified for the SPs under analysis*

if the SP under analysis provides time-based $\mathcal{T}$ick. As explained in Section 3.1.2, this situation might lead to multiple windows closing simultaneously, and the $\mathcal{P}$ick function selects which of them to $\mathcal{R}$eport.

Most of the systems under analysis do not support count windows, and when count windows are available, they are always associated to a tuple-driven $\mathcal{T}$ick. Thus, the $\mathcal{P}$ick function is never used in the context of our analysis.

### 3.2.2 Flink

Flink [29] is an Apache project built on the Stratosphere research prototype [10] developed at the Technische Universität of Berlin. Flink has gained popularity in the last few years, becoming Apache Top Level Project in March 2014. It provides both a DataSet and a DataStream API to process static and streaming data, respectively. It also offers a number of libraries built on top of such APIs for disparate application domains, such as machine learning and graph processing.

We consider Flink $1.1.4$ and we target the DataStream API. Flink processes streams of data using event time semantics. Time-related operators such as time-based windows use a *watermarking* mechanism [8] to reorder input elements before processing: an operator $o$ sends a watermark with timestamp $t$ to another operator $o'$ to guarantee that $o'$ will not receive from $o$ any more items with timestamp lower or equal than $t$. When an operator receives a watermark with timestamp $t$ from all its upstream operators, it can safely start processing all the elements up to time $t$.

Flink offers exactly once semantics even in the presence of node failures and implements fault tolerance using a distributed snapshot algorithm [71].

To model Flink in SECRET, we follow the methodology presented in Section 3.2.1, using a single source connected to a single window operator through a socket. Flink exposes some configuration parameters other than window size and slide to tune the behavior of time windows. For instance, developers can force Flink to implement a processing time semantics as opposed to the default event time semantics. In our experiments, we always adopt the default parameters and we derive the SECRET model accordingly. We defer to Section 3.3 a detailed discussion of the implications of such parameters.

In Flink, each operator is responsible for providing the watermark to all its downstream operators. In particular, each source needs to submit a new watermark upon the

delivery of a new element or periodically. In our experiments, we ingest input elements in event time order, so that the engine needs not wait for late, out of order elements. We make the source submit periodic watermarks with a frequency that is higher than the input rate. Each watermark is set to the timestamp of the last emitted element minus one second. This enables us to submit multiple elements with the same timestamp, for instance to determine the value of $\mathcal{T}$ick. All the elements with a timestamp $\tau$ are immediately processed when an element with timestamp greater than $\tau$ (and the corresponding watermark) is received, without waiting for out-of-order elements.

Table 3.1 lists the parameters of SECRET for Flink. In the case of time windows, the $\mathcal{T}$ick is time-driven and Flink reports on *window close* with *non-empty content*. The first window closes at time $ts_i + \beta - 1$, which means that it opens at time $t_0 = ts_i + \beta - \omega - 1$. In the case of count windows, Flink changes its $\mathcal{T}$ick to tuple-driven. As a consequence, it never experiences windows closing simultaneously and so it does not need to implement the $\mathcal{P}$ick function. The value of $i_0 = id_0 + \beta - \omega - 1$ is analogous to the $t_0$ extracted for time windows. Under the assumption that $t_0 = 0$ and $i_0 = 0$ the formulas for $t_0$ and $i_0$ can be simplified to $t_0 = i_0 = \beta - \omega - 1$[4].

As a side note, we report that the behavior of Flink might differ from the behavior of the SECRET model when considering the very last active window before the system shuts down. Indeed, Flink flushes the active window when the system terminates, but this behavior is not captured by SECRET, which models the SP assuming a steady state.

### 3.2.3 Storm

Storm [78] is a framework for distributed real-time computation acquired by Twitter in 2011 and released as an open source Apache project. In Storm, developers specify the behavior of individual operators using imperative code and connect operators with each other to form a direct acyclic graph. Starting from version 1.0, Storm also offers a native support for time windows, which are the subject of our model. Each Storm operator processes stream elements one at a time and implements fault tolerance using a per-element acknowledgement and resubmission. This ensures that each operator processes each input element *at least once*, but does not guarantee *exactly once* processing. This aspect might affect the semantics of processing in the case of faults, but it is not modeled in SECRET. Thus, in the case of failures, the results produced by Storm might differ from those produced by its SECRET model.

We performed our experiments using Storm version 1.0.2. We implemented a topology with a single source and a single window operator connected through a TCP socket. Since Storm supports event time semantics and handles out of order arrival of elements through watermarking, we adopted the same strategy for the submission of watermarks as in Flink.

Table 3.1 lists the parameters to model Storm in SECRET. In the case of time windows, Storm presents the same $\mathcal{T}$ick and $\mathcal{R}$eport as Flink: it has a time-based $\mathcal{T}$ick and reports on *window close* with *non-empty content*. Storm's $t_0$ differs from Flink's $t_0$ by a fixed offset of one: while this difference might appear marginal, this means that the windowing behavior of Storm and Flink are different and thus the two engines produce different results when fed with identical input.

[4] SECRET assumes the domain of time and ids to be discrete. Timestamps and ids are allowed to be negative.

Initially, we started our experiments with Storm version 1.0.1. Using this version, we could not model count windows, since in the case of elements with identical timestamps Storm reported the last $\omega$ elements in the active window $\lfloor n/\omega \rfloor$ times, where $\omega$ is the size of the window and $n$ is the number of elements not-yet reported. We reported this suspicious behavior to developers that recognized it as a bug. Storm version 1.0.2 integrates a bug fix[5] and we complete Table 3.1 accordingly. The observed behavior is identical to that of Flink. The detection of this bug highlights the importance of a formal model to analyze the execution semantics of SPs. Thanks to the adoption of SECRET we could identify the presence of a misbehavior.

### 3.2.4 Spark

Spark [85] is a general-purpose cluster computing system. Initially developed at Berkeley and currently an Apache project, Spark is widely adopted in batch and stream processing applications that involve large volumes of data, also thanks to the availability of several application-specific libraries. Spark stores data in Resilient Distributed Datasets (RDDs), a read-only, lazily evaluated, partitioned collections of records. RDDs are persisted on a distributed file system for fault tolerance and can be cached in main memory to enable low-latency computations. Spark keeps track of how RDDs are computed. If an RDD is lost due to some failure, its value is recomputed through the RDDs it depends on. This guarantees exactly once semantics even in the presence of node failures.

Spark models streaming computations as a series of stateless, deterministic batch computations on small batches of data —$\mu$-batch. In particular, streams are discretized into a sequence of immutable, partitioned RDDs, which enable Spark to reuse the strategies and algorithms for batch computation also in streaming scenarios. This approach trades latency for throughput, since it delays a computation until a $\mu$-batch is available. The $\mu$-batch approach has also some consequences on the semantics of processing. Specifically, Spark only supports time windows having a size and slide that are multiple of the $\mu$-batch size. This technological constraint is outside the scope of SECRET and thus is not captured in our model.

More significantly, Spark only supports processing time semantics and not event time semantics. Each operator considers the wall clock time of the machine it is running on and thus the order and time distance of the elements in a stream also depend on the processing capabilities of that specific machine. This means that the same input might produce different results in different deployments of the same processing topology. As discussed in Section 3.2.1, we designed the source such that its submission time is a good approximation of the value of its timestamp. This enables us to use the SECRET model, which relies on event time, also in the case of processing time. However, it is worth mentioning that this approach only works if the system is not overloaded: if the system instead introduces some processing delay, event time and processing time might differ, making our modeling imprecise.

In our experiments, we adopt Spark version 2.1. We configure Spark to use the minimum size of $\mu$-batch of 1 ms and we connect a single source to the engine using a TCP socket. Spark only supports time windows and does not offer count windows. As shown in Table 3.1, Spark implements a time-based $\mathcal{T}$ick that reports on *window change* independently from the content of the window. The value of $t_0$ is different than

[5] `https://github.com/apache/storm/pull/1568`

in the other systems we analyze. Indeed, Spark closes the first window at time $\beta - 1$ after the SP starts up at time $\tau$ and also reports empty windows (thus the *non empty window* flag is not set).

SECRET cannot model the start up time $\tau$ of the system, since it can only reason about the event time as encoded in the timestamps of incoming elements, while $\tau$ can only be captured by considering the wall clock time of the machine in which the SP runs. To solve this issue, in our experiments we assume that the SP starts up at time 0, thus obtaining the value $t_0 = \beta - \omega - 1$. This highlights a limitation of SECRET that we better discuss in Section 3.3.

### 3.2.5 Google Cloud Dataflow

Google offers the Google Cloud Dataflow[6] SP as a service in the Google Cloud Platform. Google open-sourced the Software Development Kit for Dataflow, but not the underlying engine that remains proprietary.

Since Dataflow is only offered as a Cloud service, we had to adapt the experimental methodology. We communicate with the engine using the Pub/Sub messaging service[7] in the Google Cloud Platform, which offers multicast communication on distinct channels called *topics*. Specifically, we deploy a topology composed of four operators: (i) a Pub/Sub *subscriber* that reads elements from an input topic; (ii) a window operator; (iii) a *reducer* that concatenates the content of each window into an output string; (iv) the Pub/Sub *publisher* that writes the results of the reducer on an output topic. We submit elements by publishing them on the input topic and we read the results from the output topic.

By default, Dataflow applies event time semantics and adopts a watermarking mechanism to handle out of order. Watermarks are emitted by the very first operator that receives elements from outside the SP (the *subscriber* in our topology). During the experiments, we observed a dependency between the input and the output rates. In particular, a low input rate could lead to unbounded output latency. We believe that a dynamic management of watermarks determines this behavior: in presence of a low input rate, the *subscriber* assumes a high latency in the arrival of elements, and thus becomes more conservative in the values of watermarks. As a consequence the values of watermarks do not advance and the engine stops processing new elements and emitting new output. We overcome this problem by attaching a long tail of additional elements at the end of each experiment, which ensures that we eventually receive all the output of that experiment.

Table 3.1 shows the SECRET parameters for Dataflow. Dataflow only supports time windows and does not provide count windows. The $\mathcal{R}$eport is *not-empty content* and on *window close*, while its $\mathcal{T}$ick is time-based. The $\mathcal{S}$cope is the same as in Flink $t_0 = ts_i + \beta - \omega - 1$.

### 3.2.6 Azure Stream Analytics

Microsoft offers the Azure Stream Analytics SP[8] as a Cloud service. The communication with the SP can be realized using either a publish-subscribe service or a message

[6] https://cloud.google.com/dataflow/
[7] https://cloud.google.com/pubsub/
[8] https://azure.microsoft.com/en-us/services/stream-analytics/

**Listing 3.1:** *An example of Stream Analytics query.*

```
SELECT     System.Timestamp AS ts, Collect()
INTO       output-queue
FROM       input-hub TIMESTAMP BY tapp
GROUP BY   HoppingWindow(second, 4, 2)
```

queuing service. In our experiments we submit elements through publish-subscribe primitives and we read output elements from a queue.

In Azure Stream Analytics, the developer defines the processing tasks in the Stream Analytics Query Language, a SQL-like declarative language that is compiled to a graph of operators and deployed on the Cloud. Listing 3.1 shows the query we use in our experiments. The `GROUP BY` clause introduces a `HoppingWindow`, which is the Azure Stream Analytics implementation of a time based sliding window. The query extracts the timestamp of each element —stored in the `tapp` field— and returns the entire content of the window —`Collect()`— and the (event time) timestamp of the end of the window.

Azure Stream Analytics supports event time semantics, where timestamps represent time in UTC. Developers can set a maximum time skew for late arrivals. When an element $e$ enters the engine, the engine compares the ingestion time of $e$ with the timestamp of $e$. If the difference between the two times is larger than the maximum time skew, then the engine drops $e$. Since the maximum time skew cannot be larger than few days, we could not use the same timestamps adopted for the other SPs that start from 0. Instead, we shifted all the timestamp by $\delta$, where $\delta$ is the UTC time when the experiment started, as extracted from the wall clock time of our local machine. This solution enables us to extract the parameters for the Azure Stream Analytics for $\mathcal{T}$ick and $\mathcal{R}$eport. We were not able to extract the value of $t_0$ since it probably depends on the ingestion time of the first element, which we cannot access from the Cloud platform.

As Table 3.1 shows, Azure Stream Analytics only supports time based windows, reports on *window close* with *non-empty content*, and presents a time-based $\mathcal{T}$ick.

## 3.3 Discussion

This section presents the key conclusions we draw from the modeling effort reported in Section 3.2. Under some assumptions that we better discuss in the remainder of this section, SECRET captures the semantics of windows in the SPs we analyzed and highlights a general agreement on the $\mathcal{T}$ick and $\mathcal{R}$eport of windows, based on the following rules: (i) in the case of time windows, the $\mathcal{T}$ick is always time-driven; (ii) in the case of count windows (if available), the $\mathcal{T}$ick is always tuple-driven; (iii) all systems $\mathcal{R}$eport on *window close* and *non-empty content*, with the only exception of Spark that also reports empty windows.

Conversely, the systems under analysis present differences in terms of $\mathcal{S}$cope. In most systems the first active window closes after a slide $\beta - 1$ from the arrival of the first element, meaning that $t_0 = ts_i + \beta - \omega - 1$ ($i_o = id_i + \beta - \omega - 1$ in the case of count windows). Storm considers $t_0 = ts_i + \beta - \omega$, which is probably due to a different definition of windows that includes the open time and excludes the end time. Finally,

Spark adopts a processing time semantics, and the position of windows is not related to the timestamp of the first element, but rather to start up time of the SP.

While the different $\mathcal{R}$eport and $t_0$ in Spark and the different $t_0$ in Storm might appear small variations with respect to the behavior of the other SPs, they lead to different execution semantics and thus to different results. This motivates the need for a formal model to capture the execution semantics of modern SPs and highlight their differences.

Our analysis also sheds light on some aspects of modern distributed SPs that SECRET cannot fully model. They are of great interest since (i) they hint at the key differences between “old generation” SPs (those that SECRET was designed to model) and “modern” SPs for distributed processing in cluster environments; (ii) they suggest promising lines of investigation to build a more comprehensive model that captures all the relevant aspects of modern SPs. We describe them in detail in the next sections.

### 3.3.1 Time model

SECRET assumes event time semantics and further assumes that all the stream elements enter the engine in order with respect to their timestamps. These assumptions were motivated by the nature of the stream processing systems available when SECRET was conceived, which were mainly centralized and specifically designed for continuous query answering. Stream elements often encoded occurrences of noteworthy facts in the application environments and queries typically included time boundaries. Because of this, windows —and time windows in particular— were perhaps the most relevant operators for these systems.

Conversely, modern SPs such as the ones we consider in this paper are designed to perform generic computations on large volumes of streaming data. They do not consider time as a first class citizen and do not necessarily associate a timestamp to each and every stream element. In most cases, windowing constructs are not core building blocks of the engine, but rather operators developed on top of the base engine services to better support some application scenarios. This is for example the case of Storm, which offers windows only starting from recent versions.

Moving from these premises, it is not surprising that some systems, such as Spark, use processing time as their default time semantics. Under processing time semantics stream elements are processed by an operator in the order in which they enter that operator. The engine does not assume elements to be timestamped and thus it cannot exploit timestamps to learn the semantics of time of the application at hand.

The execution semantics of a single windowing operator under processing time can be approximated in event time if the following conditions hold: (i) The order in which the elements enter the SP reflects the desired order for the application at hand. (ii) The distance between the arrival of elements in processing time reflects their distance in event time; for instance, if the source that emits elements and the windowing operator are deployed on different physical machines, this means that the variance of the network latency between them is negligible. (iii) The physical node that executes the window operator is not overloaded, and thus it does not introduce additional delay in the processing time.

The second and the third assumptions are the most critical, since they depend on runtime behaviors —processing speed, load, and network latency— that are not under the control of developers and difficult or impossible to achieve, as in the case of constant

network latency.

As for the first assumption, it is reasonable to assume that individual external sources submit elements in some meaningful order, but without timestamps it is impossible to define a total order between elements produced at different external sources. If the developer wants to ensure event time semantics on top of a system that does not support it, she has to manually implement the application logic responsible for buffering and reordering input elements before processing.

### 3.3.2 Windowing approaches

SECRET models tumbling and sliding windows. However, those are not the only windowing strategies available. The size of some type of windows is not fixed (let it be a time span, or a number of elements), but it depends on their content. We provide below some examples of different windowing behavior gathered from the state of the art.

Cutty [30] is a framework for efficient aggregate sharing across overlapping windows. Interestingly, it provides a classification of windows in deterministic and non-deterministic ones. Essentially, deterministic windows are the ones for which, when a new element occurs, it can be associated immediately to a window. Periodic windows (tumbling and sliding) are deterministic. For non-deterministic ones, on the contrary, it is impossible to state if a record is in a window or not without further information. As an example, consider a window that contains the last record in a stream every $5$ seconds: when record $r$ occurs at time $t$, it is impossible to know in advance if no record will occur before the end of the window; in other words, it is impossible to state if $r$ is in the window until the window period expires. Li et al. [57] proposed a similar approach to define windows based on the content of input elements. They adopt this formalism to define an effective evaluation strategy for window aggregates.

Google Dataflow and Apache Flink introduced the concept of session windows [9], in which the boundaries of each window are defined based on the frequency of the input elements. A session window is considered closed when there is a user-defined gap of inactivity between elements.

Frames [50] are content-defined windows that provide developers with built-in functions to simplify the statistical analysis of data.

Predicate windows [48] predicate on the content of an input element to determine whether it has to be considered as new information, or as an update (or deletion) of existing information. They were introduced to define views and to support view maintenance in DSMSs. Predicate windows are more flexible since they are not append-only, but they enable incoming elements to overwrite part of a window content.

Bugra Gedik [47] provides a unified framework to define multiple types of windows based on their *insertion*, *trigger*, and *eviction* policies: *insertion* represents the addition of an element into its respective window; *eviction* represents removal of one or more of the oldest elements from the window; *trigger* represents the application of the operator logic to the content of the window. These policies can be applied in different orders and can be composed (from a specification of standard policies provided by the author) in order to implement user-defined windowing behavior.

In conclusion, the space of windowing strategies is much wider than only tumbling and sliding. The advent of new types of windows and user-defined ones will demand for novel models that can extend or complement SECRET.

### 3.3.3 Management of out-of-order elements

As discussed in the previous section, SECRET assumes input elements to enter the engine in order with respect to their associated timestamp. Instead, modern SPs support out-of-order arrival of input elements, but differ in the way they manage such elements.

A common approach to deal with out of order consists in specifying the maximum delay for the arrival of elements and defer the processing until such a delay has elapsed. Yet, the semantics of processing in the case some element overcomes such maximum delay changes from system to system. For instance, as discussed in Section 3.2, Azure Stream Analytics discards input elements that arrive too late with respect to the UTC wall clock time. These behaviors cannot be captured within the SECRET model.

Another approach consists in producing metadata at the sources to indicate when the produced elements can be safely processed without incurring the risk of late arrival of further out of order elements. The watermarks adopted in Flink are a concrete implementation of this more flexible approach [8] that enables each source to dynamically adjust the metadata it provides to the engine based on its current operating conditions. Also in this case, the semantics of processing in the case some elements violate the content of metadata might change from system to system.

Finally, other systems produce results in the form of mutable, time-annotated datasets. In the presence of out-of-order elements that alter the values of some results produced in the past, the engine retracts the previous output from the mutable dataset and substitutes it with the newly computed values. This is the case of the Kafka Stream system [79].

### 3.3.4 Graph of operators

SECRET considers the semantics of windows in isolation. This is motivated by the default processing model of the SPs SECRET was designed to capture, which typically consists of a predefined, fixed structure with three steps [12]: (i) a stream-to-relation step that uses windows to select portions of the input stream; (ii) a relation-to-relation step that performs the actual processing by only considering the content of windows; (iii) a relation-to-stream step that converts back the results of the computation into stream elements. This fixed structure well serves the purpose of performing continuous queries on streaming data, but it is not flexible enough for general purpose computations.

Conversely, modern SPs enable developers to build complex graphs of operators. How the structure of these graphs influences the execution semantics is outside the scope of SECRET and certainly relevant to fully model modern SPs. For instance, some SPs admit cyclic topologies, but the way in which they implement cycles might differ. In general, the execution semantics might change depending on the way SPs route elements between operators, since distributed deployments might affect the mutual order of stream elements as they move from operator to operator. Distribution becomes particularly relevant in the case of processing time semantics, where the presence of heterogeneous nodes, different processing speed, or different in latency between nodes might affect the overall behavior of the SP in ways that cannot be predicted by only looking at the graph topology.

Finally, in modern SPs windows are not special entities, but rather one of the possi-

ble operators that compose the processing graph. As a consequence, it becomes worth to investigate how different windows interact with each other based on their location in the graph.

### 3.3.5 Fault tolerance

Modern SPs are designed to run on a multitude of physical nodes. In this setting, the probability of failure of at least one node is not negligible. Thus, SPs include fault tolerance mechanisms to keep processing and producing results even in the presence of some failures.

Such mechanisms include per-element acknowledgments and retransmissions as in Storm, lineage graph and recomputations as in Spark, and distributed snapshots as in Flink. Most significantly, different mechanisms provide different semantics and yield different results in the presence of failures. For instance, both Flink and Spark guarantee that each element is processed at each operator exactly once, which means that the results produced do not change in the case of failure. Conversely, Storm only guarantees at least once semantics, meaning that each operator can submit an element more than once to its downstream operators. Clearly, these differences affect the results produced by the engine in presence of faults and should be captured by a complete model of the system.

Even in the case of exactly once semantics, fault tolerance mechanisms affect the time when a result is produced. In the case of event time semantics, individual elements include a timestamp that defines the order and time distance between elements. In the case of processing time semantics, elements do not embed any notion of time and so their original order and relative time distance is lost in the case of faults. In other words, the assumptions we made to model systems with processing time semantics in SECRET no longer hold in the case of faults.

### 3.3.6 Summary and open challenges

The above discussion highlights the need to extend and complement SECRET to build a comprehensive model that fully captures the execution semantics of modern SPs. Here we summarize the aspects of modern SPs that demand for further modeling efforts and propose interesting directions for future research in this area.

First of all, SECRET assumes event time semantics. In the case of a single window in which all the elements are received in order, and in the presence of non overloaded machines, event time well approximates processing time.

However, the assumption of in order arrival of events might be unrealistic in several real world scenarios in which the clocks of different sources are not well synchronized or the channels between the sources and the SP have different latency [9]. A proper analysis of this situation requires a model that takes into account the differences between the application time perceived at the sources and the time when elements are processed within the SP.

Furthermore, under processing time semantics, the processing speed of the machines as well as temporary overloads might impact on the output produced. To capture these aspects, a model should take into account performance metrics. Given the intrinsic non-determinism of performance measurements, we foresee the adoption of probabilistic

models that encode the probability of the SP to deviate from a default expected output. In this case, the content of a window is the whole set of the elements in a stream (possibly infinite), each one with an associated probability of membership. Fuzzy sets are the mathematical abstraction that may be used to model a probabilistic $\mathcal{C}$ontent function [83].

Also, by solely relying on event time semantics, SECRET cannot model the aspects of an engine that depend on wall clock time. For instance, windows in Spark start to report after the engine starts up; similarly in Azure Stream Analytics, elements which time differs too much from the wall clock time of the physical machine get discarded. A comprehensive model that encodes both event time and wall clock time could also capture these behaviors.

Second, SECRET was designed to capture the semantics of individual windowing operators. This was motivated by the nature of the SPs SECRET was designed for: those SPs were mainly intended to answer continuous queries, where *a single* window was used to isolate the portion of the stream upon which the queries were evaluated. With well defined semantics for the queries, a precise modeling of the windowing behavior was then sufficient to capture the overall execution semantics of the engine.

Modern SPs are designed to solve a larger class of problems, and offer programming abstractions that are suitable to encode general streaming computations. In most cases, the developer does not specify the processing task in terms of a high-level declarative query language, but rather explicitly defines the graph of operators that the input elements traverse to produce the output.

Windows still represent the most critical operators to model, since they accumulate the portions of the stream upon which other operators are executed. Nevertheless, a complete model that fully captures the execution semantics of modern SPs needs to precisely encode the semantics of the operators graph formalism, defining its shape — for instance, whether loops are allowed— and behavior —for instance, how elements from multiple input streams are ordered within an operator.

Third, modern SPs are designed to run on multiple nodes without a shared memory. In most cases the distribution of processing is transparent and does not impact on the output of the SP. Yet in some cases the distribution might affect the execution semantics: for instance, we already discussed the consequences of heterogeneous processing capabilities under processing time semantics.

Thus, we believe that a precise model of SPs should also be concerned with details about the processing infrastructure and the deployment of the operators on the physical nodes. For instance, it should consider whether the SP enables the partitioning or the replication of some operators on multiple nodes to improve the performance, and how partitioning and replication might impact on the execution semantics of the engine. In this case, indeed, windowing operators can be replicated too, making it possible to have more then one window with the same open and close timestamp. SECRET's assumption of one *active window* could not be applicable in these cases. Furthermore, replicated windows are applied only to a partition of the original stream, thus definitely influencing the results of the $\mathcal{C}$ontent function.

Fourth, some modern SPs introduce new types of windows that SECRET cannot model. For instance, session, non-deterministic, predicate, or user-defined windows have boundaries not defined by the time or by the number of elements, but rather by

their content. For instance, in a web site monitoring scenario, a window might open upon receiving an element that indicates that a user started a particular operation and close when the operation ends. The $\mathcal{R}$eport function should be extended with additional policies in order to account for these new types of windows. User-defined windows represent a particular case, in that they impact both $\mathcal{R}$eport and $\mathcal{C}$ontent. The model provided by Bugra Gedik [47] is a good point to start in shaping them. The $\mathcal{C}$ontent function should be extended to accommodate for *insertion* and *eviction* policies, while $\mathcal{R}$eport should account for *trigger* policies.

Finally, modern SPs introduce fault tolerance mechanisms to cope with the rather frequent hardware failure in the cluster platforms in which they operate. Different SPs provide different guarantees in the presence of failures: in most cases, the SP guarantees exactly once semantics, meaning that no loss or duplicate processing are possible and hence failure do not affect the execution semantics. However, some platforms offer weaker guarantees. For instance, Storm offers at least once semantics, in which duplicate processing of some elements is allowed. In this case, failures impact on the execution semantics and need to be modeled. This demands for a model that not only contemplates the presence of loss or duplicate elements in one operator, but also how loss and duplicate elements affect the entire processing graph.

In conclusion, one could ask if it is really worthwhile to capture such behavior as part of the execution semantics of a system, or if it could be the case to classify them as faults. However, every aspect we analyzed so far constitutes part of the actual behavior of systems that are in use in many software stacks. If a model is distant from the actual behavior of such systems, then it should be updated in order to account for it. Justifying the need for modeling the graph of operators, operator replication, and content-based windows is straightforward, because these aspects are a must for distributed SPs, and significantly impact the semantics of the application at hand. Architectural aspects as processing time and fault tolerance could be left outside of the model. However, processing time is a processing mode that can be enabled in many systems and it is actually employed for use cases where the sources do not timestamp elements; and different fault tolerance guarantees are accepted modes of operation of distributed SPs. In general, if an architectural aspect impacts the accepted output of the system, then it should be included in the model.

CHAPTER 4

# Transactions on the Stream Processor

In this chapter, we describe our approach for unifying SPs and DBMSs under the same system. Section 4.1 provides background information for DBMSs and SPs state management capabilities. Section 4.2 describes the limitations of the SPs computational model in expressing transactional behavior. In Section 4.3 we model transactional behavior within the graph of computation of SPs. In Section 4.4 we describe the implementation of TSpoon and, finally, in Section 4.5, we evaluate its performance.

The reference paper for this chapter is [4].

## 4.1 State Management Capabilities

Before discussing in detail the limitations of SPs in Section 4.2, we compare the state management capabilities of DBMSs and SPs. As anticipated, our work combines the efforts of different communities: we extend the SP computational model with the state management that DBMSs offer. Section 4.1.1 describes the state management capabilities of databases. Section 4.1.2 provides background on the SPs computational model by summarizing the concepts already introduced in Section 3.1, although focusing on state management on the SP.

### 4.1.1 Database Management Systems

DBMSs are tailored to manage state. Their state management capabilities involve state queries and updates, state integrity, and fault tolerance:

**Queries and updates** the DBMS provides a language for querying and updating the state (e.g. SQL) and offers the abstraction of *transactions*[1], group of operations

[1] Many DBMSs have a restricted definition and scope for transactions [17, 32, 39], here, we describe the most general case.

on the state that are executed as one.

**Integrity** the DBMS guarantees that the state is kept consistent with the integrity constraint specified by the user. For instance, there cannot be two different data items that have some key, or, if a data item references another, it must exist —*referential integrity*. Moreover, the DBMS can enforce user-defined invariants on the state, such as "user balances cannot fall below 0$".

**Fault tolerance** in case of hardware/software failure, the DBMS must guarantee to recover to a consistent state.

The transaction abstraction is of paramount importance for users in order to implement queries and updates in isolation, without thinking of the nitty-gritty details about what could go wrong if some transaction is executed in parallel with another one. For performance reasons, indeed, DBMSs execute transactions in parallel, let them be single site or distributed. Transactions provide a higher level framework for the interaction with the state, letting the application designer trust the DBMS for operation concurrency, consistency of the state, and possible failure during transaction execution. ACID transactional guarantees [49, 51] elicit four different aspects of transactions:

**Atomicity** a transaction either succeeds (*commits*) or fails (*aborts*) leaving the database state unchanged. A transaction can either fail because of hardware or software failures, or at user will, based on some particular condition;

**Consistency** a transaction brings the database from a consistent state to another one; i.e., the integrity (e.g. user-defined integrity constraints, referential integrity) of the database is preserved;

**Isolation** every transaction is executed in isolation with respect to the others; i.e., the concurrent execution of transactions results in a system state that would be obtained if transactions were executed sequentially;

**Durability** if a transaction commits, its results will be visible across time, even in the case of failures and crashes.

Enforcing integrity constraints during transaction execution is notably one of the most delicate tasks for the DBMS to achieve. It involves isolating operations when needed by applying different concurrency control techniques [24, 56]. The goal of the DBMS, indeed, is to provide the abstraction of "group of operations" without sacrificing performance on one side, and ACID guarantees on the other.

Transactions are composed of *read* and *write* operations on single data items[2]. Consider now the DBMS of a banking system that executes deposit operations in parallel without any kind of concurrency control. Both Bob and Carl owe Alice $50$$ who has a balance of $100$$. So, Bob and Carl issue two deposits towards Alice at the same time. Each deposit operation consist of reading Alice's current balance ($r_{debtor}$), calculate the update, and write the value back to the database ($w_{debtor}$). Suppose that the order of execution of operations is $r_{Bob}\ r_{Carl}\ w_{Bob}\ w_{Carl}$: since there is no concurrency control, both Bob and Carl would read the same value for Alice's balance –$100$$– and write back

[2]DBMSs also provide support for predicate read and write operations that affect a set of data items that match some user-defined predicate.

the very same value —$150\$$. The result is that Alice will complain about missing a fee and Bob and Carl will claim firmly to have paid[3]. This and other *phenomena* happen when no isolation between transaction is enforced [22, 46]. As anticipated, transaction isolation is achieved at the cost of performance. In some cases, the application can tolerate some inconsistency and a minor degree of isolation for increased parallelism and performance in transaction execution. Proscribing phenomena means to raise the isolation level. The maximum level of isolation —that coincides with the standard definition of isolation— is *serializable* and it guarantees that the transactions results are the same obtained from a sequential execution.

### 4.1.2 Stream Processors

In order to describe SPs state management capabilities we briefly describe their computational model and their state model.

State-of-the-art SPs such as Apache Storm [78], Google DataFlow [9], and Apache Flink [29] enable high-throughput and low-latency distributed processing of data streams by adopting a dataflow model that organizes the computation into a directed graph of operators. The edges of the graph are the streams of data —unbounded sequences of data elements— that flow from operator to operator. Operators consume data from their input streams and append data to their output streams. For instance, a `map` operator transforms each input element into an output element according to the behavior specified by a user-defined function. Similarly, a `filter` operator propagates or discards input elements according to a user-defined predicate. Depending on the specific system, the graph can be explicitly defined by the developer or generated from a higher-level specification. Operators can be either stateless or stateful. Stateless operators do not retain any state, and the processing of each input element only depends on the content of that element. Stateful operators accumulate some local state, which can be accessed while processing input elements. For instance, a stateful `count` operator receives a stream of words and continuously stores and outputs the number of occurrences of each word received so far.

This model offers *task parallelism* by enabling different operators to run simultaneously on the same or on different machines. It also offers *data parallelism* by creating parallel instances of each operator, with each instance working on an independent partition of the input streams. Developers need only to specify the behavior of operators and how the input streams are partitioned among parallel instances. For example, in the case of the `count` operator above, the stream needs to be partitioned by word to ensure that all the occurrences of a given word are processed by the same operator instance, which retains the current count for that word. In the case of stateless operator, partitioning does not involve any particular local state; instead, partitioning stateful operators causes the state to be distributed across different instances of the same operator.

We hereby compare state integrity, query and update, and fault tolerance for SPs to DBMSs:

**Queries and updates** The set of updates on local states caused by an element entering the graph of computation, is a group of operations, i.e. a *transaction*. The SP allows for updates of states by defining operators logic, and, in some cases, it

[3]This anomaly is known as *lost update* [22].

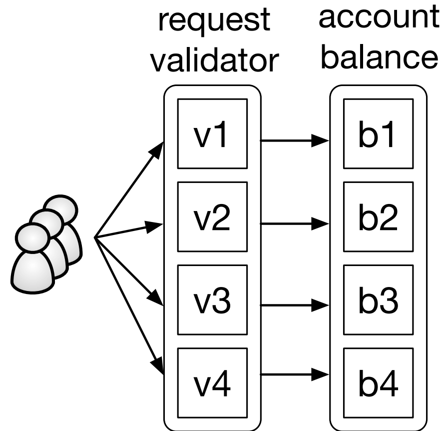



**Figure 4.1:** *A SP implementation of a bank management application.*

provides the tools to query the local states directly [27].

**Integrity** the SP does not offer state integrity. It is up to the user that defines operator's logic to guarantee state integrity;

**Fault tolerance** in case of hardware/software failure, the SP guarantees to restart processing from a consistent state; so, no state must contain the effect of a partial processing of an element[4].

The SP runtime takes care of operator deployment, data communication, and fault-tolerance, which are arguably among the most complex and critical aspects in distributed applications by enforcing some design rules that trade generality for performance and scalability. Most notably, the model requires the computation to be local to its state. This means that different operators (instances) cannot access any other operator's state. Moreover, as in the word count example, the order in which the operators process input elements must be irrelevant. Finally, queries to the local state of operators are never coordinated with state updates [27].

Indeed, transactions on a SP differ from the ones in DBMSs in three important aspects: (i) SPs do not enforce any integrity constraint on the local state of operators, and so, an operation cannot fail programmatically and cause others in the same transaction to fail; (ii) state updates can happen in any order; and (iii) external queries can access any state without checking for concurrent updates, and so, the SP does not need to handle race conditions.

In general, although these rules are key to performance and scalability, they limit the state management capabilities of SPs, as we discuss in the next section.

## 4.2 Limitations in the SP model

Our work moves from two observations: (i) companies often need to integrate *data analytics* tasks —complex computations over the input data— with *state management* tasks —transactional updates and queries to the application state; (ii) the SP model is suitable for data analytics, but presents some severe limitations in state management.

[4]Some SPs offer the possibility to trade this strong consistency guarantee with weaker, but more performing ones [27, 78].

As a consequence, companies couples SPs with state management systems, building complex architectures that hinder the design, implementation, and maintenance of the overall solution. This thesis aims to offer a unifying solution that overcomes some limitations of today's SPs to accommodate state management side by side with analytics.

To better illustrate the limitations of the SP model in state management tasks, let us consider a bank application that introduces classic problems of state management. Figure 4.1 shows part of the application: from the left, users produce a stream of bank requests, which can be either deposits, withdrawals, or transfers. Requests are first processed by a `request-validator` operator that stores the amount of money that each user has transferred from or to her account in the last month. The `request-validator` blocks further requests when the amount overcomes a given threshold. Requests are then forwarded to the `account-balance` operator that retains the current account balance for each user. Both operators consist of four instances (`v1 .. v4` and `b1 .. b4`), each responsible for a subset of the accounts. Next, we use this example to illustrate the limitations of the SP model in terms of *transactional guarantees* and *queryable state*.

### 4.2.1 Transactional guarantees

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

While all today's SPs offer fault-tolerance mechanisms, they do not always guarantee that the operations are executed in the same order upon recovery, which might lead to different states after recovering from a failure.

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

### 4.2.2 Queryable state

Even if SPs retain state information during processing, this state is hidden into operators and cannot be queried and retrieved on demand from outside the SP. To access relevant state, developers can store it in external state management systems. For instance, if `account balance` outputs the current state of each account, this information can be used to update an external DBMS. However, this leads to data duplication, with potential waste of resources and additional effort to integrate multiple systems and keep them in a consistent state.

Despite some initial proposals to make the operator state visible [27], no SP supports queries that span multiple operator instances, or considers the consistency of the returned information. For instance, in the case of bank transfers, if we could access the state of multiple accounts from `account balance`, we would like to see a transfer completed both in the provider and in the recipient accounts, or in none of them. In addition, we should not access any dirty state caused by the computation of failing transfers. Finally, once we observe the effects of a transfer, those effects should reflect in any subsequent state access.

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

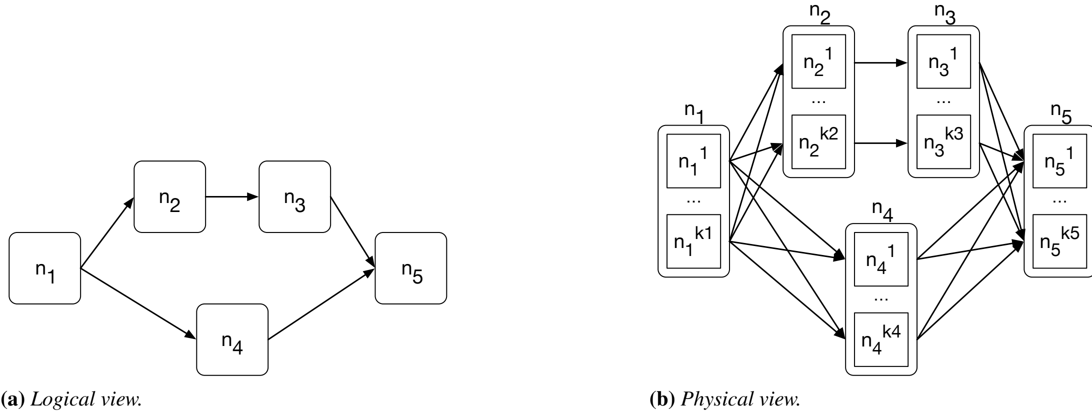
### 4.2.3 Executive summary

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

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

## 4.3 Transactions on a Stream Processor

We extend the dataflow model of distributed SPs by introducing the concept of *transactional subgraph* (*t-graph*), which identifies a portion of the graph of computation where




**(a)** *Logical view.* **(b)** *Physical view.*

**Figure 4.2:** *The topology of a graph of computation: logical and physical view.*

the state of enclosed operators is accessed and updated with transactional semantics. Each streaming element that enters the t-graph initiates a *read-write transaction*: all its effects on the state of operators within the t-graph are processed as a single transaction with ACID guarantees. The state of operators within the t-graph is also externally queryable through *read-only transactions*. By limiting the scope of transactions to t-graphs, the model provides data consistency when needed and high performance when possible. Furthermore, developers can configure the level of isolation and the durability for t-graphs, selecting the best trade-off for the application at hand.

### 4.3.1 Stream processing model

Building on the dataflow model of distributed SPs, we represent a computation as a directed graph $G = (N, E)$, where the nodes in $N$ are the processing operators and the edges in $E$ are the streams of data between operators. Streams are typed, meaning that all the elements in a stream share the same structure. Figure 4.2a shows a graph of computation that includes five operators $n_1 \dots n_5$. An operator can receive input elements from one or more streams and append output elements to one or more streams. Streams originate from sources (such as operator $n_1$ in Figure 4.2a) that receive data from the external environment, and terminate in sinks (such as operator $n_5$ in Figure 4.2a) that return results to the external environment. We abstract the behavior of an operator $n_i \in N$ with a characteristic function $f_n$ that determines how the operator processes input elements, updates its internal state (if any), and produces output elements.

Operators can be replicated in multiple *instances* for scalability, with each instance considering a portion of the input streams. We denote the $k$ instances of an operator $n_i \in N$ as $n_i^1, \dots, n_i^k$. An instance $n_i^j$ processes one input element at a time on a single processing thread, and appends zero, one, or more elements to each of its output streams, according to the characteristic function $f_{n_i}$ of the operator. Figure 4.2b shows the physical view of a graph of computation, with multiple instances of each operator. We assume that the communication channels between instances (the arrows in Figure 4.2b) are FIFO ordered, meaning that the elements are received and processed by the downstream operator in the same order in which they are produced by the upstream operator. To the best of our knowledge, this assumption holds in all distributed SPs,

which usually adopt TCP communication channels.

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

As in the original SP model, the state of operators is local to each instance such that two instances of the same or different operators do not share any state. Developers control the partitioning strategy through a `keyBy` function, which computes a *key* for a given element. Elements with identical keys are guaranteed to be processed by the same operator instance, which retains any state for that key. The partitioning strategy is relevant in the case of stateful operators. For example, in the bank management application in Figure 4.1, account balances are partitioned by account number. All the requests involving a given account need to be processed by the same operator instance, the one that stores that account. Developers can enforce this by indicating the account number as the key of the elements that enter `account balance`.

In general, each instance of an operator $n_i$ can produce elements for any instance of a downstream operator $n_j$. However, if two operators have the same partitioning strategy, then the $k$-th instance of operator $n_i$ ($n_i^k$) is guaranteed to produce elements only for the $k$-th instance of a downstream operator $n_j$ ($n_j^k$). This is exemplified by the communication between $n_2$ and $n_3$ in Figure 4.2b.

### 4.3.2 State management model

Our model introduces state management capabilities within *transactional subgraphs* (*t-graphs*). A t-graph $T = (N_T, E_T)$ is a connected subgraph of $G$ that is constrained to have a single input edge $in_T$. Developers can introduce multiple t-graphs, provided that they do not share any operator. We denote $S_T$ (the *state of* $T$) the set of all the stateful operators that are part of the t-graph $T$. Each operator $s \in S_T$ has a name $n_s$ to make it visible and queryable by name from outside the SP. As in the traditional SP model, an operator $s \in S_T$ processes elements partitioned by key: each operator instance stores the state for the partition it is responsible for in the form of key-value pairs $(k, v)$, $k \in K_s$, $v \in V_s$, where $K_s$ is the key domain and $V_s$ is the value domain for operator $s$. Keys are unique, meaning that an operator $s$ can store only one value for each key. An operator $s \in S_T$ can be associated with an *integrity constraint* that determines the set of valid values for a given key. In the bank application in Section 4.2, a developer can introduce an integrity constraint that requires the amount of money for each account to be non-negative.

Each streaming element $e$ entering a t-graph $T$ determines a *read-write transaction*: all the state changes that $e$ induces on $S_T$ take place with transactional semantics. External queries are *read (only) transactions* that retrieve part of the state in $S_T$ with transactional semantics. More precisely, we model the interaction with $S_T$ with two *operations*: *read* and *write*, which access and update the value for a key, respectively. Insert and delete operations are considered as special cases of write. We model a transaction as an ordered set of operations. Queries include only read operations. Read-write transactions include also write operations: they are initiated by an element $e$ entering $T$ and comprise all the operations performed by $e$ or by any element $e'$ caused by $e$ ($e \xrightarrow{*} e'$) on any operator $s \in S_T$ during the execution of its characteristic function $f_s$.

We associate each transaction with a unique *identifier* and we denote $t_i$ the transac-

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

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

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

### 4.3.3 Transactional guarantees

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

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

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

**Isolation** Isolation limits the interaction between concurrently executed transactions that read and write common keys. Our model allows developers to select different levels of isolation. More relaxed levels introduce fewer constraints and thus enable a higher degree of concurrency and higher performance. Conversely, stricter levels constrain the interaction between transactions more and thus offer higher guarantees but a lower degree of concurrency and performance. We inherit and extend standard isolation levels from the database literature [3] and we present them from the least to the most constraining. Each level subsumes the previous one.

Isolation level `PL1` avoids write dependencies between concurrent transactions: the effects of transactions are the same as if their write operations were performed in some

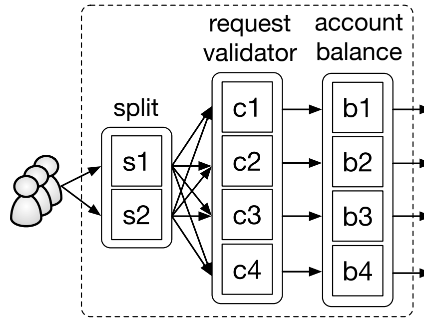



**Figure 4.3:** *Implementation of the bank management application in our model.*

sequential order. Specifically, if transaction $t_1$ installs version $v_1$ for key $k$, and transaction $t_2$ over-writes $k$ by installing version $v_2$, there should not be another key $k'$ in which the revers occurs, that is, all writes of $t_1$ must be ordered before or after all writes of $t_2$.

Isolation level PL2 additionally requires transactions to only *read* installed versions[5]. Specifically, under PL2, a valid history cannot contain a write operation $w_1(k^s_v)$ where transaction $t_1$ writes (creates) version $v$ for key $k$ in state $s$ followed by a read operation $r_2(k^s_v)$ where transaction $t_2$ reads version $v$, unless $t_1$ commits and installs $v$.

Isolation level PL3 additionally prevents transactions from overwriting versions read by other transactions that have not yet completed. Specifically, under PL3, a valid history cannot contain a read operation $r_1(k^s_{v_1})$ where transaction $t_1$ reads version $v_1$ for key $k$ in state $s$ followed by a write operation $w_2(k^s_{v_2})$ where transaction $t_2$ writes version $v_2$ before transaction $t_1$ is committed or aborted. Isolation level PL3 is also known as (conflict) serializable isolation [24] and ensures that the state of a t-graph is the same as if all the transactions were executed in some sequential order, one after the other.

We further provide a stricter level of isolation that we denote PL4. It extends level PL3 by ensuring that the state of a t-graph is the same as if all the read-write transactions were executed sequentially in the *same* order in which they enter the t-graph. This level corresponds to strict serializability in classic database literature [73].

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

### 4.3.4 The model in action

To further clarify our model, we show how it can be used to implement the bank management application in Figure 4.1. Recall that `request validator` blocks requests based

[5]Reading non-installed versions results in the *dirty read* anomaly [24]. For this reason, PL2 is also referred to as *read-committed* in the ANSI standard [46].

on the history of requests for a given account, and `account balance` stores the current value of each account. As discussed in Section 4.2, SPs presented two main limitations in this context: the impossibility to access the internal state of operators and the impossibility to process bank transfers correctly without sacrificing the distribution of data and processing. Figure 4.3 shows a possible implementation of the bank application in our model. To guarantee transactional semantics, we include all the operators in a t-graph (dashed box in Figure 4.3). We introduce a `split` operator (consisting of two instances in Figure 4.3) that processes user requests and redirects them to the instances of the downstream `request validator` and `account balance` partitioned by account — that is, the `keyBy` function specifies the account number as the key for each request. A bank transfer request is split in a withdrawal from the source account and a deposit to the receiver account. Finally, `account balance` has an associated integrity constraint that requires the amount of each bank account to be non-negative.

Our model avoids the problems discussed in Section 4.2 and correctly processes bank transfers even if the information on bank accounts is partitioned across multiple instances of `request validator` and `account balance`. Indeed, the withdrawal and the deposit that compose a bank transfer originate from a single input request that enters the t-graph and are processed as part of a single read-write transaction with ACID guarantees. Atomicity and consistency ensure that if the withdrawal violates the integrity constraints on the `account balance`, then none of the effects of the request is registered in the state of the operators. Instead, the effects of successful requests reflect on the state of both `request validator` and `account balance`. Isolation guarantees that two requests do not overlap. By selecting level `PL3` or higher, developers ensure that requests behave as if they were executed sequentially. Durability ensures that the effect of successful requests are persisted even in the case of failures. Finally, the state of `account balance` is exposed for queries, which are guaranteed to return all the effects of a bank transfer or none of them.

### 4.3.5 Limitations

The design of the presented model exploits the state-of-the-art graph of computation model for stream processing, that requires shared-nothing operators and the absence of a global state. This decision limits the expressiveness of transactional computation for the sake of locality of computation and better performance during execution.

The absence of a global state means that it is impossible to read and/or update rows in different states from a single operator. The only global condition on the state enforced by the model is invariants.

For example, take S-Store's stream-voting example [33], where a set of voters vote for candidates in a TV game-show. The voters can cast a single vote each. The candidate with the fewest votes is removed every $20,000$ votes received, because she/he is the least popular. When these candidates are removed, votes submitted for him or her are returned to the people who cast them. Those returned votes may then be re-submitted for any of the remaining candidates.

Using TSpoon to implement it, we can use an operator that keeps the count of votes and forbids voters to vote twice; an operator to store the count of votes for every candidate; and an operator that stores the candidate that is at the bottom of the ranking, in order to remove her/him every $20,000$ votes. However, TSpoon cannot implement

this streaming application, because the last operator cannot return the votes for the eliminated candidate: this would mean that it should be possible to update the state of the operator containing the number of votes per voter from the last operator of the pipeline. TSpoon's model allows to influence the computation by propagating results downstream, instead.

If needed, the model and the implementation of TSpoon could be adapted to account for these use cases in the future. However, note that this choice limits the developer in favor of a model closer to the state-of-the-art one for SPs, and in favor of efficiency.

## 4.4 Implementation

We implemented our model in the TSpoon (Transactions ON the Stream ProcessOr) system[6], which builds on the Apache Flink [29] open-source distributed SP.

### 4.4.1 TSpoon API

We illustrate the TSpoon API with an implementation of the bank application from Section 4.3.4, where we omit the `request validator` for simplicity. The application receives a stream of bank transfer requests, splits each of them into a deposit and a withdrawal, and executes them within a single transaction. Listing 4.1 shows the code of the application.

**Listing 4.1:** *Bank transfer example in TSpoon*

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

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

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

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

TSpoon augments the Flink API with two `openTransaction` and `closeTransaction` operators to define the boundaries of a t-graph. In Listing 4.1, TSpoon takes in input a stream of bank transfers `transferStream`. The `openTransaction` opens a t-graph and transforms the input `DataStream` into a `TransactionalDataStream t`. A `flatMap` operator splits each transfer into the corresponding deposit and withdrawal, creating a stream of `BankOperation`. TSpoon offers an overload of several Flink operators that makes the internal state and its changes explicit. In Listing 4.1, the `account balance` operator is implemented as a stateful `map`: the first three arguments are the name of the operator and the types of the key and value. The fourth argument indicates how the value for a given key is updated when a new bank operation `op` is received. The fifth

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

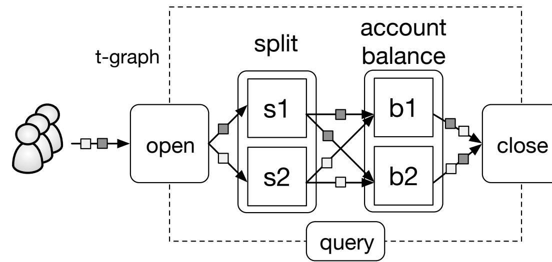



**Figure 4.4:** *Bank management application: architecture of the t-graph.*

argument is the integrity constraint on the value. The last argument is the output of the operator. Finally, the `closeTransaction()` closes the t-graph.

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

### 4.4.2 TSpoon architecture and transactional guarantees

Figure 4.4 shows how TSpoon instantiates the t-graph defined in Listing 4.1. The t-graph contains the `account balance` stateful operator with the current balance of bank accounts partitioned across two instances; `split` is the `flatMap` function that receives transfer requests and redirects them to the instances of `account balance` that are responsible for the bank accounts in each request. As an example, Figure 4.4 shows the stream elements involved in the processing of two transfer requests, represented as square boxes of different colors (light grey for one request and dark grey for the other one). Each request is managed by a different instance of the `split` operator that transforms the transfer into a deposit and a withdrawal, each handled by an instance of `account balance`. The `close` operator propagates downstream all the results of `account balance` enriched with the outcome and set of changes of the transaction they belong to.

For each t-graph, TSpoon automatically and transparently instantiates the `open`, `query`, and `close` operators, also shown in Figure 4.4, which implement the algorithms to process queries and to enforce the required transactional guarantees. In a nutshell, the `open` operator wraps all incoming elements into data structures that carry metadata about transactions. Since individual transactions might be invalidated and re-executed multiple times to satisfy isolation constraints, the `open` operator also stores pending transactions. The `query` operator acts as a proxy for read-only queries and ensures that they achieve the desired isolation level. Stateful operators process the input elements and try to apply the requested changes to their local state. They propagate downstream the outcome of the changes, which might also be negative (abort) in the case of the violation of some integrity constraint. The `close` operator collects all outgoing elements to determine the global outcome of a transaction, and communicates it back to the stateful operators involved. When all these stateful operators acknowledge the communication, the `close` operator propagates the result of the transaction downstream. At this point, we say that the transaction is *complete*. Although not shown in Figure 4.4, TSpoon can create multiple instances of the `open` and `close` operators to process different transactions in parallel.

#### Data structures

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

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

The `open` operator assigns the fields *id*, $ts_{exec}$, and $ts_{compl}$. Each operator in the t-graph that processes an element $e_1$ producing element $e_2$ copies these fields from $e_1$ to $e_2$. The `open` operator ensures that timestamps are unique. The *fragment* tracks the number of elements that each operator in the t-graph produces. For instance, consider again the bank management application in Figure 4.4: when the `split` operator processes a bank transfer request, it generates a deposit and a withdrawal, and uses the *fragment* field to notify downstream that it produced two elements. The `close` operator inspects the *fragment* field to determine when it received all the elements for a transaction. Stateful operators process incoming elements and propagate downstream the state changes they perform — *update* field — and the local *outcome* of the processing, which might indicate the occurrence of errors such as the violation of an integrity constraint.

#### Atomicity and consistency

TSpoon achieves atomicity and consistency by implementing a two phase commit (2PC) protocol [24], where stateful operators are participants and the `close` operator is the coordinator. Consider a transaction $t_i$ with *id* $i$ and $ts_{exec}$ $ts$. Consider a stateful operator $o$ that processes an element $e$ that is part of $t_i$. While processing $e$, operator $o$ can access its local state and create new versions for its local keys. Operator $o$ decorates the output elements with the *outcome* metadata, which is propagated downstream to all the elements caused by $e$. The outcome is *commit* if the processing terminates successfully, *abort* if the processing violates some integrity constraints, or *retry* if the processing violates some isolation policy. A *retry* is semantically equivalent to an *abort*, but additionally causes TSpoon to attempt re-executing the transaction.

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

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

#### Isolation

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

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

This strategy serializes operations on individual keys, but still allows write operations from concurrent transactions to be installed in different orders in different instances of one or more operators, which violates the requirement of PL1. Consider for example the two bank transfer requests in Figure 4.4 and assume that they involve the same two bank accounts $a_1$ and $a_2$. Since the requests are handled concurrently in the split operator, it is possible that account $a_1$ processes the light grey request first, while account $a_2$ processes the dark grey request first. TSpoon prevents this violation by forcing transactions to execute in order with respect to their execution timestamp $ts_{exec}$. In particular, it aborts transactions that attempt to execute operations out of timestamp order and schedules them for re-execution (with a higher timestamp). This strategy avoids deadlocks: if transaction $t_2$ is waiting for some operation of transaction $t_1$ to complete, then the execution timestamp of $t_2$ is larger than that of $t_1$. As a consequence, if some operator receives an element from $t_1$ after an element from $t_2$, it aborts $t_1$ preventing it from locking any resource. Finally, to further reduce the probability of

re-executing transactions, the lock-based protocol reorders queued elements according to their execution timestamps while they wait to acquire some resource.

The above protocol ensures that a transaction can access a key only when the previous updates to that key have been installed, thus preventing the read-uncommitted anomaly. Moreover, all the (read and write) operations that involve the same keys are executed in timestamp order. This ensures that the results of transactions are the same as if they were executed sequentially in timestamp order. Thus, this algorithm guarantees isolation level PL3. The protocol, however, does not guarantee isolation level PL4, since in the case of re-execution the order of execution timestamps may not reflect the order of transaction ids. If the developer requires level PL4, TSpoon introduces an additional sequencer component before *each* stateful operator in a t-graph, which consists of a single instance and reorders transactions according to their *id*. The sequencer adopts the same mechanism (based on *fragment*) of the close operator to determine the number of elements that compose a transaction.

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

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

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

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

#### Queries

TSpoon provides access to the state of a t-graph using the query operator as a proxy. It ensures that queries access a consistent snapshot of the t-graph by obtaining the

timestamp of complete transactions from the `open` operator. When trying to access a key, a query (that is, a read-only transaction) is assigned the timestamp $t_c$ of the last completed (committed or aborted) transaction, and it always accesses the latest version with timestamp lower or equal to $t_c$.

#### Durability

TSpoon provides durability by relying on and extending the fault-tolerance algorithm of Flink, based on distributed snapshotting [28]: special markers periodically flow through the network of operators from sources to sinks; upon receiving a marker, each stateful operator stores its state to some durable storage and propagates the marker downstream. Upon failure, operators restore their state from the last snapshot, and sources replay all the elements that were not part of the snapshot.

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

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

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

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

| | Latency | Sust. throughput |
|---|---|---|
| TSpoon | 3.57 ms | 8580 el/s |
| Flink | 0.6 ms | 59609 el/s |
| VoltDB | 4.03 ms | 243 tr/s |

**Table 4.1:** *Default scenario: comparison with Flink and VoltDB.*

## 4.5 Evaluation

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

### 4.5.1 Experiment setup

We deploy TSpoon on a cluster of 5 Amazon EC2 t2 xlarge instances (with 4 CPU cores and 16 GB of RAM) and 15 t2 large instances (with 2 CPU cores and 8 GB of RAM), for a total of 50 CPU cores and 184 GB of RAM. As a default scenario, we consider the bank application presented in Section 4.4: TSpoon receives a stream of input bank transfers, and splits each transfer in a deposit and a withdrawal that are processed within a single transaction. We store 100k bank accounts partitioned across 50 instances of the `account balance` stateful operator, one for each CPU core. The source and destination accounts for each bank transfer are selected randomly following a uniform distribution. We assess the performance of TSpoon by measuring its throughput and latency. We measure the average latency when the system is unloaded, by submitting input requests sequentially, sending an element at a time. In terms of throughput, we want to determine the maximum value of input elements that TSpoon can sustain before becoming overloaded and losing responsiveness (we call this "the *sustainable throughput*"). In practice, for each experiment we increase the input rate stepwise until the latency overcomes a given threshold (20 times the latency of the unloaded system). We repeat all experiments 8 times. For each measure, we plot the average value and the standard deviation.

### 4.5.2 Default scenario

We use the default scenario described above to compare TSpoon against Flink 1.3.2 and the VoltDB in-memory distributed DBMS version 8.0[7], which are state-of-the-art representatives of their categories, well known for their excellent level of performance.

[7] `https://github.com/voltdb/voltdb`

We configure both Flink and TSpoon to deploy 50 instances of each operator — one per CPU core, a typical Flink configuration. We set the level of isolation for TSpoon to PL3 (the same adopted by VoltDB) and we use timestamp-based concurrency control. VoltDB only enables developers to configure the number of database partitions per machine. We configure 2 partitions per machine, for a total of 40 partitions, since 15 out of 20 nodes in the cluster have 2 CPU cores, and we do not want to over-commit the available resources. We implement the bank transfer transaction as a stored procedure that is analyzed and compiled at deployment time to eliminate the overhead of creating a query plan at runtime. We compute the throughput and latency of VoltDB using the provided benchmarking tools[8]. They measure the maximum throughput by submitting 200k bank transfer transactions in a single burst, and then computing the average latency with an input rate that is below the maximum throughput. The comparison is fair, since the maximum throughput is an *over*-estimation of the sustainable throughput.

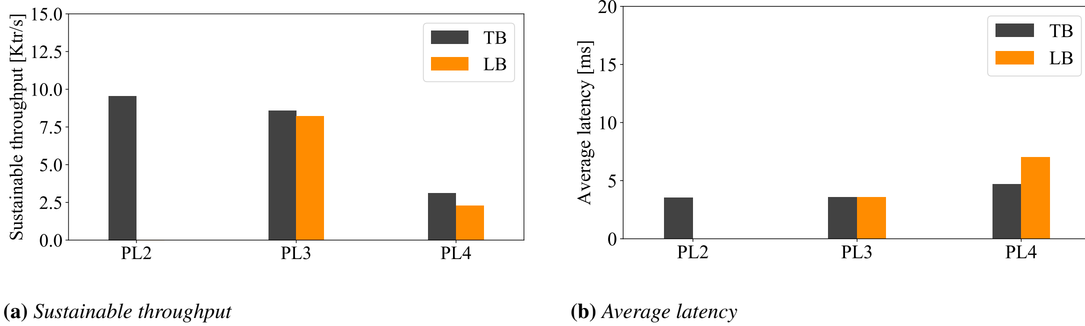
Table 4.1 shows the results we measured. TSpoon achieves a sustainable throughput of more than 8500 input elements/s with 3.57 ms latency. By comparison Flink processes 59609 input elements/s with an average latency of 0.6 ms. However, it is worth mentioning once for all that the application implemented in Flink differs from that implemented by TSpoon and VoltDB as Flink does not provide any transactional guarantee and the results it produce can violate the requirements of our bank transfer scenario. In practice, these tests measure the overhead of TSpoon in enforcing transactional guarantees. Flink can process deposits and withdrawals in parallel, in any order, while TSpoon introduces concurrency constraints to enforce isolation and atomicity. As a more fair comparison, VoltDB offers the same transactional guarantees of TSpoon, but achieves a throughput of only 243 transactions/s with a latency of around 4 ms. Here we may observe that VoltDB is optimized for transactions that involve a single partition and this result indicates that multi-partition transactions are very expensive. Indeed, if we change our workload, forcing bank transfer requests to move money from accounts being part of the same partition, VoltDB performance grows considerably, providing a throughput of more than 400k transactions/s with an average latency of 0.03 ms. On the other hand, VoltDB partitions cannot span multiple CPU cores or machines, which force using multi-partition transactions in all those applications that do not naturally map transactions on a statically identifiable subset of data.

In summary, the results shown in Table 4.1 demonstrate that TSpoon is competitive with state-of-the-art data processing and management systems: it provides the same performance as Flink when used for pure stream processing tasks, and reduces the throughput by less than $7\times$ when providing strong (PL3) transactional guarantees. It also significantly outperforms VoltDB in terms of multi-partition transactional updates.

### 4.5.3 Isolation levels and concurrency control strategies

Figure 4.5 shows the performance of TSpoon in our default scenario with different isolation levels (PL2, PL3, PL4) and concurrency control strategies (lock-based and timestamp-based). Moving from PL2 to PL3 does not introduce a significant drop in throughput (Figure 4.5a) or latency (Figure 4.5b). Indeed, our default scenario includes a large number of bank accounts that lead to minimal state access conflicts. At level PL3, lock-based and timestamp-based protocols exhibit comparable behaviors, with

[8] https://github.com/VoltDB/voltdb/blob/master/examples/voter/README.md




**(a)** *Sustainable throughput*

**(b)** *Average latency*

**Figure 4.5:** *Default scenario: comparison of isolation levels and concurrency control strategies.*

a small advantage of timestamp-based in terms of throughput. PL4 is clearly more expensive, leading to a throughput of about 2280 elements/s for lock-based and about 3100 elements/s for timestamp-based. Indeed, the strong requirement of processing transactions in *id* order demands for a single-instance operator that enforces this order. lock-based imposes the order upfront, while timestamp-based checks the order before transactions complete, aborting and rescheduling those transactions that violate it. In our default scenario, the first strategy is more expensive and leads to an increase in latency (up to 7.5 ms).

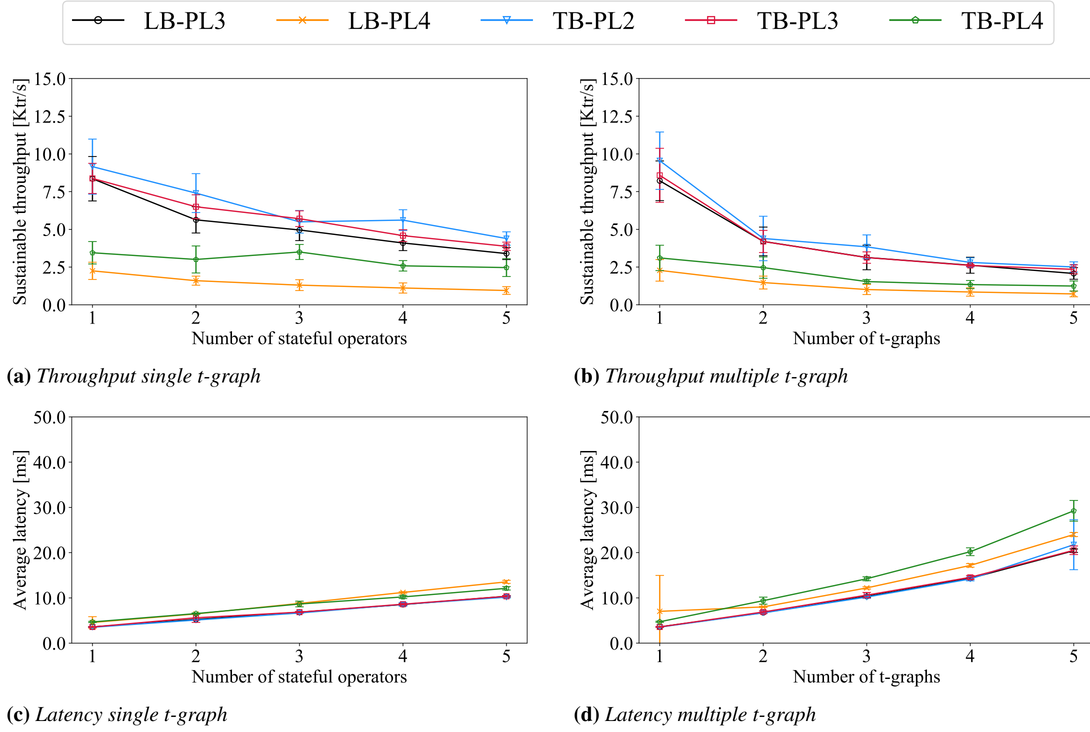
### 4.5.4 Sensitivity to parameters

We now investigate how workload parameters influence TSpoon.

#### Chain of dependent updates

We first consider a chain of updates performed one after the other, mimicking a scenario where the data produced by a state update is elaborated downstream and produces updates in other operators. We consider both the case in which all the involved stateful operators belong to the same t-graph and the case in which each stateful operator belongs to a different t-graphs. Each stateful operator includes 100k different keys, as in our default scenario. Figure 4.6 shows that the throughput decreases with the number of stateful operators, both in the case of a single t-graph and in the case of multiple t-graphs. In the case of a single t-graph (Figure 4.6a), the elements of a transaction traverse the entire pipeline of operators before the transaction complete. The longer the pipeline, the higher the probability of conflicts between transactions. This problem does not occur in the case of different t-graphs (Figure 4.6b), which instead introduce the overhead of opening and closing multiple transactions. The throughput is higher in the case of a single t-graph, meaning that the opening and closing multiple t-graphs is more expensive than processing a single, longer transaction.

In the case of a single t-graph, the latency only slightly increases when moving from one to five stateful operators due to the longer path from sources to sinks, and remains below 14 ms for all the configurations we tested (Figure 4.6c). In the case of multiple t-graphs, the latency increases more (up to almost 30 ms in the case of PL4 with timestamp-based protocol), due to the presence of an additional open and close operators for each t-graph.




**(a)** *Throughput single t-graph*

**(b)** *Throughput multiple t-graph*

**(c)** *Latency single t-graph*

**(d)** *Latency multiple t-graph*

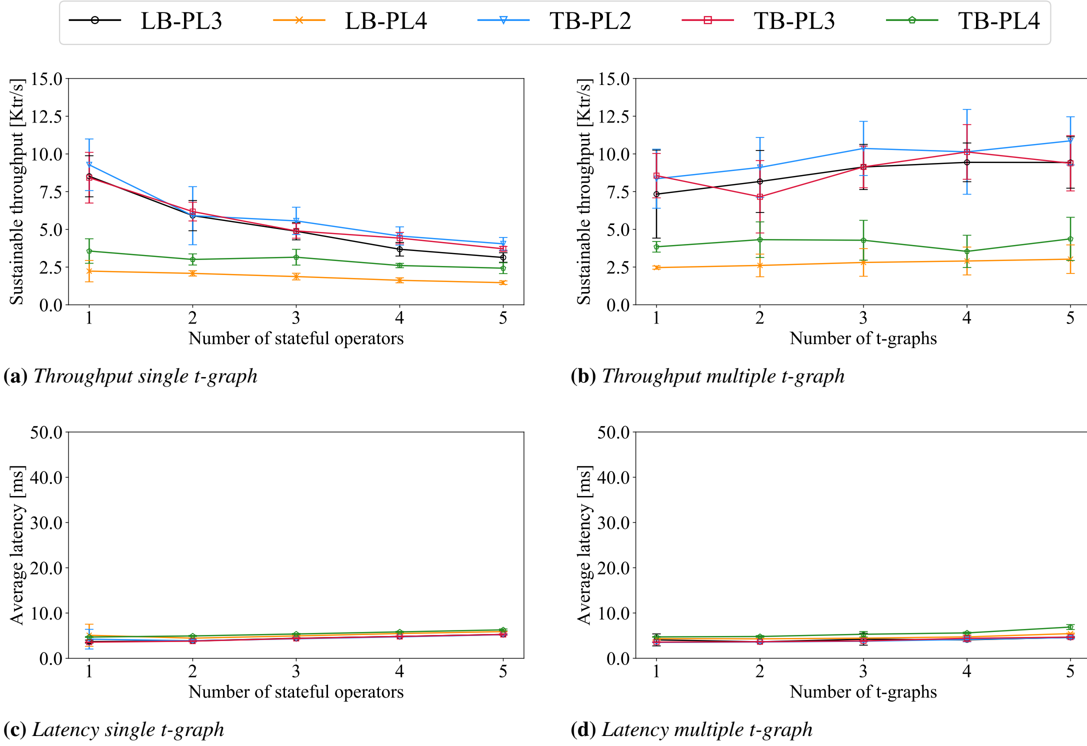
**Figure 4.6:** *Chained updates.*

#### Independent updates

Figure 4.7 shows the performance of TSpoon when considering independent state updates that occur in parallel, in a single t-graph or in distinct t-graphs. In the case of a single t-graph (Figure 4.7a), the throughput decreases with the number of stateful operators. Indeed, a higher number of stateful operators increases the probability of access conflicts and also forces the `close` operator to wait for more outcomes. The throughput decreases from about 8000 elements/s to less than 3700 elements/s with isolation levels up to `PL3`. `PL4` exhibits the lower throughput with about 2500 elements/s for timestamp-based and about 1500 elements/s for lock-based. The overhead of `PL4` protocols dominates the costs associated to the increased number of stateful operators, leading to the same throughput from one to five stateful operators. In the case of multiple t-graphs (Figure 4.7b), the throughput remains almost constant, since TSpoon can process transactions entirely in parallel. The latency (Figure 4.7c and Figure 4.7d) also remains almost constant in all the scenarios we tested.

#### Number of keys

We now study how the probability of state access conflicts between transactions influences the performance of TSpoon. We consider again our default scenario and we change the number of keys (bank accounts) within the `account balance` operator. As Figure 4.8a shows, lock-based tolerates access conflicts better, and its throughput does not significantly decrease even in the extreme case of only 100 keys. Instead, timestamp-based is more affected. The throughput decreases when reducing the number of keys for all isolation levels. In the case of `PL2`, the throughput remains stable

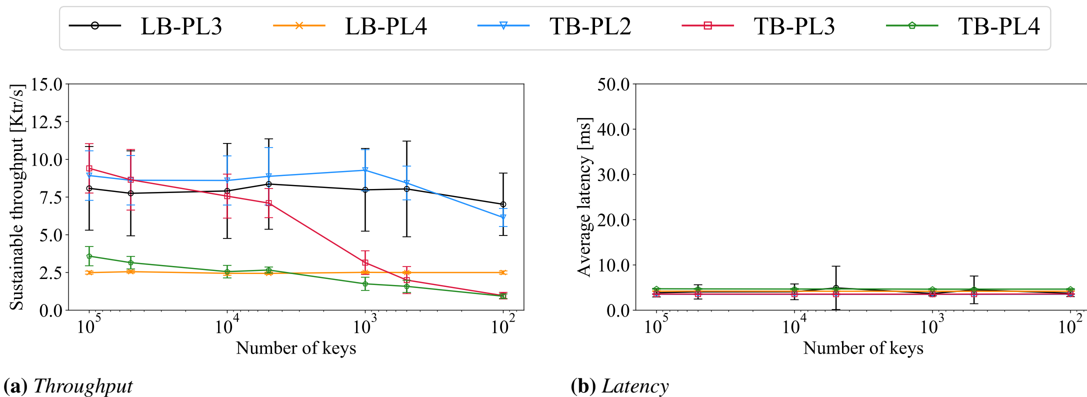



**(a)** *Throughput single t-graph* **(b)** *Throughput multiple t-graph*

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

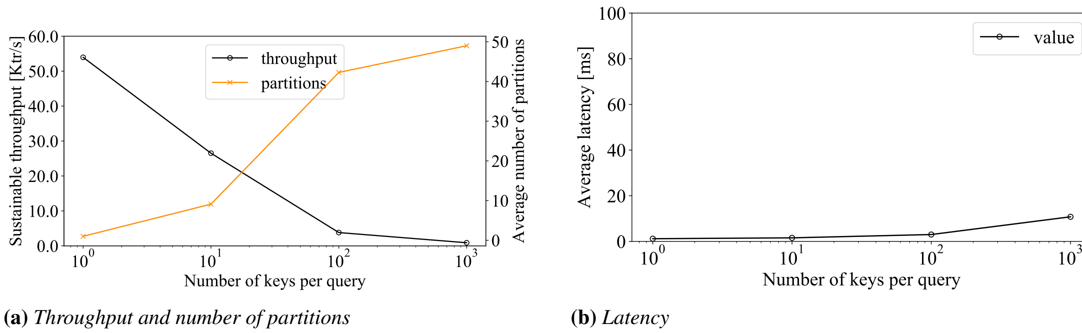
**Figure 4.7:** *Independent updates.*

from 10k to 1000 keys, and decreases with fewer keys from 8900 to less than 6200 elements/s. State access conflicts influence PL3 and PL4 the most, since these two levels of isolation introduce more constraints and increase the number of transactions that have to be re-executed. The throughput decreases in both cases, reaching less than 1000 elements/s in both cases (less than PL4 with lock-based protocol). The latency, reported in Figure 4.8b, remains almost stable when changing the number of keys. Indeed, we measure latency when the system is not overloaded and there are no state access conflicts.




**(a)** *Throughput* **(b)** *Latency*

**Figure 4.8:** *Number of keys.*




**(a)** *Throughput and number of partitions*

**(b)** *Latency*

**Figure 4.9:** *Read queries at different selectivity.*

### Queries

We now study the throughput for external queries. Since the isolation level does not affect queries, we fix it to PL3. We use our default scenario with a fixed rate of updates of 1000 bank transfer requests/s and we change the selectivity of queries, that is, the number of accounts that each query selects. The yellow line in Figure 4.9a shows the average number of `account balance` instances each query accesses. The black line shows the sustainable throughput for queries: TSpoon supports 53900 queries/s when accessing a single instance of `account balance`. As the number of involved instances increases, the throughput decreases, reaching 890 queries/s in the extreme case in which a query accesses 1000 keys. Query latency, reported in Figure 4.9b, ranges from less than 1 ms when querying a single partition to 10 ms when querying all partitions.

### Cost of durability

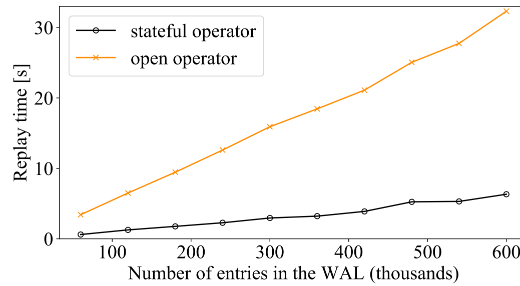



**Figure 4.10:** *Number of pending transactions.*

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

Next, we measure the time to recover from a failure. Recall that we rely on the Flink snapshot algorithm for fault-tolerance, which we augment with the WAL to ensure transactional semantics. The time to recover includes (1) the time to discover a failure; (2) the time to restore Flink to the last snapshot; (3) the time to restore the state of t-graphs from the WAL. The first two contributions only depend on Flink and its configuration. Thus, we measure the last contribution, which is specific to our model. Restoring the state of t-graphs involves the costs to restore: (i) the local state of each stateful operator; (ii) the state of the `open` operator. These two operations are performed in parallel, and so the overall cost of recovery is the maximum of the two contributions. Figure 4.10 shows the cost of these two contributions in our default scenario, when changing the number of elements stored in the WAL (which depends on the input rate of transactions and on the frequency between two snapshots). In our scenario, the cost to restore the `open` operator dominates the cost to restore each stateful operator, resulting in a recovery time of 3.5 s with 60k transactions in the WAL and 32 s with 600k transactions in the WAL.

### 4.5.5 Scalability

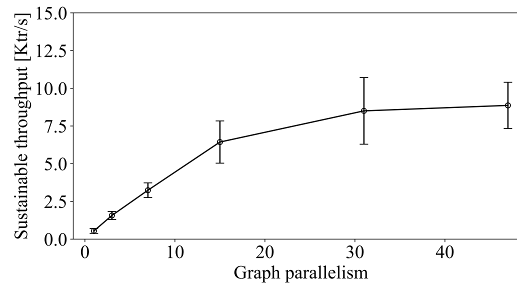



**Figure 4.11:** *Number of partitions.*

Finally, Figure 4.11 shows how TSpoon scales when increasing the number of cores (and, correspondingly, the number of partitions for the stateful operators). In our default scenario, the throughput increases from 544 to 6436 elements/s (11.8×) when moving from 1 core to 16 cores. After this threshold, adding new cores brings fewer benefits, and the throughout only increases to 8866 with 48 cores.

CHAPTER 5

# Conclusions and Future Work

In the first part of this thesis we studied the execution semantics of modern distributed SPs focusing on the key notions of time and windows. Our analysis grounds on the SECRET model that was developed in 2010 to capture the semantics of the SPs available at that time. On the one hand, SECRET can capture the window behavior of most modern SPs, which indicates that the same abstractions that were introduced in early systems are still adopted. The analysis highlights a general agreement on the semantics of time windows, supported by all the systems we analyzed, and count windows, present only in few systems. On the other hand, modern SPs are far more complex that the systems SECRET was designed for, and a precise understanding of their behavior demands for additional modeling efforts.

We identified some key points that SECRET lacks in expressing, such as the effect of distribution, out-of-order arrival of elements, behavior in case of failure, and dynamic windows. By shading light on these aspects, we posed the foundations for a precise understanding of modern distributed SPs and we drew a road map for future research efforts.

In the second part of this thesis, we tackled DBMS/SP integration. Data-intensive applications, indeed, increasingly often combine consistent state management with analytics on large volumes of dynamic (streaming) data. Current architectures satisfy these needs by exploiting multiple subsystems, but this leaves developers with the daunting task of coherently integrating these subsystems. This approach increases the maintenance cost of the entire system, forces developing team to adopt a variety of programming models, and degrades the responsiveness of the system. To overcome these limitations, we proposed a novel model that seamlessly integrates transactional state management within a distributed stream processor. The model introduces transactional regions within dataflow computation graphs: each element entering a transactional re-

gion initiates a read-write transaction, and the internal state of the region can be queried with read-only transactions. We implemented the model in the TSpoon system, which offers different levels of isolation and durability to let developers choose the best trade-off between performance and consistency for the application at hand. We evaluated TSpoon thoroughly measuring its performance under different workloads, transactional semantics, and implementation strategies, showing that it can outperform state-of-the-art state management tools in common scenarios. The key advantage of our model with respect to DBMSs is its capability of expressing transactional workloads with a dataflow graph of computation which enables for inherent task and data parallelisms. TSpoon, indeed, proves its high performance when dealing with workloads of transaction composed of operations that can be executed in parallel. This trait is key to the performance in executing transactional workloads. Indeed, it has already been investigated in the past in the form of multi-level transactions and, more recently, in actor-oriented databases. We argue that both approaches obtain the expected performance increase, but they sacrifice usability of the query language by forcing the user to use a mixture of declarative SQL and asynchronous computation. This would require personnel able to use both programming paradigms. We identify the dataflow as key to expressing concurrent computation in an easy and understandable way and, as such, as an enabler for expressing parallel transactional computations.

However, there is still work to do. For instance, our model does not treat *single-partition updates* as special cases: these type of transactions do not require coordination upon commit/abort, because the outcome of the transaction can be determined locally to the partition, thus achieving higher throughput and lower latency.

Moreover, our model does not allow the flexibility that DBMSs provide; for example it does not provide the abstraction of global state, but only enables local computation thus making it impossible to implement transactions that repeatedly update the same piece of state. In the future, we plan to extend our model to manage this use case.

Finally, the query language for queryable state is still programmatic and not SQL based.

However, we are confident that this work has the potential to open a new line of research and innovation, and lead to architectures that are more efficient and easier to design, develop, and maintain.

# Bibliography


[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. The design of the borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research*, CIDR '05, pages 277–289, 2005.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

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

[4] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. Flowdb: Integrating stream processing and consistent state management. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 134–145. ACM, 2017.

[5] Lorenzo Affetti, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola, and Emanuele Della Valle. Defining the execution semantics of stream processing engines. *Journal of Big Data*, 4(1):12, 2017.

[6] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the International Conference on Management of Data*, SIGMOD'08, pages 147–160, New York, NY, USA, 2008. ACM.

[7] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. In *Proceedings of the International Conference on Management of Data*, SIGMOD '05, pages 882–884, New York, NY, USA, 2005. ACM.

[8] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of VLDB*, 6(11):1033–1044, 2013.

[9] 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. 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):1792–1803, 2015.

[10] 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. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[11] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: A unified language for event processing and stream reasoning. In *Proceedings of the International Conference on World Wide Web*, WWW'11, pages 635–644, New York, NY, USA, 2011. ACM.

[12] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[13] A. Artikis, M. Sergot, and G. Paliouras. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):895–908, 2015.

[14] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

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

[16] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52:1–52:34, 2013.

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

[18] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, Rene Mueller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam J Storm, Yuanyuan Tian, Pinar Tözün, et al. Evolving databases for new-gen big data applications. In *CIDR*, 2017.

[19] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *Proceedings of the International Conference on Management of Data*, 39(1):20–26, 2010.

[20] P. Basanta-Val, N. C. Audsley, A. J. Wellings, I. Gray, and N. Fernández-García. Architecting time-critical big-data systems. *IEEE Transactions on Big Data*, 2(4):310–324, 2016.

[21] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI '15, pages 1431–1438, Palo Alto, California, 2015. AAAI Press.

[22] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[23] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical report, Microsoft, 2014. MSR-TR-2014–41.

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

[25] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *The VLDB Journal*, 3(1-2):232–243, 2010.

[26] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine. In *Proceedings of the International Conference on Management of Data*, SIGMOD'07, pages 1100–1102, New York, NY, USA, 2007. ACM.

[27] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[28] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

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

[30] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1201–1210. ACM, 2016.

[31] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the International Conference on Distributed and Event-based Systems*, DEBS '16, pages 69–80, New York, NY, USA, 2016. ACM.

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

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

[34] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[35] Gianpaolo Cugola and Alessandro Margara. Tesla: A formally defined event specification language. In *Proceedings of the International Conference on Distributed Event-Based Systems*, DEBS'10, pages 50–61, New York, NY, USA, 2010. ACM.

[36] Gianpaolo Cugola and Alessandro Margara. Complex event processing with t-rex. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

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

[38] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[39] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '07, pages 205–220. ACM, 2007.

[40] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.

[41] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. Rsp-ql semantics: A unifying query model to explain heterogeneity of rdf stream processing systems. *International Journal of Semantic Web & Information Systems*, 10(4):17–44, 2014.

[42] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. Modeling the execution semantics of stream processing engines with secret. *The VLDB Journal*, 22(4):421–446, 2013.

[43] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An update algorithm for distributed reactive programming. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[44] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications, 2010.

[45] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[46] American National Standard for Information Systems. Ansi x3.135-1992, database language sql, November 1992.

[47] Buğra Gedik. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience*, 44(9):1105–1128, 2014.

[48] Thanaa M Ghanem, Ahmed K Elmagarmid, Per-Åke Larson, and Walid G Aref. Supporting views in data stream management systems. *ACM Transactions on Database Systems (TODS)*, 35(1):1, 2010.

[49] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[50] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. Frames: Data-driven windows. In *Proceedings of the International Conference on Distributed and Event-based Systems*, DEBS '16, pages 13–24, New York, NY, USA, 2016. ACM.

[51] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[52] Annika Hinze and Agnès Voisard. Eva: An event algebra supporting complex event specification. *Information Systems*, 48:1–25, 2015.

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

[54] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the International Conference on Management of Data*, SIGMOD'16, pages 555–569, New York, NY, USA, 2016. ACM.

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

[56] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[57] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the International Conference on Management of Data*, SIGMOD '05, pages 311–322, New York, NY, USA, 2005. ACM.

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

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

[60] A. Margara and G. Salvaneschi. On the semantics of distributed reactive programming: the cost of consistency. *IEEE Transactions on Software Engineering (Preprint)*, 99(0):1–25, 2018.

[61] Alessandro Margara and Guido Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of the International Conference on Distributed Event-Based Systems*, DEBS '14, pages 142–153, New York, NY, USA, 2014. ACM.

[62] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 25(C):24–44, 2014.

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

[64] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[65] John Eliot Blakeslee Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1981.

[66] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[67] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[68] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the International Conference on Data Engineering*, ICDE '06, pages 49–, Washington, DC, USA, 2006. IEEE.

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

[70] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-store: A real-time oltp and olap system. *arXiv preprint arXiv:1601.04084*, 2016.

[71] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "all roads lead to rome": optimistic recovery for distributed iterative data processing. In *Proceedings of the International Conference on Information & Knowledge Management*, CIKM'13, pages 1919–1928, New York, NY, USA, 2013. ACM.

[72] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the International Conference on Distributed Event-Based Systems*, DEBS'09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[73] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)*, 29(2):394–403, 1982.

[74] Vivek Shah and Marcos Antonio Vaz Salles. Reactors: A case for predictable, virtualized actor database systems. In *Proceedings of the International Conference on Management of Data*, SIGMOD '18, pages 259–274. ACM, 2018.

[75] Michael Stonebraker. Newsql: An alternative to nosql and old sql for new oltp apps. *Communications of the ACM. Retrieved*, pages 07–06, 2012.

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

[77] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[78] 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. Storm@twitter. In *Proceedings of the International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[79] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, 2015.

[80] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems (TODS)*, 16(1):132–180, 1991.

[81] Gerhard Weikum and Hans-Jörg Schek. Concepts and applications of multilevel transactions and open nested transactions, 1992.

[82] Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. What is "next" in event processing? In *Proceedings of the Symposium on Principles of Database Systems*, PODS'07, pages 263–272, New York, NY, USA, 2007. ACM.

[83] Lotfi A Zadeh et al. Fuzzy sets. *Information and control*, 8(3):338–353, 1965.

[84] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 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, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

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

[86] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.