How to become a (Throughput) Billionaire: The Stream Processing Engine PipeFabric

Constantin Pohl
TU Ilmenau, Germany
constantin.pohl@tu-ilmenau.de

ABSTRACT

The ability to process data in real time has gained more and more importance in the last years through the rise of IoT and Industry 4.0. Stream processing engines were developed to handle huge amounts of data with high throughput under tight latency constrains. Trends in modern hardware have led to further specializations to efficiently utilize their chances and opportunities, like parallelization to multiple cores, vectorization, or awareness of NUMA.

In this paper, we present the stream processing engine PipeFabric, which is under ongoing development at our department. We will describe internal concepts and stream semantics along with decisions taken in the design space. In addition, we will show challenges posed by modern hardware that we are considering to improve performance and usability of our engine. Finally, we underline the potential of PipeFabric by running parallelized queries on a single Xeon Phi processor, resulting in about 1.3 billion tuples processed per second.

Keywords

Stream Processing, SPE, PipeFabric, Xeon Phi

1. INTRODUCTION

Various applications require processing and analysis of data continously with short response times. To point an example, smart manufacturing machines of Industry 4.0 use sensors to stream their status information, allowing to detect and correct anomalies in their behavior as fast as possible.

In the early 2000s, the first stream processing engines (often referred to as SPEs) were published, clearly outperforming relational DBMS for this task [1]. The *one-tuple-at-a-time* concept (also known as *Volcano style* from Graefe [2]) for data streaming allowed SPEs to keep individual tuple latencies low. To increase throughput in terms of tuples processed per second, micro-batching strategies as well as data parallelization by partitioning were applied and refined over time.

Recent work focuses on exploitation of modern hardware, since memory as well as processors tend to become more and more specialized to better solve different requirements of applications. GPUs, Multi- and Manycore CPUs, FPGAs, or even Vector Engines on processor side, HBM, NVM, HDD, or SSD on memory side show massively different behavior under different tasks and come with various configurations and challenges to utilize them efficiently. To combine high throughput as well as low latency data processing with opportunities given by modern hardware, we introduce our SPE *PipeFabric*[1] in this paper, along with its concepts and design decisions.

2. RELATED WORK

There are many SPEs published in the past, some being frameworks developed by research groups, others being commercially used engines in industry. In this section, we give a short overview of selected SPEs, classified into the *scale-up* (single node) and *scale-out* (distributed) principle.

Aurora [3] was one of the first general purpose SPEs that specialized on answering queries on data streams in real-time. Queries are described by directed graphs, connecting different operators together and thus forming the data flow. Since Aurora was designed to run on a single node, the **Borealis** [4] SPE added fault-tolerance and consistency to run in a distributed setting.

Other recent distributed SPEs are **Apache Flink** [5] (forked from the Stratosphere engine), **Apache Storm** [6], and **Apache Spark Streaming** [7]. All of them can be commonly found in various companies, having a large user base. Their main goal in addition to low latency and high throughput is scalability, along with fault tolerance within a distributed setting. Processing Big Data under real-time constraints requires the distribution of computation to multiple machines in a cluster eventually, since scale-up is limited.

Nevertheless, scale-up solutions can also come very far for a fraction of monetary cost of a distributed solution. **SABER** [8] and **StreamBox** [9] are two SPEs that are optimized to run on a single node. While SABER can be executed on heterogeneous processing units like GPUs, StreamBox can run on Manycore CPUs supporting out of order tuple processing.

Finally, our SPE **PipeFabric** can be classified into the field of scale-up SPEs, focused on efficient execution of queries on Multicore and Manycore CPUs.

31st GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 11.06.2019 - 14.06.2019, Saarburg, Germany.

[1]Open Source, `https://github.com/dbis-ilm/pipefabric`

3. STREAM PROCESSING PARADIGM

With the goal of low latency in mind, the general streaming workflow follows the one-tuple-at-a-time strategy. However, if latency requirements can be relaxed, gathering tuples together into batches for less communication efforts (like function calls) and vectorized processing can greatly increase throughput.

PipeFabric provides a query structure called *Topology* (like Apache Flink), which contains one or more streaming sources, operators applied on them, and optional stream sinks. Topologies can be conceptually seen as directed acyclic graphs routing tuples through different operators (see Figure 1).

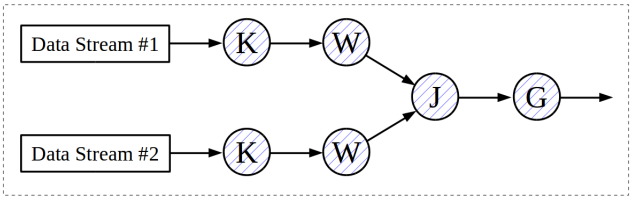

Figure 1: Example of a Stream Query

This query example consist of two input data streams, two operators applied on each of them (specifying the key attribute and window semantics), combined together by a join with a final grouping on a certain attribute.

Operators (called *Pipes*) are connected via channels, following the *publish/subscribe* pattern. They can connect to any operator upstream by subscribing, publishing their own results to other operators downstream. To reduce overhead, only tuple pointers are passed between them. The following subsections briefly describe the different sources, operators, and sinks in our SPE.

3.1 Stream Sources

Stream sources produce tuples for individual queries, thus being the necessary query starting points. The main sources of streaming are tuples provided via different network protocols, files (or tables), other streams, or specialized sources.

Network protocols. The most common source for tuples are servers or sensors that deliver data being processed continuously. PipeFabric can connect via REST API, RabbitMQ, Apache Kafka, MQTT, and ZeroMQ. Protocol logic for the connection is internally realized within the parametrizable source operators, hidden from the user.

Files/Tables. Data streams can also subscribe to different files like CSVs or binaries, as well as relational tables. Therefore, a query can also run different benchmarks provided as files on the file system. A special use case are tables e.g. from RocksDB, allowing also to use transactional semantics on operations under ACID guarantees.

Streams. Another source is the subscription on already defined PipeFabric streams. This allows queries to send their results conceptually as a new data stream on which other queries can subscribe to.

Specialized sources. In addition, PipeFabric provides various specialized source operators for different use cases. To run the Linear Road benchmark [1], a synchronized source is provided, publishing tuples from a file in real-time according to its timestamp. Another specialized source is the data generator, which will continuously generate tuples according to a format specified by the user. One last example is the matrix source, sending lines, columns, or even full matrices represented as tuples.

3.2 Operators

PipeFabric supports various operator types being applied on incoming tuples. The most common single source operators are the projection of attributes, applied predicates (selections), aggregations, or groupings. Each of them has its own operator which can also be configured, e.g. to choose an aggregation type (count, sum, etc.). To join multiple sources, PipeFabric uses the non-blocking symmetric hash join in addition to the recently published ScaleJoin algorithm [10]. With a customizable operator called notify, it is also possible to apply any UDFs on incoming tuples via lambda functions.

3.3 Stream Sinks

Sinks are operators which logically terminate a stream or query. It is possible to write query results on the fly into files, tables, or a new data stream. Results can also be returned as a general output, e.g. for visualization in a GUI. However, PipeFabric currently does not have an own visualization tool like e.g. Aurora has.

4. STREAMING CONCEPTS

In this section, we further describe streaming concepts of SPEs which are also realized in PipeFabric.

Partitioning/Merging. To utilize intra-query parallelism, it is possible to create multiple instances of the same operator, splitting tuples with a partitioning function and merging results of partitions afterwards. This concept is shown in Figure 2.

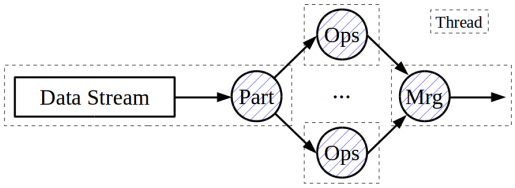

Figure 2: Partitioning and Merge Schema

Each partition is run by a separate thread, exchanging tuples with a synchronized queue. This allows the utilization of all cores on a Multicore or even Manycore CPU, increasing throughput massively if the computational requirements within a partition are high enough to justify synchronization efforts.

Batching/Unbatching. As previously mentioned, batching tuples together reduces communication efforts (especially between threads) and enables vectorized execution of operations. In PipeFabric, a batch as well as unbatch operator is provided. The batch operator stores incoming tuple pointers

internally until a given batch size is reached, forwarding the batch at once by creating and passing a batch pointer to the next operator. The unbatch operator does the opposite, extracting tuple pointers from a batch and forwarding them again one after another.

Window. A window operator tracks incoming tuples for marking them as outdated after a while. Outdated tuples do not participate in stateful operations like aggregates or joins, being removed from those states for further calculations. Long running queries can therefore discard tuples after a while, keeping the memory footprint low and also manageable. Most common window algorithms are the tumbling and sliding window. The former drops all tuples at once when its size is reached while the latter slowly fades out tuples individually. Figure 3 visualizes the concept for the sliding window calculating an aggregate.

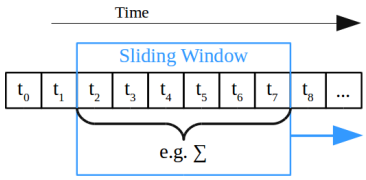

Figure 3: Sliding Window Semantics

Both window types can discard tuples based on time or tuple count. PipeFabric uses a list data structure for the window state internally to allow efficient appending and removing at both ends of the list.

Transaction Support. Transactions are a common concept in relational database systems. They wrap operations on tables together, providing ACID guarantees to always ensure consistency of the database. Since PipeFabric can also stream data from or to tables, it contains basic transaction support for recovery and consistency [11]. To point an example, a query writing to a table executes the changes through running transactions, while multiple queries reading and writing need isolation to guarantee correctness additionally.

5. MODERN HARDWARE CHALLENGES

In this section, we give an overview of our ongoing work with PipeFabric regarding modern hardware, mainly focusing on Manycore CPU utilization and support for upcoming NVM technology. Manycore CPUs provide high core numbers and thus high thread counts, resulting in challenges in terms of synchronization and thread contention. It is also important to notice that intra-query parallelism through multithreading usually leads to tuples arriving out of order after partitioning which can be a problem for queries detecting patterns over time.

On the memory layer, NVM and also high-bandwidth memory (HBM) pose new challenges for stream processing. NVM offers persistence with latencies comparable to main memory (with a read/write asymmetry), which is interesting to explore especially for transactional operations on tables. HBM on the other hand offers great performance for applications being memory bound at the cost of small capacity, leading to optimization problems where to use it for the greatest performance benefit.

5.1 Adaptive Partitioning

Data stream behavior can change during runtime. This means that the amount of tuples arriving per second can change, leading to different amounts of partitions that would be ideal to solve that moment of the query. If the degree of partitioning is too high, computing resources are wasted, which is also a common problem for cloud providers. On the other hand, if the workload is underestimated, too less partitions cannot catch up with tuple arrival rates leading to wrong results because of tuples being discarded due to full buffers.

Dynamic partitioning approaches address this problem by using a partitioning function that can be changed over time. This allows to counter skewed streams by dynamically changing the tuple routes to underutilized partitions. Nevertheless, one step further is the adaptive partitioning strategy where not only the function is variable but also the number of partitions can change. Figure 4 shows recent work for an adaptive partitioning strategy within PipeFabric, where the y-axis describes the number of tuples arriving per second. The partitions are directly converted into throughput to allow a better comparison.

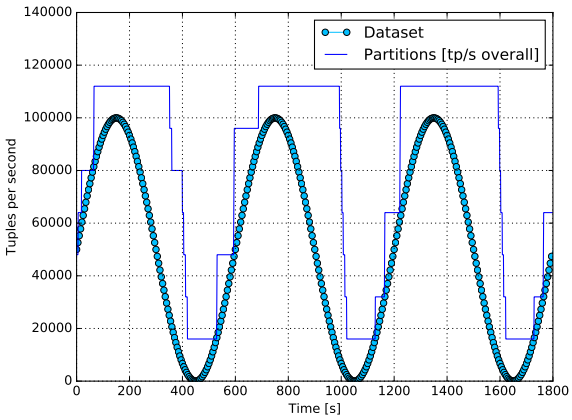

Figure 4: Adaptive Partitioning Behavior

Such an approach, scaling the number of partitions up and down, raises mainly two problems:

- **State migration** when a partition is removed.
- **Decision for scaling**, i.e. when to add or to remove a partition.

The migration problem can be solved by stopping the query, performing the state migration, and resuming. However, this can break latency constraints since during a stop the processing cannot continue. A better proposed solution is to create a parallel state which gets duplicates of new tuples until both states are equal. Then, the original state can be dropped safely.

PipeFabric currently uses a static partitioning concept where the number of partitions as well as the partitioning function does not change. At the moment we are investigating possibilities and options to apply an adaptive approach to fully utilize a Manycore CPU under skewed data stream behavior.

5.2 Order-Preserved Merging

As mentioned in the last section, partitioning can lead to out of order tuples afterwards, e.g. when a predicate within a partition drops more or less tuples or a hash table for joining tuples has a chain of cache misses on probing. Ordering the output *without blocking results* can be difficult.

A solution for this problem is to store incoming tuples in different queues per partition. Then, the merge operation can check and compare the first elements in all of the queues, forwarding the oldest tuple among them (see Figure 5 for the general idea).

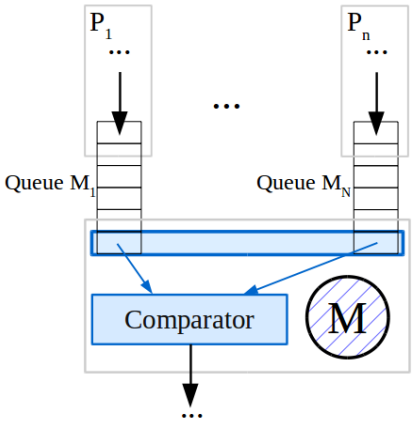

Figure 5: Order-Preserved Merging

For the special case that one partition is not producing any outputs, it is possible to add a dummy element after a certain time to guarantee ordered execution only within a certain time frame. This strategy is also named as *k-slack* algorithm [12].

Regarding PipeFabric, we would like to combine an order-preserved merge with an adaptive partitioning approach. This means that on a change on the partition number the sorted merge operation has additional overhead on synchronization with the adaptive partitioner, since it has to know which partition will be removed soon.

5.3 Real-time Query Modification

Even if not directly hardware-related, it would be a quality of life feature to be able to change the query (which possibly runs for weeks or months) without restarting it [13]. Over a longer period, the query states can become huge, especially when the query has a lot of operators, not even to mention the time lost when the state is migrated into a new query. To add real-time query modification, we would like to address the following use cases:

- Add or remove a new operator within the query dataflow.
- Change the function (UDF) of a single operator.

For our SPE PipeFabric, these features need an additional controller thread that can be invoked by the user to trigger a query modification. To add or remove a new operator, the previous as well as next operator within the query need a notification to not exchange tuples anymore. After that notification, the new operator must be created and connected to others with publish/subscribe channels. When the connection is finished, new notifications must be sent to again allow tuple exchange.

When the UDF of an operator is changed, only this operator is involved in modification. This means that the function cannot be executed while it is changing, leading to notifications being necessary again.

For both modifications, tuples have to be buffered while the query is changed. Ideally, the modification is done during a delay in tuple arrival of the data stream, else a short blocking is inevitable.

5.4 HBM Allocation on States

One of our previous works [14] investigated HBM impact on different query states. We concluded that for operations with small states like aggregates it is not useful at all, while windows can benefit a little from more bandwidth. Stream sources on the other hand greatly benefit from HBM.

In a followup work, we will integrate HBM detection within our SPE along with the provision of custom HBM state allocators. In addition to that, we would like to add a cost model for HBM to allow a query optimizer to choose between the different memory types. Finally, since the symmetric hash join only marginally improves with more bandwidth (being mostly latency bound), we are investigating different stream join algorithms to improve bandwidth utilization especially on a Xeon Phi processor.

5.5 Lockfree Data Structures

With Manycore CPUs, the degree of contention on shared data structures can nullify any advantage of parallelization. The usage of fine grained locks or latches along with optimistic concurrency protocols can improve scalability a lot. Recent work on PipeFabric investigated lockfree data structures for states that are accessed by multiple threads concurrently.

Queues between threads are a prominent example, where specialized lockfree queues (the so-called *Single Producer Single Consumer* (SPSC) queues) realized as ring buffers greatly enhance performance. Hash tables for joins are another common structure to benefit from the lockfree paradigm, which is not only restricted to stream processing but also for joins on relational tables.

Lockfree programming usually has a huge disadvantage when it comes to debugging or guaranteeing thread safety. However, there are high level abstractions in libraries like Boost[2] or Intel TBB[3] which hide lockfree operations behind a user-friendly interface. Currently PipeFabric still uses locks and latches, but to improve throughput, we plan to add lockfree pendants in the near future.

5.6 Multiway-Stream Join

The symmetric hash join within PipeFabric is a binary hash join, which means that it can only join two connected input streams. To join a higher number of stream sources, the current solution leads to a binary tree of symmetric hash join operators with the following problems:

- Individual tuple latency can become extremely bad if it has to be repeatedly joined from bottom up within the tree.

[2] `https://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html`
[3] `https://software.intel.com/en-us/node/506169`

- Since intermediate join results are fully materialized in each join operator, the memory footprint can become very large for intermediate hash tables inevitably.

Multiway join operators that can connect to any number of streams on the other hand look very promising, since a single join instance has a lot of opportunities to optimize tuple storage and probe sequences. Figure 6 shows the concept of a binary join tree compared to a multiway join.

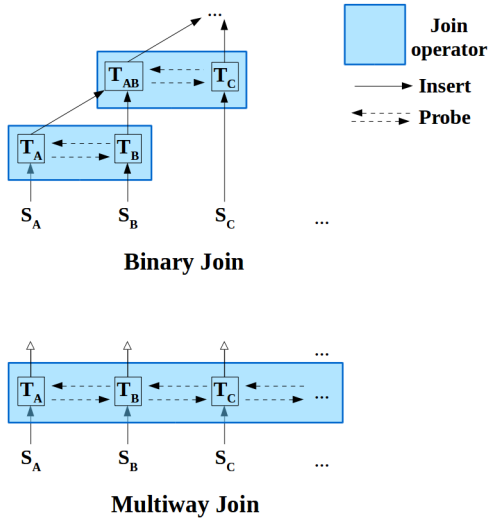

Figure 6: Binary vs. Multiway Join

To efficiently join many concurrent data streams, the join operator has to minimize probe misses to reduce contention (e.g. by only probing when matches can be found in all hash tables). In addition to that, different parallelization strategies (like data parallelism or fully sharing states) are possible which we like to investigate in future work.

6. EXPERIMENTAL EVALUATION

With this section, we will prove the statement in the title of this paper experimentally. First, we will list our experimental setup, followed by the results and discussion afterwards.

6.1 Setup

On the hardware side, we used a Xeon Phi Knights Landing Manycore CPU (KNL 7210) with 64 cores à 1.3GHz, supporting up to 4 threads each due to hyperthreading. It runs in SNC4 mode, which means that the cores are distributed into four distinct regions classified as NUMA nodes. Along with the CPU comes 16GB HBM on chip, the so-called Multi-Channel DRAM (MCDRAM). This MCDRAM provides over 420GB/s memory bandwidth and is configured in Flat mode, therefore it can be manually addressed via Numactl[4] or Memkind API[5], else it is not used at all.

We built our SPE PipeFabric with the Intel compiler version 17.0.6. The operating system is CentOS version 7 running Linux kernel 3.10. The most important compilation flags are code optimization with -O3 and -xMIC_AVX512 for auto-vectorization with AVX512 instruction set.

[4] https://www.systutorials.com/docs/linux/man/8-numactl/
[5] http://memkind.github.io/memkind/

6.2 How to become a Billionaire

A query that is able to process a billion tuples per second needs some tuning along with simplifications, obviously. First, all input streams are fully allocated within the MCDRAM first, there is no regular DDR4 RAM involved, not to speak of disks like SSDs. The stream query only applies a selection predicate on each input tuple, forwarding those tuples that satisfy the predicate to an empty UDF operator. More computations lead to more work for the threads, reducing overall throughput. The different predicates as well as their measured impact on performance are summarized in Table 1.

Predicate	Selectivity	tp/s (256 threads)
true	100%	720M
key mod(2)	50%	937M
key mod(10)	10%	1.16B
false	0%	1.39B

Table 1: Selection Operator

Next, the query is realized as inter-query parallelism which means that all 256 threads of the KNL run a local query version subscribed to an replicated input stream without contention between them, to overcome the low clock frequency. We ran the query with a different degree of inter-query parallelism. The selection predicate is *false*, however, the query would also allow more than a billion tuples per second with 10% selectivity, as shown in Table 1. The results can be found in Figure 7.

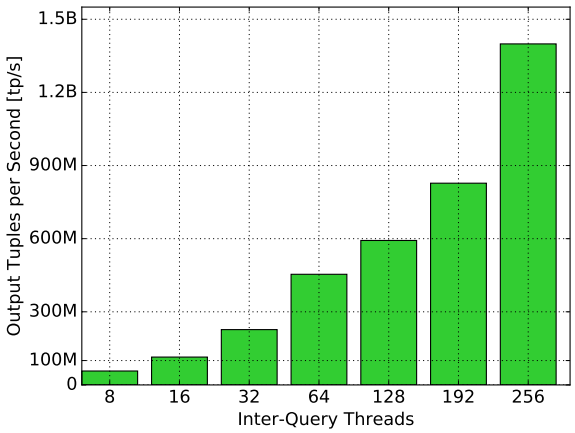

Figure 7: Stream Query Results

The scaling of throughput is close to ideal, where doubling the number of threads doubles the overall throughput. However, including concurrent actions like synchronized access to data structures reduces the scalability the more threads are added. When using DDR4 memory that has only around 80GB/s bandwidth, the highest throughput is reached at 64 threads with approximately 200 million tuples per second (not shown in the plot). More than 64 threads degrade performance on DDR4, since the threads exceed the bandwidth and thus are idling while the memory controllers finish their requests.

7. CONCLUSION

In this paper, we presented *PipeFabric*, a SPE developed at our department with focus on scale-up performance. First, we gave an overview of other well-known SPEs, classified by their decision on scaling up or scaling out. After that, we briefly described stream processing characteristics on the base of PipeFabric. In addition to the basic concepts, we extended the section by discussing various streaming paradigms to better utilize given hardware, like partitioning of the data flow or batching tuples.

Then, we came to our current research heavily influenced by modern hardware. We explained the challenges posed mainly by Manycore CPUs as well as HBM, followed by our recommendations and ideas to improve our SPE. Adaptive partitioning will allow queries to scale with data stream behavior, which is even more important on a Manycore CPU that can provide hundreds of partitions easily. In combination with an order-preserved merge step, results from the partitioning can be reordered again, allowing further analysis downstream (like pattern matching). With long-running queries, we plan to add query modifications in real time, where operators can be added as well as removed without restarting the query as well as online changeable UDFs. To better utilize HBM, we will add allocators accordingly, leading to a cost model for an optimizer being able to decide on which memory type states should be placed. To further improve throughput under high contention, lockfree pendants to our used data structures will be added. And finally, since a binary join tree badly utilizes bandwidth and hurts individual latency, we plan to investigate multiway stream joins in the future.

After the discussion on modern hardware challenges, we described our experiments on which we were able to create a stream query written in PipeFabric, running on the Xeon Phi processor, leading to more than a billion tuples processed per second finally. Although the query is more a synthetical one, it underlines the potential of our SPE, nevertheless.

8. REFERENCES

[1] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB Proceedings, Toronto, Canada, August 31 - September 3 2004*, pages 480–491, 2004.

[2] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[3] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A Data Stream Management System. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 666, 2003.

[4] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 13–24, 2005.

[5] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *PVLDB*, 10(12):1718–1729, 2017.

[6] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm @Twitter. In *SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156, 2014.

[7] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012*, 2012.

[8] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 555–569, 2016.

[9] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. StreamBox: Modern Stream Processing on a Multicore Machine. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 617–629, 2017.

[10] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 144–153, 2015.

[11] Philipp Goetze and Kai-Uwe Sattler. Snapshot Isolation for Transactional Stream Processing. In *Proceedings of the 22th International Conference on Extending Database Technology, EDBT 2019*. OpenProceedings.org, March 2019.

[12] Christopher Mutschler and Michael Philippsen. Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 1133–1144, 2013.

[13] Adrian Bartnik, Bonaventura Del Monte, Tilmann Rabl, and Volker Markl. On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines. In *BTW Proceedings, 4.-8. March 2019, Rostock, Germany*, pages 127–146, 2019.

[14] Constantin Pohl. Stream Processing on High-Bandwidth Memory. In *Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, Wuppertal, Germany, May 22-25, 2018.*, pages 41–46, 2018.