Brief Announcement: Concurrent Data Structures for Efficient Streaming Aggregation

Daniel Cederman
Chalmers University of Technology
cederman@chalmers.se

Vincenzo Gulisano
Chalmers University of Technology
vinmas@chalmers.se

Yiannis Nikolakopoulos
Chalmers University of Technology
ioaniko@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
ptrianta@chalmers.se

Philippas Tsigas
Chalmers University of Technology
tsigas@chalmers.se

ABSTRACT

We briefly describe our study on the problem of streaming multiway aggregation [5], where large data volumes are received from multiple input streams. Multiway aggregation is a fundamental computational component in data stream management systems, requiring low-latency and high throughput solutions. We focus on the problem of designing concurrent data structures enabling for low-latency and high-throughput multiway aggregation; an issue that has been overlooked in the literature. We propose two new concurrent data structures and their lock-free linearizable implementations, supporting both order-sensitive and order-insensitive aggregate functions. Results from an extensive evaluation show significant improvement in the aggregation performance, in terms of both processing throughput and latency over the commonly-used techniques based on queues.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Lists, stacks, and queues; H.2.4 [**Database Management**]: Systems—*Concurrency*; D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords

data streaming; data structures; lock-free synchronization

1. INTRODUCTION

Data streaming [13, 1, 6] emerged as an alternative to store-then-process computing. In data streaming, *continuous queries* (defined as directed acyclic graphs of interconnected operators) are executed by Stream Processing Engines (SPEs) that process incoming data in a real-time fashion, producing results on an on-going basis. As emphasized in [7], the low-latency and high-throughput requirements of the real-time complex processing of increasingly large data volumes makes parallelism a necessity. A good portion of the research has so far focused on leveraging the processing capacity of clusters of nodes and originally centralized SPEs [1] evolved rapidly to distributed [3] and parallel [8, 10] ones.

A data streaming application can be seen as a pipeline where data is continuously produced, processed and consumed. In a parallel environment the underlying data structures should provide the means for organizing the data so that the communication and the work imbalance between the concurrent threads of the computation are minimized while the pipeline parallelism is maximized. Although providing the appropriate data structures that best fit the needs of the application in a concurrent environment is a key research issue [11], providing the data structures that best fit data streaming applications has been overlooked. Existing SPEs such as [3, 8] still rely on basic data structures such as queues; similar is the case with work focusing on the improvement of SPEs' architectures [2] or continuous queries accessing the same stream of data [12].

Contributions. The shared access to the data by the collaborating threads defines new synchronization needs that can be integrated in the functionality provided by the shared data structures. By studying the use and limitations of existing aggregate designs and the data structures they use, we motivate the need for a new approach. We propose concurrent, linearizable and lock-free data structures (*T-Gate* and *W-Hive*) upon which we build enhanced multiway aggregate operators that outperform existing implementations in both order-sensitive and order-insensitive functions. We include indicative results from a study we conducted using two large datasets extracted from the SoundCloud[1] social media and from a smart grid metering network. For both datasets the enhanced aggregation resulted in large improvements, up to one order of magnitude, both in terms of processing throughput and latency. The full study is presented in the technical report [5].

1.1 Problem Description

A stream is defined as an unbounded sequence of tuples $t_0, t_1, \ldots$ sharing the same schema of attributes. Given a tuple t, attribute $t.ts$ represents its creation timestamp at the

[1] `https://soundcloud.com/`

SPAA'14, June 23–25, 2014, Prague, Czech Republic.
ACM 978-1-4503-2821-0/14/06.
http://dx.doi.org/10.1145/2612669.2612701.

data source. Following the data streaming literature (e.g., [3, 8]), we assume that each stream contains timestamp-sorted tuples. In the presence of multiple streams, tuples from different streams may arrive out of timestamp-order, posing a need to merge and synchronize them before processing. Data streaming *continuous queries* are defined as directed acyclic graphs. Nodes represent operators that consume and produce tuples, while edges specify how tuples flow among operators. Operators can be divided into *stateless* or *stateful*, depending on whether they keep any state while processing tuples. Due to the unbounded nature of streams, stateful operations are computed over a *sliding window*, defined by parameters *size* and *advance* (e.g., to group tuples received during periods of 5 minutes every 2 minutes, or the last 10 received tuples every 3 incoming tuples).

The multiway aggregate operator is defined by its window's *size* and *advance* parameters, by a function F applied to the tuples and by an optional *group-by* parameter K (a subset of the input tuple's attributes), which specifies if F is applied independently on tuples with different K value. We focus on deterministic functions, which can be *order-sensitive* (e.g., forward only the first received tuple) or *order-insensitive* (e.g., count the number of tuples) with respect to the processing order of the tuples that contribute to the same window. If the group-by parameter K is defined, the operator needs to keep separate windows not only for different time intervals, but also for different values of K. We define a *winset* as the set of windows covering the same time interval for different values of K.

In scenarios such as parallel-distributed SPEs [8, 4] and replica-based fault tolerant SPEs [3], it is desirable to provide deterministic processing of input tuples (i.e., to produce the same sequence of output tuples given the same sequences of input tuples). When dealing with multiple input streams, processing is not deterministic if tuples are simply processed in the order they are received (i.e., if the processing order depends on the input streams' inter-arrival times). To ensure deterministic processing, tuples from multiple input streams need to be *merged* into one sequence and *sorted* in timestamp order [8], an operation we refer to as *S-Merge*. A tuple is *ready* to be processed if at least one tuple with an equal or higher timestamp has been received at each input stream.

We consider systems of concurrent threads. Communication and information exchange relies on shared data and concurrent shared data structures provide common means for that. Concurrent shared data structures can be implemented in a *lock-free* way, i.e. guaranteeing that at least one of the threads operating on it is guaranteed to finish its operation in a bounded number of its own steps. The correctness of such implementations is commonly shown through *linearizability* [9], which guarantees that, given a history of concurrent operations, there exists a sequential ordering of them, consistent with their real-time ordering and with the sequential semantics of the data structure.

2. AGGREGATION'S PARALLELISM AND THE ROLE OF DATA STRUCTURES

Widely used SPEs such as Borealis [3] or StreamCloud [8] perform multiway aggregation by relying on per-input queues to store incoming tuples. Distinct threads insert and remove tuples from such queues and concurrent accesses are synchronized with the help of locks. Figure 1 presents this

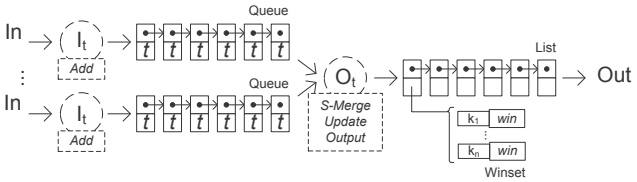

Figure 1: Baseline Multi-Queue based aggregate design.

design, which we refer to as Multi-Queue (MQ). The output thread O_t peeks the first tuple in each queue to determine which one is *ready* to be processed (input threads and output threads are denoted by I_t and O_t respectively). The same thread is also responsible for *updating* the windows a tuple contributes to, as well as producing and forwarding the *output* tuples when they are ready. Since O_t is the only thread in charge of updating windows, no synchronization mechanism is required to access the *winsets*, usually implemented as hash tables to easily support arbitrary numbers of windows and to locate them quickly given the tuple's group-by parameter K.

Parallelization challenges. In existing implementations, *S-Merge* usually relies on simple sorting techniques, whose cost is linear to the number of inputs. Examples include the *Input Merger* operator [8] or the *SUnion* operator [3]. For this reason, the first challenge relies on the parallelization of the *S-Merge* operation. To this end, extra inter-thread synchronization is needed to ensure *deterministic* processing. Another challenge is on the parallelization of the *Update* stage. Again, to guarantee deterministic processing, the result of a window should be outputted only after all its contributing tuples have been processed. For order-sensitive functions, *Update* cannot be invoked in parallel on tuples sharing the same K value or when no group-by parameter is defined. This restriction can be relaxed for order-insensitive functions, since the result of a window would not be affected by the order in which concurrent threads update it. In both cases, parallelism can be enhanced by a concurrent data structure that coordinates the access to the windows.

Utilizing concurrent data structures. We are looking for concurrent data structures that are capable of sorting input tuples at insertion time. In principle, tree-like data structures could provide concurrent logarithmic-time insertion operations. The need for extracting such tuples in timestamp order though, made us expect that a lock-free concurrent skip list [14] would be the right candidate due to its nodes' structure. Nevertheless, a skip list would not differentiate between tuples that are *ready* and tuples that are not. Because of that, checking whether a tuple is *ready* or not would still be penalized by a cost that is linear to the number of inputs as for the multi-queue implementations. Furthermore, it would provide unnecessary functionality (i.e., a more complex implementation) such as deletion of elements at arbitrary positions (only head elements need to be removed in our scenario). Similar considerations hold for a lock-free concurrent skip list that could potentially be used to maintain the operator's *winsets*.

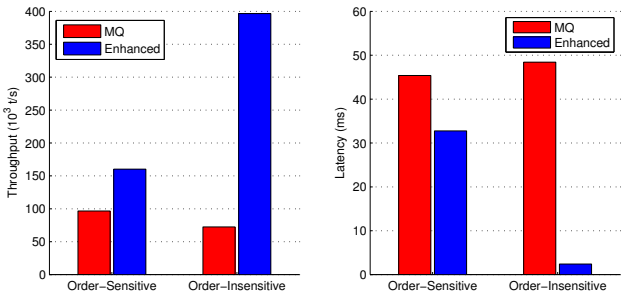

Figure 2: Throughput and latency for a fixed input rate.

New data structures and aggregate designs. We propose two concurrent, linearizable and lock-free data structures, T-Gate for managing tuples and W-Hive for managing *winsets*, which we use to build our enhanced multiway aggregate operators. T-Gate is used in a first step to efficiently store the incoming data in a concurrent manner and to offer proper synchronization on when the data will be accessed in order to provide a deterministic aggregate operator. In order to further parallelize the computation of the aggregate values with the use of multiple threads that produce output data, W-Hive takes care of synchronizing the output of the data according to the streaming model requirements (e.g. sorted). In another design we use only the W-Hive to focus on order insensitive aggregate functions (i.e. the processing order of tuples within a window does not affect the result - e.g.average - but still the order of characterizing tuples ready or inserting tuples in the windows is important for the determinism). Figure 2 shows how the respective enhanced aggregate designs perform against the baseline MQ implementation, for both the cases of order sensitive and insensitive functions, in terms of throughput and latency, for a fixed input rate of equal numbers of incoming streams.

3. CONCLUSIONS

We give an overview on how the parallelism of streaming multiway aggregation can be enhanced by leveraging application-tailored concurrent data structures. We propose new data structures for managing tuples and windows. Their operations and their lock-free implementations enable better interleaving and hence improve the balancing and the parallelism of the aggregate operator's processing stages. As shown in an extensive evaluation based on real-world datasets [5], our enhanced aggregate implementations outperform existing ones both in terms of throughput and latency, and are able to handle heavier streams, increasing the processing capacity up to one order of magnitude.

4. ACKNOWLEDGMENTS

The research leading to these results has been partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) through the EXCESS Project (www.excess-project.eu) under grant agreement 611183, through the SysSec Project, under grant agreement 257007, through the FP7-SEC-285477-CRISALIS project, by the collaboration framework of Chalmers Energy Area of Advance and by the Chalmers Center for E-science.

5. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The International Journal on Very Large Data Bases*, 2003.

[2] S. Akram, M. Marazakis, and A. Bilas. Understanding and improving the cost of scaling distributed event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, 2012.

[3] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 2008.

[4] C. Balkesen, N. Tatbul, and M. T. Özsu. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, DEBS '13, pages 15–26. ACM, 2013.

[5] D. Cederman, V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Concurrent data structures for efficient streaming aggregation. Report, Chalmers University of Technology, 2013.

[6] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[7] B. Gedik, R. R. Bordawekar, and S. Y. Philip. Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 2009.

[8] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 1990.

[10] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann. Stormy: an elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, 2012.

[11] M. M. Michael. The balancing act of choosing nonblocking features. *Commun. ACM*, 2013.

[12] A. L. Shenoda Guirguis, Panos K. Chrysanthis and M. A. Sharaf. Three-level processing of multiple aggregate continuous queries. *Proc. of the 28th IEEE International Conference on Data Engineering*, 2012.

[13] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 2005.

[14] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 2005.