Data-Streaming and Concurrent Data-Object Co-design: Overview and Algorithmic Challenges

Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou(✉), and Philippas Tsigas

Chalmers University of Technology, Gothenburg, Sweden
ptrianta@chalmers.se

Abstract. Processing big volumes of data generated on-line, implies needs to carry out computations on-the-fly, in the streams of data. In parallel data-stream computing, the underlying data objects can provide the means for exchanging the data so that the communication and the work imbalance between the concurrent threads performing the computation are reduced, while the pipeline parallelism is enhanced. By shedding light on the concurrent data objects and their role as articulation points in data-stream processing, we place some cornerstones to analyze the problems, propose appropriate new data structures suitable for a set of functions and identify new key challenges to improve data-stream processing through co-design with fine-grain efficient synchronization combined with the data exchange.

It is interesting to point out that research in distributed computing on multiprocessor efficient and consistent data sharing through fine-grain synchronization emerged from questions in concurrent database-related research; approximately three decades since then, it is interesting to see several returns of the fruits of this expedition, helping with the new problems in the massive-data research domain, with applications in e.g. cyberphysical systems.

Keywords: Concurrent data structures · Data-streaming · Stream processing engines · In-memory data analysis

1 Introduction

Concurrent data objects are commonly described as implementations of Abstract Data Types (ADTs) shared by concurrent execution threads or processes. ADTs form abstractions of high re-usability across different applications and provide structured access to the data through their interface. The goals of the algorithmic implementation are about correctness and minimal complexity overhead of data-access, modification and retrieval.

One of the challenges in parallel and concurrent programs and applications, that also applies to their data objects, stems from the communication overhead, which needs to be minimized, too — besides the computational complexity—, so as to ensure that the underlying system's parallelism is properly utilized. Consequently, concurrent data structures need to integrate communication patterns,

DOI: 10.1007/978-3-319-24024-4_15

besides access patterns, that will serve the needs of the application domain that they will be used in.

In the journey from traditional methods for implementing shared objects through mutual exclusion to *lock-free* or *wait-free* algorithmic implementations, motivation and boost came from research communities focusing on the analysis and exploration of data. Nearly concurrently with the very important foundational steps in formalization of concurrency requirements in database-transactions [44,45,52], there came ideas of allowing concurrency in shared data object algorithmic implementations; e.g. first through proposing to safely allow read-only operations to execute concurrently with each-other in [14], later on with the seminal step by Lamport in [33], providing algorithms that allow concurrent reading and writing without assuming process synchrony or mutual exclusion, while also guaranteeing safety properties of the values obtained.

The above are all the more important in the new era of cyberphysical systems, with needs for computationally- and energy-efficient systems to analyse the big streams of data and extract useful information [18–20]. While the leveraging of concurrent algorithmic implementation of shared data objects might have been limited by the earlier times nature of analysis needs by applications —based on stored-data, rather than in-memory data— we now are in an era where data is generated in massive rates —e.g. by cyberphysical systems— and where in-memory analysis is needed to cope with such rates. Leveraging the concurrent data objects that best fit the needs of an application in a concurrent environment is a key issue, as highlighted by Michael in the "Balancing act of choosing non-blocking features" [40]. Moreover, quoting from [3], "*Not having to read from the disk and write computation results back saves hours to days of scientific work, giving scientists more time to investigate the data*". To process data in such a fashion, *data-streaming* is one of the new computation methodologies that have been proposed [2,5,8,9,21,25]. In data streaming, continuous queries, defined as Directed Acyclic Graphs (DAGs) of interconnected operators, are executed by Stream Processing Engines (SPEs) that process incoming data and produce results in a continuous fashion, without the necessity of storing the data. As highlighted in [16], parallelism is a necessity due to the low-latency and high-throughput requirements of such continuous real-time complex processing of increasingly large data volumes.

What are the shared objects that meet the needs of concurrent data streaming applications is an important issue that has only recently been brought for addressing in the literature [13]. By shedding light on the data structures, these recent findings place essential cornerstones in this study avenue and identify new key challenges to improve data streaming. In other words, what we observe is that in the journey where big data and concurrent data transactions meet concurrent data objects, there is a new crossing: data and computation meet in new forms and with new needs. In-memory, close-to-source analysis of data can provide useful information and services in cyberphysical systems. Data-streaming is significant in this context [20]. Concurrent data objects are a means to enhance the data streaming parallelism as needed, while new data objects and interfaces are required to be defined to meet the needs of data streaming.

Outline of the Paper. In the rest of this paper we outline the key points in the aforementioned findings, emphasizing the role of concurrent data objects serving as articulation points in data streaming and stream processing engines in particular. We highlight new challenges and the benefits that can be brought by the co-design of data stream processing and concurrent object co-design.

Section 2 provides a short overview of the evolution of concurrency in shared objects algorithmic implementations. Section 3 provides an introduction to data streaming and the requirements in synchronization and determinism in data processing. Subsequently, Sects. 4 and 5 elaborate on architectural aspects of Stream Processing Engines and the challenges in parallelization, in connection to the way that concurrent threads process and exchange data, in order to meet the above requirements. Section 6 introduces a new abstract data type, whose concurrent lock-free and linearizable implementation can allow threads to work efficiently in an asynchronous fashion and meet the requirements for synchronization and determinism in data stream processing. Section 7 provides an example evaluation of possible throughput and latency improvements in data streaming, by the use of the new shared object and its concurrent algorithmic implementation. Section 8 concludes with a discussion, pointing out also directions for further research.

2 Concurrent object Algorithmic Implementations - Preliminaries

As discussed in Sect. 1, shared objects have been traditionally implemented through mutual exclusion. The first steps in introducing true concurrency in shared object implementations [14,33] can be characterized as ideas and results that opened a big avenue in research. Setting the foundations for arguing about concurrency in shared object implementations that allowed concurrent access became an important next goal. In this context, we can find Lamport's definitions [34] of shared object implementations with progress guarantees (*wait-freeness*) and safety properties (*safeness, regularity, atomicity*) of such, describing the data consistency. The latter were proposed to formulate requirements for the object constructions and assumptions of the underlying system description. It was argued that they can e.g. describe consistency guarantees of asynchronous hardware. Related here is also the work by Misra [42], formulating axioms for memory access in asynchronous hardware systems, and by Lynch and Tuttle [38] setting foundations for hierarchical correctness proofs for distributed algorithms through automata and executions on them. Another significant step has been the formulation of *linearizabilty* as correctness condition for concurrent objects, by Herlihy and Wing [31].

For ease of reference, we paraphrase here the definitions of some of the key terms, that are also used later in the paper. A *wait-free* object implementation ensures that any operation on the object can complete in bounded number of steps, independently of other contending processes. A relaxed condition is lock-freedom: a *lock-free* object implementation ensures that at least one of the contending operations makes progress in a finite number of its own steps. These properties are often referred to in the literature as *non-blocking.* An implementation of an object is *linearizable* if each operation execution appears to take effect at some point (linearization point) between its invocation and response; thus, given an execution of concurrent operations and by using the linearization points, it should be able to define a total order of the operations, which is consistent with their real-time ordering and their effects are consistent with the sequential semantics of the data structure.

Following the foundations, there was a "movement" in the scientific community, providing challenging algorithmic implementations of shared data objects allowing concurrency and guaranteeing a variety of safety properties (linearizability or weaker forms) and progress properties (e.g. wait-freeness, lock-freeness, obstruction-freeness); we refer the reader to [6,11,29,37] and references therein, for overview and more detailed presentations of key results.

In parallel, the hardware point of view is also worthwhile to comment on. At the first stages, the concept of asynchronous parallel hardware was mainly studied from a theoretical point of view, with the exception of some elegant efforts such as the work by the group of Ebergen [15,46]. Similar has been the perspective of massive parallelism, until the relatively recent era of multicore and manycore systems in hardware. The latter triggered a new "movement", that brought changes including the consideration of new abstractions, most notably the one of transactional memory [28,47] as well as the deeper consideration of asynchronous hardware [26]. This evolution makes the need for asynchronous concurrent implementations of shared objects even more significant.

3 Data Streaming - Preliminaries

In this section, we introduce basic concepts of the data streaming processing paradigm. We also illustrate them through a sample data streaming continuous query that analyzes traffic gathered from the *SoundCloud* [48] social network. Besides, we overview the evolution of Stream Processing Engines (SPEs) from centralized to parallel-distributed ones, also introducing the definition of *deterministic processing* (also referred to as *semantic transparency* [22]).

Data Streaming Model. A data stream S is an unbounded sequence of tuples sharing a given schema composed by attributes $\langle ts, A_1, A_2, \ldots, A_n \rangle$. We refer to attribute A_i of tuple t as $t.A_i$. Attribute $t.ts$ represents the time when the tuple is created. As common in the literature [12], we assume that tuples generated by a given data source and delivered through the same data stream have non-decreasing ts values. We also suppose data sources have clocks that are well-synchronized using a clock synchronization protocol like NTP [41].

Table 1 presents a sample schema, composed of four attributes, for tuples carrying comments (exchanged by users in relation to songs) from the SoundCloud platform.

It should be noted that tuples belonging to the same *logical* data stream, sharing the same schema and carrying similar information (e.g., comments about

Table 1. Sample tuple schema.

Attribute	Content
ts	The creation timestamp of the tuple
user	The user commenting a song
song	The song to which the comment refers to
comment	The comment itself

songs, as in Table 1), might be delivered by multiple distinct *physical* streams (e.g., generated by crawlers running at different physical nodes). As we explain in the following, such distinction plays an important role if tuples must be processed deterministically. In the remainder, we refer to logical streams simply as streams, specifying explicitly when physical streams are in focus.

In the data streaming model, input tuples coming from one or multiple input streams are consumed by *Continuous Queries* (or simply queries in the following), which subsequently produce one or more output streams. A query, defined as a directed acyclic graph (DAG) with additional input and output edges, produces results "continuously" while consuming input tuples. Vertexes represent operators that consume tuples (from at least one input stream) and produce output tuples (for at least one output stream). Edges define how input and output tuples flow among the operators of a query.

Data Streaming Operators. Data streaming operators, the base unit used to process and produce tuples, are classified depending on whether they maintain a state that evolves accordingly with the input tuples being processed. *Stateless operators* such as *Map*, *Filter* and *Union* do not maintain such a state and perform a one-by-one processing of input tuples. On the other hand, *stateful operators* such as *Aggregate* and *Join* maintain a state and process multiple input tuples in order to produce one output tuple. Due to the unbounded nature of data streams, stateful computations are usually performed over *sliding windows* covering portions of the input tuples. Time based windows are defined over period of times (e.g., tuples received in the last 10 min) while tuple based windows are defined over the number of stored tuples (e.g., last 50 received tuples).

Continuous Query Example. Let us take a look at a sample continuous query that consumes the tuples sharing the schema presented in Table 1 to count the number of positive comments (i.e., comments containing certain predefined keywords) exchanged by users in relation to each song. As presented in Fig. 1[1], the query is composed by three operators.

An initial *Map* operator transforms each comment (attribute *comment* into a stream of words typed by users in relation to songs. Its resulting output stream schema is composed by attributes $\langle ts, song, word \rangle$. Subsequently, a *Filter* operator is used to forward only the words that belong to a given subset of positive

[1] Tuples shown in this example are not extracted from SoundCloud, but handcrafted for the specific example.

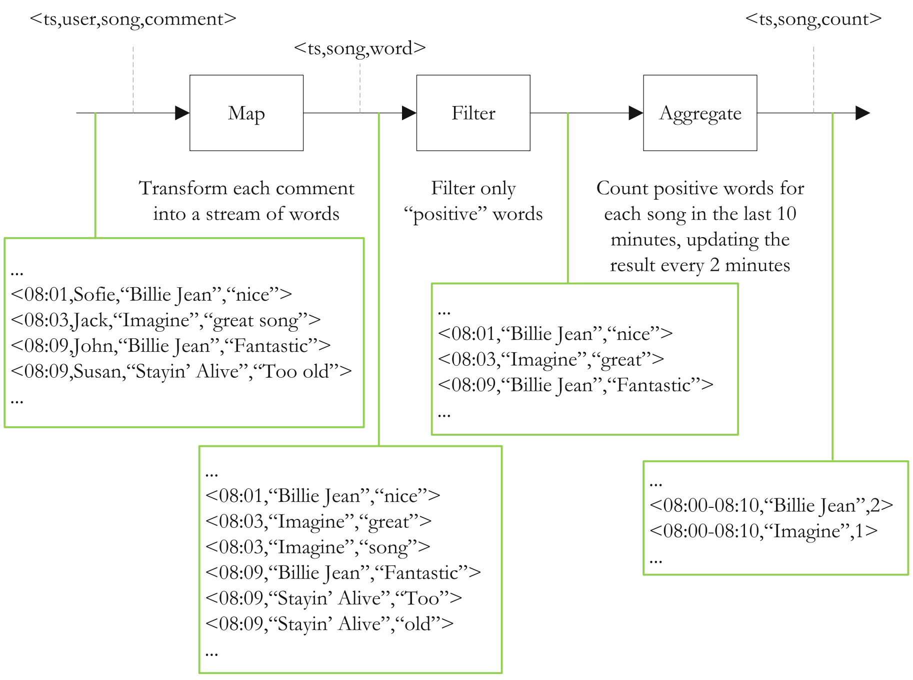
<ts,user,song,comment>
<ts,song,word>
<ts,song,count>
Map
Filter
Aggregate
Transform each comment into a stream of words
Filter only "positive" words
Count positive words for each song in the last 10 minutes, updating the result every 2 minutes
...
<08:01,Sofie,"Billie Jean","nice">
<08:03,Jack,"Imagine","great song">
<08:09,John,"Billie Jean","Fantastic">
<08:09,Susan,"Stayin' Alive","Too old">
...
<08:01,"Billie Jean","nice">
<08:03,"Imagine","great">
<08:03,"Imagine","song">
<08:09,"Billie Jean","Fantastic">
<08:09,"Stayin' Alive","Too">
<08:09,"Stayin' Alive","old">
...
<08:01,"Billie Jean","nice">
<08:03,"Imagine","great">
<08:09,"Billie Jean","Fantastic">
...
<08:00-08:10,"Billie Jean",2>
<08:00-08:10,"Imagine",1>

Fig. 1. Sample query that consumes tuples sharing the schema in Table 1 and counts the number of positive comments exchanged by users in relation to each song.

words (e.g., nice, great or fantastic). The output tuples produced by this operator share the same schema of its input tuples. Finally, an *Aggregate* operator is used to count, for each song, how many positive words are received over a sliding window of 10 min and to produce a new result every 2 min.

3.1 Parallel Data Streaming and Deterministic Processing

As emphasized in [16], real-time continuous processing of large volumes of data demand for low-latency and high throughput processing. During the last decade, such increasing demand drove the evolution of SPEs from centralized [2,5,9] to distributed [1] and to parallel-distributed ones [17,22,23]. As shown in Fig. 2 (in relation to the sample query of Fig. 1), queries are entirely deployed and run by exactly one SPE instance when the latter is centralized. By providing *inter-operator* parallelism, distributed SPEs allow for the execution of different operators belonging to the same query at different SPE instances. Finally, by providing *intra-operator* parallelism, parallel-distributed SPEs also allow for a single operator to be executed at multiple SPE instances. For simplicity, the figure shows distinct SPE instances running at distinct physical machines. Nevertheless, multiple SPE instances can be deployed within the same physical node (e.g., to leverage multi-core architectures).

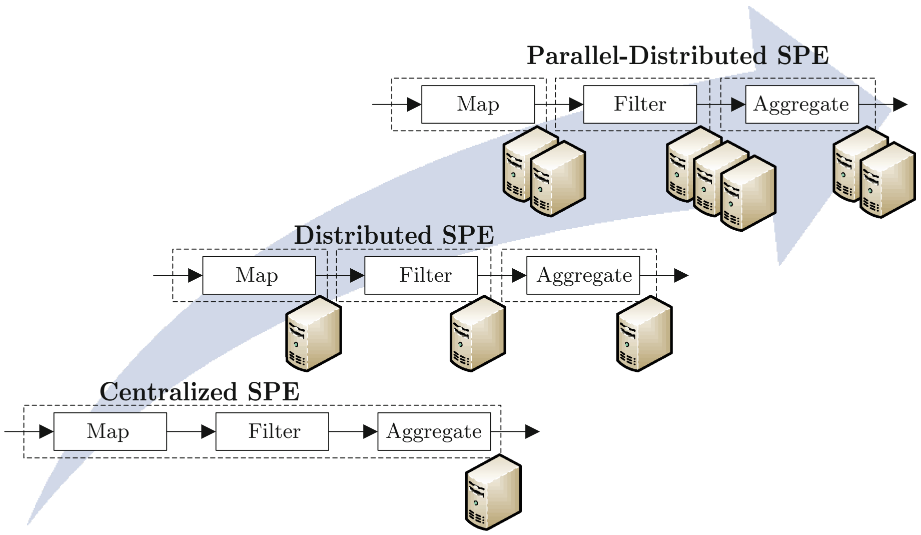

Fig. 2. Evolution of Stream Processing Engines from centralized to distributed and to parallel-distributed ones.

The parallel execution of data streaming operators (and thus of the queries they compose) is the only means for a single operator to avoid to get overloaded because of its volume of input data; the latter would of course be unwanted as it would degrade the performance of the entire data streaming query. Challenging aspects in the design and implementation of parallel data streaming operators do not only aim at improving their performance, but also aim at preserving their semantic.

Definition 1. *[22, 23] The property of semantic transparency or deterministim in parallel stream processing guarantees that, by consuming the tuples delivered by a given set of physical input streams, a parallel operator produces exactly the same output that would be produced by its centralized counterpart.*

As explained in [13], a condition to enforce deterministic processing for the operators of a query is to process tuples delivered by distinct physical streams in timestamp-order (that is, to process them deterministically independently of their inter-arrival times). In the context of parallel-distributed SPEs, the distinct physical input streams of an operator do not only refer to the physical streams generated by distinct sources, but also to the distinct physical streams generated by the multiple instances of a parallel operator. In the example in Fig. 1, distinct physical streams could be delivered to the Map operator by distinct sources, while distinct physical streams could be delivered to the aggregate from the multiple SPE instances running the Filter operator in parallel. In [12], the authors introduce the concept of *ready* tuple as follows:

Definition 2. *Let t_i^j be the i-th tuple in timestamp-sorted physical stream j. Tuple t_i^j is ready to be processed if $t_i^j.ts \leq merge_{ts}$, where $merge_{ts}$ is the minimum among the latest timestamps from each timestamp-sorted physical stream j, i.e. $merge_{ts} = min_j\{max_i(t_i^j.ts)\}$*

Based on this definition, deterministic processing is enforced if operators consume timestamp-sorted *ready* tuples from their physical input streams.

It should be noticed that the way in which input tuples are distinguished between the ones that are *ready* and the ones that are not is not orthogonal to the data structures used to maintain such tuples. Naïve solutions that rely on data structures oblivious to the concept of *ready* tuples bound the parallelism and concurrency degree of the analysis, usually incur in high processing costs and introduce processing bottlenecks. On the other hand, streaming-aware data structures enable for finer-grained and scalable cooperation among the different threads operating on them, as we will explain in Sect. 5.

4 Inter-thread Communication in SPEs Architecture

In this section, we present the common architecture of a SPE, focusing especially on the data structures defined for each SPE instance to maintain the tuples being consumed and produced in a deterministic fashion and the threads operating on them. For the clarity of the discussion, we illustrate the architecture of a SPE considering a single query consuming one input stream and producing one output stream, and do not overview the data structures internal to the query's operators (maintaining partial computations and windows). While overviewing the architecture, we also discuss its limitations motivating the discussion that follows, in Sect. 5.

Common SPE Architecture. The common architecture of a SPE (presented in Fig. 3) usually defines three main modules, which we refer to as M_{in}, M_{proc} and M_{out}[1,2,22,36][2]. Module M_{in} maintains the queues in which tuples from a given set of physical input streams $I_1, \ldots, I_n$ are collected from the network. The collection of such tuples is usually performed by a dedicated thread, which we refer to as T_{in}, running the *add* method. Tuples from each physical input stream can be maintained at individual queues [22,23] or concurrent data structures such as the LMAX Disruptor [50] (as for the Storm [36] SPE).

Tuples stored at M_{in} are subsequently copied to M_{proc} and consumed by the different data streaming operators composing the query run by the SPE. More concretely, a dedicated thread T_{proc}:

1. copies the tuples from each physical input stream (method *copy*),

[2] Complementary modules, not in the scope of this discussion, might be defined for features such as fault tolerance, scheduling, balancing or self-provisioning and self-decommissioning.

2. merges the timestamp-sorted physical streams into a single timestamp-sorted logical stream of tuples in order for the query to process tuples deterministically (method *merge*),
3. processes them (method *process*) and, finally,
4. stores the resulting output tuples in a dedicated queue (method *store*).

Each time an output tuple is produced, a dedicated thread T_{out} copies it to the queue maintained at the M_{out} module (method *copy*) and, finally, forwards it to other SPE instances or to the external user applications (method *forward*)[3].

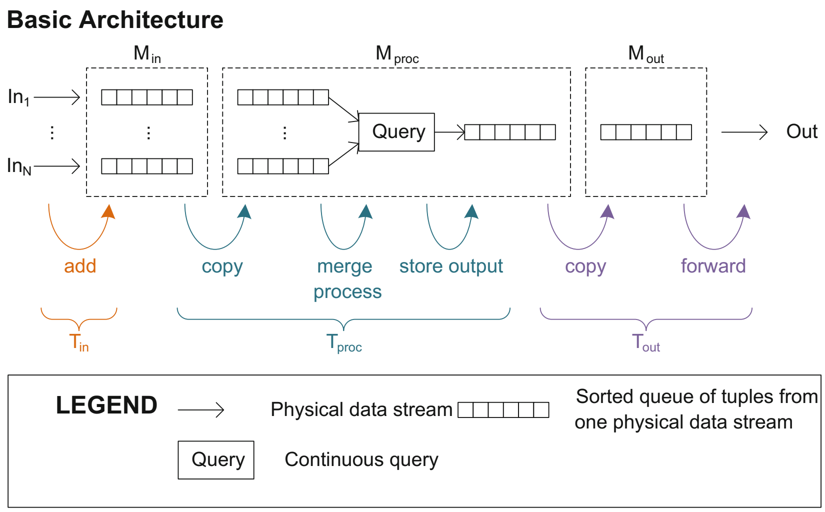

Fig. 3. Basic architecture of a SPE, presenting the different modules and threads operating on them.

Enforcing Deterministic Processing. As presented in Fig. 3, the execution of the method *process* is preceded by the execution of the method *merge* for thread T_{proc}. The merging of the physical streams referring to the same logical stream of tuples fed to the query is performed in order to enforce deterministic process (as discussed in Sect. 3.1). The merging of the physical input streams can happen in different ways. On one hand, each new tuple from a physical stream I_i can be compared with the ones previously received from any physical stream $I_j | j \neq i$ in order to identify the *ready* tuples and process them. Such aggressive merging, performed by operators such as the Transparent Input Merger [22] or the SUnion operator [7], results in low processing latency, as each tuple is processed as soon as it becomes ready. On the other hand, the tuples from the different physical streams can also be sorted periodically, by using punctuation tuples [49] or by performing the sorting each time a given time period expires.

3 Depending on how the data structures in modules M_{in}, M_{proc} and M_{out} are defined, locking mechanism can be in place, as in [23].

With respect to the output tuples produced by the query running at the SPE, the method *forward* is executed by exactly one thread in order for the output tuples to be delivered in timestamp order to the following SPE instance or to the end user application.

Shortcomings. The architecture outlined above results in several shortcomings, as we explain in the following.

First, by keeping dedicated threads for the different stages, it is prone to unbalanced work (and thus does not leverage at their full potential the available threads). As it can be observed, threads T_{in} and T_{out} perform a reduced set of methods (*add*, *copy* and *forward*) with respect to thread T_{out}. Moreover, such operations are light when compared with computational intensive operations such as *merge* and *process* (the latter depending on the number of operators composing the query run by the SPE).

Secondly, *copy* operations are performed by both the T_{in} and T_{out} threads in order to retrieve the tuples from their preceding modules. As discussed in [4], the copies required to maintain tuples at different queues incur in a significant cost, which in turns affects the performance of the SPE.

Finally, independently of whether tuples are sorted periodically or upon reception of each new incoming tuple, the merging techniques proposed in the literature [7,22] usually have a processing cost that grows linearly in the number of physical streams being merged. Moreover, the merging itself constitutes a *potential bottleneck* for the entire system (whose throughput is bounded by the speed with which input tuples can be merged) and it limits the pipelining of the operations performed by the different threads. That is, independently of the rate at which input tuples are retrieved by thread T_{in}, as long as available tuples are not merged, no tuple can be processed by thread T_{proc} and, consequently, no new result can be forwarded by the thread T_{out}.

5 Leveraging Concurrent Data Structures in SPEs

As we discuss in this section, three main actions can be undertaken to balance the work of the different threads running within a SPE instance and thus enhance its inner concurrency and maximize its performance. As we complement in our evaluation, such actions result in a significant performance improvement for the throughput of a SPE.

Switch from Inter-module to Intra-module Data Access. The first action towards an improved architecture is for modules to share the data structures where tuples retrieved from the network are maintained before being fed to the query. While the thread(s) operating on such modules can be in charge of different tasks (either retrieving tuples from the network or consume input tuples), the joint access prevents unnecessary copies of tuples across modules M_{in} and M_{proc}. In this context, fine-grained synchronization mechanisms should be defined for threads M_{in} and M_{proc} intercommunication.

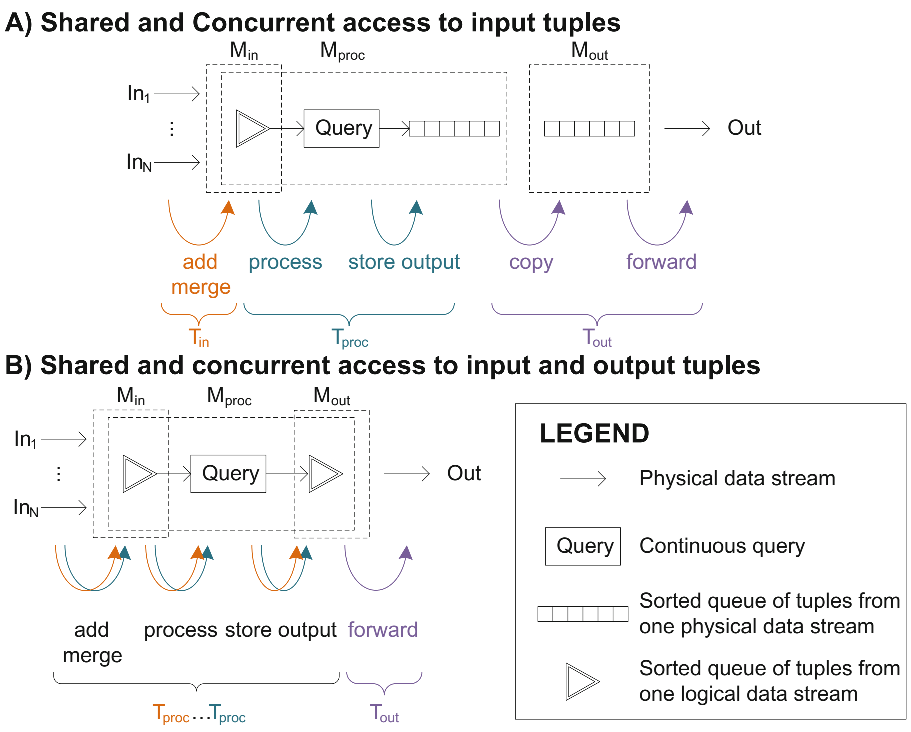

Fig. 4. SPE architectures leveraging concurrent data structures.

Merge Physical Streams Concurrently. As discussed in Sect. 3.1, the tuples delivered by timestamp-sorted physical input stream streams of a query must be merged and fed in timestamp order in order to enforce deterministic processing. Hence, the second action is to enable for the merging of tuples delivered by physical streams to happen concurrently with their processing (as presented in Fig. 4A). This approach provides a better balancing of the threads work by shifting the *merge* operation to thread T_{in}.

Consume Logical Streams Concurrently. The third action aims at overcoming the potential bottleneck caused by merging the tuples delivered by the distinct physical input streams. This can be achieved by relying on a concurrent data structure that not only enables for concurrent addition of tuples (being merged) and retrieval of tuples (being processed), but also allows for such operations to be performed by arbitrarily number of processing threads. Such an architecture allows the parallelism degree of a SPE instance to grow beyond the number of threads usually defined by the latter. In such a case, though, the synchronization is no only required for the merging of input tuples, but also for the merging of output ones (as presented in Fig. 4B). That is, since multiple physical output streams are produced by the processing threads running at the

SPE, the former must be merged deterministically into a single logical stream. As we explain in [12,24], the way in which input tuples are processed by the different threads depends on the semantics of the operators running at the SPE instance.

6 ScaleGate: A Novel, Concurrency- and Streaming-Aware Data Object

As discussed in Sect. 5, a shared data structure that enables for concurrent addition and merging of tuples delivered by multiple physical streams, while also allowing for an arbitrary number of threads to retrieve *ready* tuples in timestamp order, is the key to balance the workload of an arbitrary number of processing threads running in a SPE instance. In the following, we overview such a data structure, *ScaleGate*, focusing on its functionality and core ideas and also presenting its interface. We refer the reader to [24] for its implementation details.

Overview and Interface. The common architecture of SPEs outlined in Sect. 4, can be seen as a pipeline where data is continuously produced, processed and consumed across the different stages, in this case the three main modules. In a parallel implementation, each computational thread associated with one or more modules will communicate with the rest by accessing and/or modifying the shared data structures, which are the focal point of this section. The ideal concurrent data structures should organize the data so that the communication cost and computational complexity of each access is minimized while the parallelism within the modules and the overall system efficiency is maximized. Moreover, this should be done under an interface that provides semantics that enhance the parallelism across modules.

ScaleGate is a recently proposed abstract data type that becomes a cornerstone in achieving the parallelization challenges presented in the previous section. *ScaleGate* guarantees properties essential for concurrently merging physical streams at the articulation points where data and threads meet, while it integrates the necessary synchronization for allowing multiple threads to consume *ready* tuples concurrently. It allows for an arbitrary number of timestamp-sorted streams, each delivered by a *source* thread, to be merged into a timestamp-sorted stream of *ready* tuples (Definition 2). At the same time, it allows for an arbitrary number of *reader* threads to consume in timestamp order all the *ready* tuples of the resulting timestamp-sorted stream. *ScaleGate* integrates, in a decentralized manner, the necessary communication between the source and reader threads in order to decide whether a tuple is *ready* or not. The interface of *ScaleGate* provides the following methods:

– addTuple(timestamp,tuple,sourceID): which allows a tuple from the source thread sourceID to be merged by *ScaleGate* in the resulting timestamp-sorted stream of *ready* tuples.

- getNextReadyTuple(readerID): which provides to the calling reader thread readerID the next earliest *ready* tuple that has not been yet consumed by the former.

Algorithmic Design for Concurrent Implementation of *ScaleGate*. As explained earlier in the paper, synchronization is one of the fundamental design considerations for a concurrent data structure implementation. Lock-free (a.k.a. non-blocking) implementations ensure system-wide progress, by guaranteeing at least one of the threads operating on the data structure to make progress independently of the behavior of other threads. Following the expectations based on their basic properties, such implementations demonstrate higher scalability and better fairness when compared with coarse- or fine-grain locking mechanisms [10,35,39]. This behavior remains across several multiprocessor hardware architectures, with varying characteristics such as uniform/non-uniform memory access, or memory hierarchies. All the above contribute to the choice in [24], for lock-free algorithmic implementation of the *ScaleGate*.

A basic requirement for an algorithmic implementation of the *ScaleGate* is to maintain items in a sorted manner. Tree-like implementations, especially balanced ones, have not been proven efficient in concurrent environments due to the strong dependencies that appear in balancing operations [30]. On the contrary, shared concurrent skip lists [27,51] have been used extensively for such requirements. In a nutshell, skip lists maintain a sorted linked list of elements (e.g., tuples), while allowing for probabilistically logarithmic concurrent insertions of new elements and the concurrent deletion of existing ones. This is made possible by multiple levels (pointers), for each element, that act as shortcuts for quickly locating the appropriate insertion position of a new element. The number of additional levels for each element is chosen randomly during its allocation.

Inspired by skip lists, the *ScaleGate* algoritmic implementation incorporates a multi-level pointer mechanism adapted to its requirements. Such adaption aims at enabling fine-grained synchronization that boosts parallelism and is carried out (1) by making *ScaleGate* inherently aware of the concept of *ready* tuples and (2) by exploiting the specific access patterns of *ready* tuples (e.g. consumed in timestamp order by the threads executing the queries) and thus allowing for a more lightweight implementation than the general purpose delete operations of skip lists (such operations carry a considerable overhead in the respective implementations).

Claim. The concurrent implementation of *ScaleGate* in [12,24] follows the above elements and satisfies strong safety and liveness requirements, namely linearizability and lock-freedom. Also, as shown in [12,24], *ScaleGate* enables deterministic execution of data streaming operators.

7 Evaluation Study

In this section we describe an experimental study of how concurrent data structures can enhance the performance of SPE by finer-grained synchronization among the threads operating on the tuples. This is not meant to be a thorough evaluation of the proposed data structure and stream processing engine designs. Some detailed experimental studies for a range of operators and input streams that vary in character and volumes can be found in [12,13,24].

In particular, here we show an evaluation of the performance for the query introduced in Sect. 3, in different SPE architectures presented in Figs. 3 and 4. More concretely, we measure both their per-tuple processing time (in μs) and overall throughput (in tuples/second, t/s). We begin by discussing the evaluation setup.

Experiment Setup

This evaluation study has been run with a workstation equipped with a 2.0 GHz Intel Xeon E5-2650 (16 cores over 2 sockets) and 64 GB of memory. The different data structures and modules of SPEs' architectures have been implemented in Java. We use a dataset, which we refer to as *SC*, collected from the online audio distribution platform SoundCloud from a subset of approximately 40, 000 users exchanging comments about 250, 000 songs between 2007 and 2013. Tuples contain comments sent by users in relation to songs and are composed by the attributes $\langle ts, user, song, comment\rangle$.

We fed tuples from the *SC* dataset into the query presented in Sect. 3, which counts the number of positive comments (a comment is considered as positive if it contains keywords such as nice, great, fantastic and so on) in relation to each song given a window of size 10 min and advance 2 min. In all the experiments, we assume input tuples are delivered to the query by 20 distinct physical input streams.

We refer to the basic architecture (Fig. 3), the architecture defining shared and concurrent access to the input tuples (Fig. 4A) and the one defining shared and concurrent access for both input and output tuples (Fig. 4B) as architectures A_1, A_2 and A_3, respectively. In all the experiments, we measure the average per-tuple processing time based on the operation performed by threads T_{in} and T_{proc} in the different architectures. Subsequently, we compute the maximum expected throughput based on such per-tuple processing time. Since both threads T_{in} and T_{proc} perform the same operations for architecture A_3, we refer to the threads as T_1 and T_2 in all the experiments. All the presented results are averaged over 100 runs. In all the experiments, we do not take into account the per-tuple processing time incurred by the thread T_{out} in order to forward each output tuple to the following SPE instance or the external end-user application.

Illustrative Outcome and Discussion. In the following, we present the results for the three different setups we considered, namely A_1, A_2 and A_3. Results about the per-tuple processing time are summarized in Fig. 5 while throughput results are summarized in Fig. 6.

Architecture A_1 *(Fig. 3).* In the first experiment, we consider the SPE architecture presented in Fig. 3, in which thread T_{in} is in charge of the *add* operation while thread T_{proc} is in charge of the *copy*, *merge*, *process* and *store* operations.

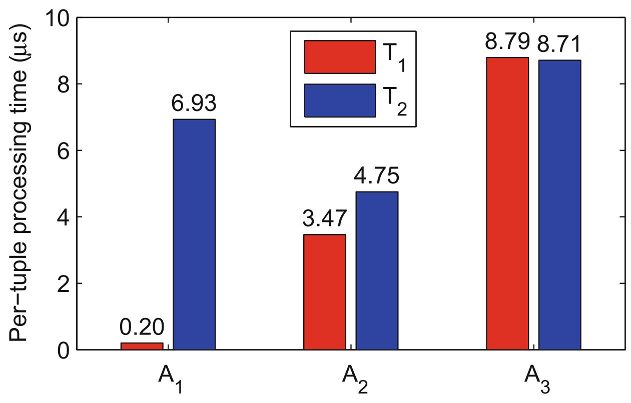

Fig. 5. Per-tuple average processing time for threads T_1 and T_2 and the different architectures A_1, A_2 and A_3.

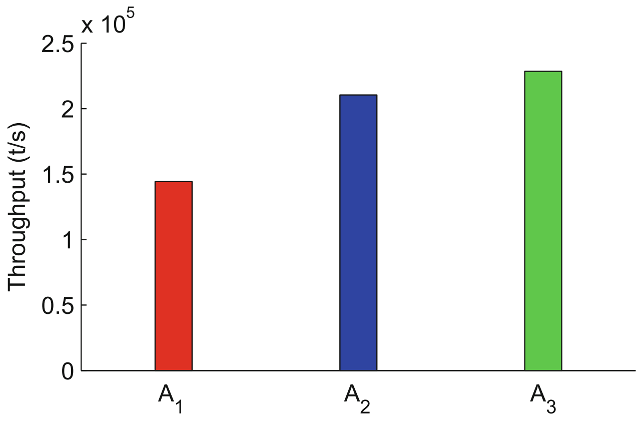

Fig. 6. Throughput achieved by the different architectures A_1, A_2 and A_3.

As presented in Fig. 5 and explained in Sect. 4, the two threads are heavily unbalanced. Thread T_{in} is responsible for a single, lightweight operation which on average takes 0.2 ms to be run. At the same time, thread T_{proc} is responsible for merge sorting all the input tuples and process them, which on average takes 6.9 ms (that is, the per-tuple processing time for thread T_{proc} is 35 times higher than the one for thread tone). For such a setup, the operations run by thread T_{proc} result in a bottleneck of the system, allowing for a maximum throughput of $150,000$ t/s.

Architecture A_2 (Fig. 4A). For this architecture, the *merge* operation is now performed by thread T_{in} rather than T_{proc}. While the work between the two threads is still unbalanced, it can be observed that thread T_{in} per-tuple processing now grows to approx 3.5 ms while thread T_{proc} per-tuple processing time decreases to 4.8 ms. As a result, while still containing a bottleneck, this setup allows for an increased throughput, up to 200,000 t/s (thus increasing the baseline A_1 throughput by 33 %).

Architecture A_3 *(Fig. 4B).* For this architecture, the operations *add*, *copy*, *merge*, *process* and *store* are not partitioned among the two threads T_{in} and T_{proc} but rather executed (all of them) by both threads in a parallel and concurrent fashion. As we explained in Sect. 5, while such an architecture requires extra synchronization in order for the output tuples produced by each thread to be merged into a single logical stream (in order to enforce deterministic processing), it allows for the parallel and concurrent execution of an arbitrary number of processing threads (as long as the merging of their output tuples does not constitute a bottleneck).

As presented in Fig. 5, the per-tuple processing time incurred by both threads grows to approximately 8.7 ms. As only one of them is needed in order to process a tuple and each of them is able to process 115000 t/s, independently from the other, the overall throughput of the system grows to 230000 t/s (thus increasing the baseline A_1 throughput by more than 50 %).

Overview Comment. The significant improvements are achieved through more balanced work among the threads and the possibility for each thread to make progress asynchronously and nearly independent of the progress of the other threads. These are enabled through the*ScaleGate* concurrent object, as expected.

8 Conclusions

In this paper we give an overview of the motivation and the idea of co-design of data stream processing and concurrent data structures. We point out that abstract data types and their concurrent implementations play a key role in data streaming efficiency and we give examples of newly proposed ones that can offer significant benefits. Symmetrically, we show that stream processing designs can benefit significantly by awareness of the concurrency in the "articulation" points, where data and computation "meet". This is all the more important given the needs for processing in the continuously increasing data volumes in e.g. cyber-physical systems [20], where data streaming becomes a must in order to extract efficiently useful information from the data and justify the existence of these systems.

Continued research in this new space is expected to have significant impact, both from the point of view of concurrent algorithmic challenges that are brought in, as well as from the point of view of usefulness in actual applications' needs. Possible topics include efficiency and consistency (and their trade-offs) in the context of intra-node and inter-node concurrency for data streaming operators, as well as in the context of operations such as range queries, possibly through snapshots or iterations on these objects [32,43]; the analysis of data transformation pipelines (with a query) given the progress properties of the concurrent object implementations; new abstract data types and efficient concurrent implementations that can improve such pipelines.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR, pp. 277–289 (2005)
2. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. VLDB J. **12**, 12–139 (2003)
3. Ailamaki, A., Kantere, V., Dash, D.: Managing scientific data. Commun. ACM **53**(6), 68–78 (2010)
4. Akram, S., Marazakis, M., Bilas, A.: Understanding and improving the cost of scaling distributed event processing. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS 2012, pp. 290–301. ACM, New York (2012)
5. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: the stanford data stream management system. Book chapter (2004)
6. Attiya, H., Welch, J.: Distributed Computing: Fundamentals. Simulations and Advanced Topics, Wiley Online Library (2004)
7. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the Borealis distributed stream processing system. ACM Trans. Database Syst. **33**(1), 3 (2008)
8. Callau-Zori, M., Jiménez-Peris, R., Gulisano, V., Papatriantafilou, M., Fu, Z., Patiño Martínez, M.: Stone: a stream-based ddos defense framework. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, pp. 807–812. ACM (2013)
9. Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams: a new class of data management applications. In: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB 2002. VLDB Endowment (2002)
10. Cederman, D., Chatterjee, B., Nguyen, N., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: A study of the behavior of synchronization methods in commonly used languages and systems. In: IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS) (2013)
11. Cederman, D., Gidenstam, A., Ha, P., Sundell, H., Papatriantafilou, M., Tsigas, P.: Lock-free concurrent data structures (2013). arXiv:1302.2757
12. Cederman, D., Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: Concurrent data structures for efficient streaming aggregation. Technical report, Chalmers University of Technology (2013)
13. Cederman, D., Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: Brief announcement: concurrent data structures for efficient streaming aggregation. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2014, pp. 76–78 (2014)
14. Courtois, P.-J., Heymans, F., Parnas, D.L.: Concurrent control with readers and writers. Commun. ACM **14**(10), 667–668 (1971)

15. Ebergen, J.: Circuits without clocks: what makes them tick? In: Papatriantafilou, M., Hunel, P. (eds.) OPODIS 2003. LNCS, vol. 3144, pp. 2–2. Springer, Heidelberg (2004)
16. Gedik, B., Bordawekar, R.R., Philip, S.Y.: Cell Join: a parallel stream join operator for the cell processor. VLDB J. **18**, 501–519 (2009)
17. Gulisano, V.: StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine. Ph.D. thesis, Universidad Politécnica de Madrid (2012)
18. Gulisano, V., Almgren, M., Papatriantafilou, M.: Metis: a two-tier intrusion detection system for advanced metering infrastructures. In: Proceedings of the 5th International Conference on Future Energy Systems, e-Energy 2014, pp. 211–212. ACM (2014)
19. Gulisano, V., Almgren, M., Papatriantafilou, M.: Online and scalable data validation in advanced metering infrastructures. In: Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2014 IEEE PES, pp. 1–6 (2014)
20. Gulisano, V., Almgren, M., Papatriantafilou, M.: When smart cities meet big data. ERCIM News. Smart Cities, p. 40 (2014)
21. Gulisano, V., Jimenez-Peris, R., Patiño-Martinez, M., Soriente, C., Valduriez, P.: A big data platform for large scale event processing. ERCIM News **2012**(89), 2 (2012)
22. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: an elastic and scalable data streaming system. IEEE Trans. Parallel Distrib. Syst. **99** (2012)
23. Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Valduriez, P.: Streamcloud: a large scale data streaming system. In: ICDCS 2010: International Conference on Distributed Computing Systems (2010)
24. Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join enabled by concurrent data structures. Technical report, Chalmers University of Technology (2014)
25. Gulisano, V., Nikolakopoulos, Y., Walulya, I., Papatriantafilou, M., Tsigas, P.: DEBS grand challenge: deterministic real-time analytics of geospatial data streams through scalegate objects. In: DEBS 2015: the 9th ACM International Conference on Distributed Event-Based Systems (2015)
26. Hardavellas, N., Ferdman, M., Falsafi, B., Ailamaki, A.: Toward dark silicon in servers. IEEE Micro. **31**(EPFL-ARTICLE-168285), 6–15 (2011)
27. Herlihy, M.P., Lev, Y., Luchangco, V., Shavit, N.N.: A simple optimistic skiplist algorithm. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 124–138. Springer, Heidelberg (2007)
28. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA 1993, pp. 289–300. ACM, New York (1993)
29. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, Boston (2008)
30. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier, Revised Reprint (2012)
31. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
32. Kirousis, L.M., Spirakis, P.G., Tsigas, P.: Reading many variables in one atomic operation: solutions with linear or sublinear complexity. IEEE Trans. Parallel Distrib. Syst. **5**(7), 688–696 (1994)
33. Lamport, L.: Concurrent reading and writing. Commun. ACM **20**(11), 806–811 (1977)

34. Lamport, L.: On interprocess communication. Part I: basic formalism. Distrib. Comput. **1**(2), 77–85 (1986)
35. Liu, Y., Zhang, K., Spear, M.: Dynamic-sized nonblocking hash tables. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC 2014. ACM (2014)
36. LMax Disruptor. https://lmax-exchange.github.io/disruptor/
37. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
38. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10–12, 1987, pp. 137–151 (1987)
39. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002. ACM (2002)
40. Michael, M.M.: The balancing act of choosing nonblocking features. Commun. ACM **56**(9), 46–53 (2013)
41. Mills, D.L.: A brief history of ntp time: memoirs of an internet timekeeper. Comput. Commun. Rev. **33**, 9–21 (2003)
42. Misra, J.: Axioms for memory access in asynchronous hardware systems. ACM Trans. Program. Lang. Syst. **8**(1), 142–153 (1986)
43. Nikolakopoulos, Y., Gidenstam, A., Papatriantafilou, M., Tsigas, P.: A consistency framework for iteration operations in concurrent data structures. In: IEEE 29th International Symposium on Parallel and Distributed Processing (IPDPS) (2015)
44. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
45. Papadimitriou, C.H.: The Theory of Database Concurrency Control. Computer Science Press, Rockville (1986)
46. Papatriantafilou, M., Hunel, P. (eds.): OPODIS 2003. LNCS, vol. 3144. Springer, Heidelberg (2004)
47. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1995, pp. 204–213. ACM, New York (1995)
48. SoundCloud. https://soundcloud.com/
49. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 263–274. ACM, New York (2004)
50. Storm project. http://storm.incubator.apache.org/
51. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. **65**, 609–627 (2005)
52. Tuzhilin, A., Spirakis, P.G.: A semantic approach to correctness of concurrent transaction executions. In: Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS 1985, pp. 85–95. ACM, New York (1985)