Future Generation Computer Systems 88 (2018) 297–308

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Viper: A module for communication-layer determinism and scaling in low-latency stream processing☆

Ivan Walulya *, Dimitris Palyvos-Giannas, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, Philippas Tsigas

Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

ARTICLE INFO

Article history:
Received 17 November 2017
Received in revised form 21 March 2018
Accepted 27 May 2018

Keywords:
Data streaming
Determinism
Low-latency
Shared-nothing and shared-memory parallelism
Stream processing engines
Data parallelism

ABSTRACT

Stream Processing Engines (SPEs) process continuous streams of data and produce results in a real-time fashion, typically through one-at-a-time tuple analysis. In Fog architectures, the limited resources of the edge devices, enabling close-to-the-source scalable analysis, demand for computationally- and energy-efficient SPEs. When looking into the vital SPE processing properties required from applications, determinism, which ensures consistent results independently of the way the analysis is parallelized, has a strong position besides scalability in throughput and low processing latency. SPEs scale in throughput and latency by relying on shared-nothing parallelism, deploying multiple copies of each operator to which tuples are distributed based on its semantics. The coordination of the asynchronous analysis of parallel operators required to enforce determinism is then carried out by additional dedicated sorting operators. To prevent this costly coordination from becoming a bottleneck, we introduce the Viper communication module, which can be integrated in the SPE communication layer and boost the coordination of the parallel threads analyzing the data. Using Apache Storm and data extracted from the Linear Road benchmark and a real-world smart grid system, we show benefits in the throughput, latency and energy efficiency coming from the utilization of the Viper module.

1. Introduction

Data streaming emerged to meet the stringent demands of massive on-line data analysis in various contexts, such as cloud and edge-computing architectures. Stream Processing Engines (SPEs) allow programmers to formulate continuous queries, defined as Directed Acyclic Graphs of interconnected operators, to process incoming data producing results in a continuous fashion; e.g., StreamCloud [1], Apache Storm and Flink [2,3] and Saber [4].

In upcoming IoT-based cyber–physical systems, edge and fog devices can enable close-to-the-source analysis minimizing latency for time-critical applications and adding high cumulative computational power to the resources available in existing data centers. To do so, nevertheless, the limited resources of individual edge and fog devices demand for computationally and energy efficient SPEs.

Parallelism in SPEs is key for achieving *high-throughput* and *low latency* processing for large data volumes in evolving cyber–physical infrastructures [5]. The importance of scaling in throughput while keeping low-latency processing in SPEs is clearly understood, it has also been manifested by work in [6,7,1,8,9]. Pipeline and task parallelism are easily extracted from Directed Acyclic Graphs with operators or tasks assigned to different processing units. However, with data parallelization or fission [10–14], careful orchestration of operators' execution is required to preserve *determinism*, which is required to ensure consistent results independently of the way in which the analysis is parallelized. Data parallelism involves replicating instances of operators, that work concurrently on data streams. An operator's parallel implementation is *deterministic* if, given the same sequences of input tuples, the same sequence of output tuples is produced independently of the tuples' inter-arrival times or the parallelism degree of the operator [15,16,12,13].

Previous attempts to guarantee determinism in SPEs under execution of parallel instances of an operator rely on dedicated merge-sorting operators. These operators are either added to continuous queries by query compilers [1,12,13] or left for developers to place within their streaming applications in SPEs, such as Apache Storm [2]. This type of approach is henceforth referred

☆ Preliminary results have been presented at the International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (AutoDaSP 2017).

* Corresponding author.
E-mail address: ivanw@chalmers.se (I. Walulya).
URL: https://www.chalmers.se/en/staff/Pages/Ivan-Walulya-.aspx (I. Walulya).

https://doi.org/10.1016/j.future.2018.05.067
0167-739X/

to as *operator-layer* determinism in the paper. Minimizing the computational overhead introduced by such dedicated operators is challenging, especially for one-at-a-time, fine-grained low latency tuple processing.

We address the issue of guaranteeing determinism in a modular, automated and efficient way. We start by observing that, commonly in SPEs, each physical stream is piped from a producer (e.g., an incoming link from a sensor, or an outgoing link of an operator instance) to its consumer (another operator instance), without coordination or sharing state. Efficient synchronization over shared memory achieved transparently, is challenging but integral to providing determinism to application developers without requiring the latter to develop custom solutions.

Gulisano et al. [15] proposed *ScaleGate*, a data structure which is customized to guarantee determinism, and which has been used for aggregate and join operators in shared memory systems. In this paper we build upon ScaleGate and provide the following contributions: (i) We modularly shift a procedure of guaranteeing determinism, from the operator-layer to the *communication layer* of an SPE, thus relieving application developers from the burden of devising application-dependent methods. (ii) We design and implement a module, called *Viper*, which can be transparently integrated in an SPE communication layer. Building on ScaleGate, we lift the data-structure's context into the communication layer of an SPE architecture. From ScaleGate to Viper, the novelty is on the transparency provided to the application developer in efficiently guaranteeing determinism. (iii) We integrate Viper in Apache Storm (as a representative example of an SPE) and demonstrate the idea of modularly providing determinism, while caring for efficiency in parallelism, through an experimental evaluation of the proposed methodology. For the evaluation, we chose streaming operators of the Linear Road benchmark [17] and a use-case from a real-world smart grid system as representative examples of where stream processing in fog and edge architectures can be far better than processing in the cloud, as also discussed in [18]. The study clearly shows the throughput, latency and energy-efficiency benefits induced.

In the paper, we present preliminary concepts in Section 2; we describe our proposal for enforcing determinism at the communication layer (rather than the operator layer) of an SPE and discuss the advantages of the former as we introduce the Viper module, in Section 3. We show the use of Viper, as an SPE module, by using Apache Storm as a use case in Section 4.1 and we evaluate the benefits of Viper in Section 5. We discuss related work and conclude in Sections 6 and 7, respectively.

2. System model

This section introduces data streaming, parallel and deterministic execution of *continuous queries* and the performance metrics to assess the results.

2.1. Data streaming

A stream is defined as an unbounded sequence of tuples $t_0, t_1, \ldots$ sharing the same schema $\langle ts, A_1, \ldots, A_n \rangle$. Given a tuple t, $t.ts$ represents its creation timestamp while $A_1, \ldots, A_n$ are application-related attributes.

Continuous queries (henceforth simply queries) are defined as DAGs of *operators* that consume and produce tuples. Operators are distinguished into *stateless* or *stateful*, depending on whether they keep any state that evolves with the tuples being processed. Stateless operators include Map (to alter the schema of tuples) and Filter (to discard or route tuples). Stateful operators include Aggregate (to compute aggregation functions such as sum or average over tuples) and Join (to match tuples coming from multiple streams). Due to the unbounded nature of streams, stateful operations are computed over *sliding windows*, which can be time-based or tuple-based and are defined by parameters *size* and *advance*. Following the data streaming literature (e.g., [19,1,20]), we assume that streams fed by each data source contain timestamp-sorted tuples. If this is not the case, sorting mechanisms such as [21] can be leveraged.

The performance of an operator depends on its *cost* and *selectivity*. The cost represents the average time needed to process an input tuple and (optionally) produce any resulting output tuple. It is thus coupled with the selectivity, which represents the average number of output tuples produced upon the processing of one input tuple (e.g., an operator with selectivity 0.5 will produce, on average, one output tuple each time it processes two input tuples).

To illustrate the aforementioned terms and notions, Fig. 1A presents a sample streaming application from the Linear Road benchmark [17],[1] where position reports are forwarded by vehicles traveling on a highway. The application performs three updates for each incoming report. First, it checks if the report refers to a vehicle entering, leaving or changing segment. It then updates the number of vehicles and the tolls of the involved segments. Finally, it notifies the interested vehicles.

Fig. 1B presents an example centralized query that implements the application's semantics through basic streaming operators. The schema of each stream is presented on top of the operators. A first Aggregate A1 enriches each position report with the previous segment observed for the same vehicle. Subsequently, a Filter F discards reports referring to vehicles that have not changed segment. Aggregate A2 updates the count for each segment and Map M computes the toll for a segment, based on the number of vehicles in it, and notifies vehicles.

2.2. Parallelism, determinism and syntactic transparency

A parallel version of a centralized query (such as the sequential one in the example of Fig. 1B) is desirable because of two reasons: (i) to cope with the large and fluctuating volume of data (in this example the position reports observed in a highway); (ii) to possibly deploy the analysis over a distributed network of nodes, each responsible for e.g. a subset of segments. The latter would avoid centralized data gathering and processing, which indeed might not be feasible because of too high data transmission latency, infrastructure limitations or privacy legislation, among other reasons.

For a parallel query, deterministic execution should ensure that the results given by it are exactly the same that would be given by its centralized counterpart. This is equivalent to the notion of *external determinism* (or *determinacy*), as described in e.g., [22,23]. As shown in [23], processing systems that are described as directed graphs of operations guarantee this property on the global level, if they satisfy determinacy (i) on the operation level and (ii) in the data flows between communicating operations. This implies that if we have a parallel implementation of the query using deterministic processing components and deterministic flow of the results to downstream operators, then the issue is addressed. As argued in [24,1], in the context of query processing in SPEs, for having global determinism it is sufficient to enforce that (i) splitting streams to downstream operator is done according to the semantics of those operators (e.g. for aggregates, tuples with the same group-by key and same timestamp are routed along the same outgoing link); and (ii) when merging streams, attention is paid to order the tuples, so that they provide a single timestamp sorted stream. For this reason, special *m* erge-sorting (M) operators are defined before each operator *instance* fed by a parallel upstream

[1] Section 5 contains a detailed description of the benchmark.