High-Performance Stateful Stream Processing on Solid-State Drives

Gyewon Lee
Seoul National University

Jeongyoon Eo
Seoul National University

Jangho Seo
Seoul National University

Taegeon Um
Seoul National University

Byung-Gon Chun
Seoul National University

ABSTRACT

Stream processing has been widely used in big data analytics because it provides real-time information on continuously incoming data streams with low latency. As the volume of data increases and the processing logic becomes more complicated, the size of internal states in stream processing applications also increases. To deal with large states efficiently, modern stream processing systems support storing internal states on solid state drives (SSDs) by utilizing persistent key-value (KV) stores optimized for SSDs. For example, Apache Flink and Apache Samza store internal states on RocksDB. However, delegating state management to persistent KV stores degrades the performance, because the KV stores cannot optimize their state management strategies according to stream query semantics as they are not aware of the query semantics. In this paper, we investigate the performance limitations of current state management approaches on SSDs and show that query-aware optimizations can significantly improve the performance of stateful query processing on SSDs. Based on our observation, we propose a new stream processing system design with static and runtime query-aware optimizations. We also discuss additional research directions on integrating emerging storage technologies with stateful stream processing.

ACM Reference Format:
Gyewon Lee, Jeongyoon Eo, Jangho Seo, Taegeon Um, and Byung-Gon Chun. 2018. High-Performance Stateful Stream Processing on Solid-State Drives. In *9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3265723.3265739

APSys'18, August 2018, Jeju, South Korea

ACM ISBN 978-1-4503-6006-7/18/08…$15.00
https://doi.org/10.1145/3265723.3265739

1 INTRODUCTION

Stateful stream processing enables complex data analytics in real time, and thus has been widely adopted. For example, stream queries with window operators provide real-time statistics on fast-incoming data within given periods (e.g., the number of total clicks for each item in an e-commerce store during the last 24 hours). As the volume of data to be processed becomes larger and stream processing applications become more complicated, the size of states grows to terabyte scale [5]. As a result, it is necessary to handle huge internal states that do not fit into the physical memory.

To address this problem, modern stream processing systems such as Apache Flink [10] and Apache Samza [18] support storing internal states in secondary storages, which offer more capacity with lower cost compared to DRAMs. Among the secondary storages, solid-state drives (SSDs) are widely adopted because they provide good random access performance with affordable price. The stream processing systems utilize persistent key-value (KV) stores optimized for SSDs (e.g., RocksDB [4]) and delegate the state management to the stores.

However, because external KV stores are not aware of stream query semantics, they often provide sub-optimal state management strategies that degrade the overall performance of stream processing applications. For example, the write buffer of RocksDB called memtable organizes its data in sorted order by default. Using this approach, RocksDB provides reasonable performance for various KV-store operations including range read/write and iteration. Through our preliminary evaluation, however, we demonstrated that this approach incurs additional CPU overhead when dealing with stream queries with frequent point updates, and thus results in poor performance of the entire stream pipeline (Section 3).

To mitigate the performance problem of the current stateful stream processing on SSDs, we believe that it is necessary to develop a new stream processing system that optimizes the state management strategies on SSDs according to the state-access patterns derived from diverse stream query semantics.

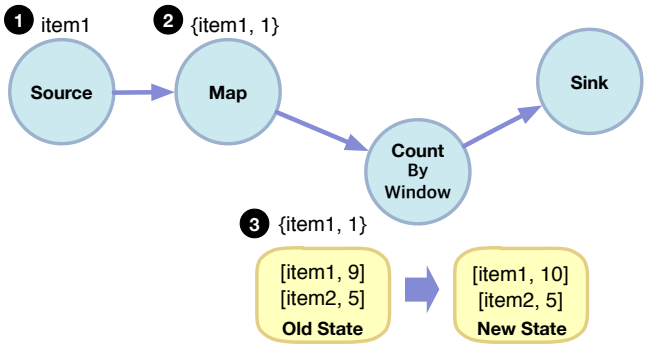

Figure 1: Stateful stream processing in an e-commerce click log example. ❶ Source accepts each item as key. ❷ Map operator transforms an event to a $(item, clicks)$ tuple. ❸ CountByWindow operator aggregates the tuples and update the number of clicks for each item within a window.

We implement a proof-of-concept optimization on Apache Flink for the query workload with frequent point updates and show that the performance of stateful stream processing on SSDs can significantly be improved by optimizing the state management strategies according to the state-access patterns. We then propose a design for the new stream processing system that automatically optimizes state management strategies on SSDs by leveraging both static and runtime analysis. We also discuss research directions on applying emerging storage technologies such as non-volatile memory, remote direct memory access, and near-data processing to accelerate stateful stream processing on SSDs.

2 BACKGROUND

In this section, we briefly review the concept of stateful stream processing and the current design of stateful stream processing on SSDs using persistent KV stores. We focus on RocksDB because RocksDB is widely adopted in popular stream processing systems including Flink and Samza.

2.1 Stateful Stream Processing

A stream processing query is represented as a dataflow graph, whose vertex acts as a source, operator, or sink. Figure 1 illustrates a stateful stream query that counts the number of clicks for each item in a window. When the source vertices receive data events, the events are fed to operator vertices in real time where they are transformed according to the operator logics. Similar to other big data analytics frameworks, stream processing systems can work in a distributed fashion to ingest a large volume of data streams. When deployed on a cluster of machines, the operators in a stream query graph are partitioned and distributed to multiple machines.

Stream operators often maintain internal states to support complex stream processing applications. In Figure 1, the input data streams are transformed to $(item, clicks)$ tuples and aggregated by keys to count the number of clicks per item. Here, the "CountByWindow" operator is stateful in that it stores the aggregated click counts for each item within a given window.

The types of states and how they are accessed vary according to the semantics of stream operators. We explain a few commonly used stateful stream operator examples and discuss how they differ in accessing their states.

Associative window aggregation. Window operator enables real-time analytics on a recent set of data. Associative aggregation operators are used for computing simple statistics within a window. Counting the number of clicks for each item in an e-commerce store inside a window is an example of such aggregations (Figure 1). If an aggregation function is associative, it is possible to build the aggregated result for the entire window by merging multiple partially aggregated values. Leveraging this property, the operator maintains only the partially aggregated values that are consistently updated upon each data event arrival.

Non-associative window aggregation. Aggregation functions are often non-associative such as calculating median value or extracting top-k items within a given window. In a window operator with non-associative aggregation, all the data events inside the window should be maintained to produce the aggregated result. The states are managed as a list of recent data events and incoming data events are continuously appended to the list upon their arrivals.

Operators with historical data. Stream operators sometimes need to access historical data. For example, a user may want to join the current data streams with the data collected a month ago to analyze the trend changes over time. Such stream operators that leverage historical data lead to frequent range reads because they need to access the data for a specified time range.

Traditionally, the internal operator states are managed in main memory. However, as the volume of the data stream increases and the stream query logic becomes more complicated, it becomes necessary to handle states whose size is bigger than that of the main memory. In this case, the states need to be stored on persistent storages. For the purpose, SSDs are widely adopted because they have higher random access performance compared to traditional hard-disk drives (HDDs).

2.2 State Management via a KV Store

To handle stateful stream processing queries whose state does not fit into the main memory, recent stream processing systems such as Flink [10] and Samza [18] provide stateful

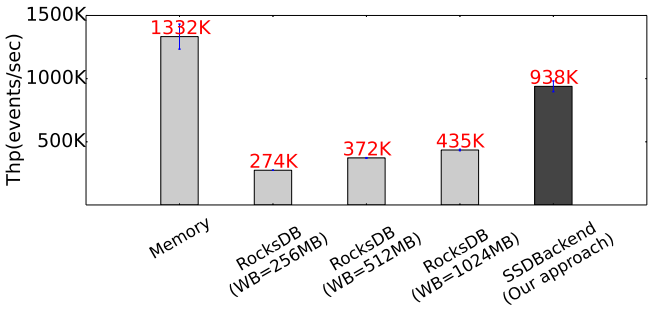

Figure 2: The performance comparison for ReadWrite with various state backends. WB refers to the size of write buffers in RocksDB.

stream processing on SSDs by delegating the state management to a persistent KV store. RocksDB [4] is widely used for stateful stream processing because it offers high random read and write throughput on SSDs.

RocksDB was developed based on the Log-Structured Merge (LSM) tree [19] and manages its data using two data structures: *memtable* and sorted static tables (SSTs). The written data are initially pushed into memtable, which is an in-memory write buffer for write batching. When a memtable is full, it creates an immutable SST, which is stored on the persistent storage. By doing this, RocksDB avoids frequent re-writing of existing blocks, which degrade the performance and durability of SSDs.

Flink and Samza utilize RocksDB as their local state stores. They launch a separate KV-store instance for each node inside the cluster, and operators in each node use the local RocksDB instance running in the same node to avoid network transfer overhead.

3 ANALYSIS OF STATEFUL STREAM PROCESSING ON FLINK WITH ROCKSDB

Delegating internal state management to a persistent KV store is convenient, but it often degrades the performance of stream processing pipelines. This is because persistent KV stores are not aware of query semantics, and thus they cannot optimize the state management strategy according to the semantics.

In this section, we analyze the overhead of delegating stream state management to persistent KV stores. Previous work by Noghabi et al. [18] has evaluated the performance of Samza with the memory and the RocksDB state backend. Compared to their work, our evaluation has two differences. First, we find the root cause of the performance degradation by breaking down the time spent in each step of stream processing pipelines. Second, we suggest an optimized state management strategy for the query used in our evaluation and compare its performance with that of Flink with the RocksDB state backend.

Action	Time (ms)
read	97248 (44.17%)
write	39157 (17.79%)
serialization and deserialization	47650 (21.6%)
computation, query setup, ...	36095 (16.40%)
Total query execution time	220150

Table 1: Performance Breakdown of Flink with RocksDB (WB=1024MB).

3.1 Evaluation Setup

We evaluate Flink with the memory state backend and Flink with the RocksDB state backend, which is an officially recommended option [1] for managing large states on persistent storages.

Environment. We ran Flink 1.6 on a 28-core NUMA machine (2x Intel Xeon E5-2680 2.4GHz, 8x 16GB RDIMM, Ubuntu 16.04.1) with an Intel Optane 900P 480GB NVMe SSD connected via a PCI-Express 3.0 4x interface.

Query Workload and Metrics. We evaluate the systems with the *ReadWrite* query presented in the Samza paper [18]. The ReadWrite query has a single data stream with tuples (id, padding). *id* is a randomly generated integer key within a certain range, and *padding* is a sequence of a randomly generated byte values. We set the size of padding to be 100 bytes, matching the value in the original experiment. The query maintains internal state containing (id, count, padding) tuples for all the ids. The count and padding values are updated upon each data arrival. We set the number of keys to ten millions, and thus its internal state size reached 1.1GB. In our evaluation, we measure the time taken to digest 11GB tuples for ReadWrite query and calculate the throughput as the number of tuples processed per second.

System Configuration. We launched eight Flink tasks in a single machine for our evaluation. We ran Flink with the RocksDB state backend and varied the write buffer sizes to 256MB, 512MB, and 1024MB, which determines the size of the RocksDB memtable. Among the memtable implementations, we chose the skip list [20], which is the default option in RocksDB, because the other implementations using different data structures do not support concurrent insertions. For SST format, we chose the block-based table, the default SST format in RocksDB, because it is recommended for storing data in SSDs or HDDs [6]. We allocated 40GB memory for the RocksDB block cache to support fast read operations.

3.2 Results

Figure 2 shows the result of our preliminary evaluation on the ReadWrite query with 1.1GB state. The result shows that the performance of Flink with RocksDB is much lower than that of in-memory Flink in every configuration, even when the write buffer size is 1024MB, which is almost equal to the total state size. When the allocated memory for the RocksDB write buffer becomes smaller, the throughput worsens because of extra I/O overhead on SSDs. To identify the cause of the performance degradation in RocksDB with 1024MB write buffer, we measured the time taken for each step (RocksDB read & write, serialization, and others) inside a Flink worker. The result presented in Table 1 shows that the RocksDB read and write time takes more than half of the entire execution time, even though the majority of the RocksDB operations are done in memory.

The performance degradation results from extra operations in the RocksDB that are not necessary for the ReadWrite query. To deal with the ReadWrite query, high random read-/write performance is important because of frequent random point updates. The RocksDB memtable manages its data in sorted order via a skip list, which causes $O(\log n)$ computation overhead for accessing its data. With this strategy, RocksDB offers reasonable read/write performance with the support of efficient range read and iterations in sorted order. Nevertheless, the queries whose point-update performance is important, this approach adds unnecessary overhead to the stream processing pipeline.

There are implementations of RocksDB memtable that support $O(1)$ read and write through hash function, such as hash skip list and hash linked list. However, because they do not support concurrent writes, they cannot be applied to our evaluation environment with multiple Flink tasks.

It is important to note that this result does not indicate that there is a general performance problem on RocksDB. As RocksDB is developed as a general-purpose KV store, it is not optimized for stream query workloads that require high point-update performance. Instead, RocksDB provides good performance for diverse KV-store operations including iterations and range operations.

3.3 State Management Optimization on SSDs

In this section, we show the potential performance benefit of query-aware optimizations on stateful stream state management with SSDs by implementing proper state management strategies for the ReadWrite query on Flink. We take the following two approaches from existing works on persistent KV stores [2, 4, 9, 14, 23].

- **Non-sorted data organization:** As the ReadWrite query does not require range reads/writes or iterations in sorted order, we manage the index table and write buffer in non-sorted hash maps that guarantee higher random read/write performance [9, 14]. To efficiently support concurrent writes from multiple tasks, each task in Flink TaskManager maintains a separate index table, a write buffer, and data files stored in SSDs. By taking this approach, we eliminate the overhead of dealing with concurrent writes on a shared data structure, which degrades the scalability in multi-core machines. Each element in the index table keeps the byte offset of the value stored on the SSD for fast data retrieval. This way, we provide $O(1)$ access for data stored in both in-memory write buffer and SSDs.
  In some query workloads, keys themselves with small values can dominate the total state size. In this case, the index table could be too large to be stored in the memory. This problem can be mitigated by adopting prior techniques used to reduce the size of metadata [23].
- **Batched append-only writes:** We also follow the append-only write strategy of persistent KV stores that are optimized for SSDs [2, 4]. We maintain a write buffer cache that stores pending writes until it reaches its size limit. When the write buffer is full, it flushes the batched writes to the log file and saves the byte offset to the index table. To avoid frequent block rewrites, which degrade the performance and life span of SSDs, all the writes are done via appending. Each updated value is appended to the end of the log files. We set the write batch size to be 10000 for our evaluation.

In Figure 2, we compare the ReadWrite query performance of Flink with our custom SSDStateBackend that adopts the two optimization techniques above to that of Flink with the RocksDB state backend. Our custom SSDStateBackend improves the performance by 2.16×–3.42× for the ReadWrite query compared to the RocksDB state backend.

4 SYSTEM DESIGN PROPOSAL

Although our optimized state backend works well on the ReadWrite query, it would not work well on non-associative window operators or operators with historical data access, because it does not support append and range read operations efficiently. To address the performance problem in stateful stream processing on SSDs, we believe that it is necessary to develop a new stream processing system that optimizes state management plans automatically according to the stream query semantics. In this section, we propose such a new system design.

4.1 System Overview

The main goal of our system is optimizing large state management on SSDs via inferred stream query semantics and runtime metrics. Figure 3 shows the overall structure of our

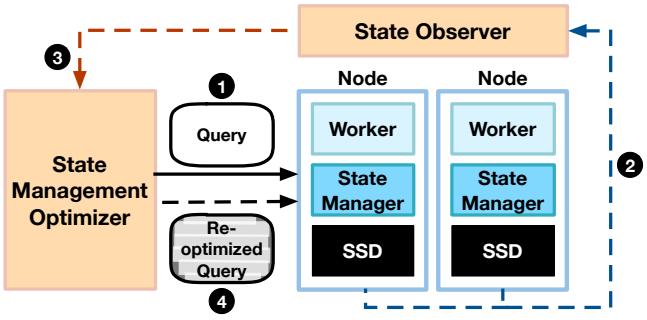

Figure 3: System design overview. ❶ A query plan with static state management optimization is submitted to worker nodes. ❷ For stream operators that need runtime optimization, the states are initially stored in memory. State observer monitors the state access operations (e.g., point/range read, update, append) and gathers the statistics on them. ❸ Based on the statistics on runtime state access, the optimizer builds a new query plan with an optimized state management strategy for the operator. ❹ The states are migrated from main memory to SSDs according to the optimized strategy.

proposed system design. Before executing a query, we apply static optimization based on the inferred state read/write pattern from the query semantics. During query execution, runtime optimization is done based on the monitored state access patterns.

4.2 Stream-Query-Aware Static Optimization

When a stream query is submitted to the system, the state management optimizer analyzes the semantics of stateful operators inside the stream query and infers the access pattern for each operator. The inferred state read/write patterns are classified into several categories such as "Point-Update Intensive" for associative window aggregation, "Append Intensive" for non-associative window aggregation, and "Range-read Intensive" for operators with historical data.

If an operator is written in pre-defined functions, the optimizer can easily understand the state access pattern of the operator. However, stateful stream operators are often written in user-defined functions (UDFs) whose semantics are hidden from the stream processing system. One possible solution is letting users provide hints on the state read/write pattern of the given UDF through annotations, such as "@PointUpdate". For UDFs without user hints, it would be hard to accurately anticipate their state access patterns. In this case, we rely on runtime optimizations, which we explain in Section 4.3.

After the analysis is done, the optimizer creates a state management plan that is optimized to the classified state access pattern for each stateful operator. We briefly summarize three examples of state management strategies for the state access patterns of commonly used stream operators described in Section 2.1.

Point-Update Intensive. For operators that create frequent updates on tuples of the internal states, the system applies the optimization strategy described in Section 3.3.

Append Intensive. Non-associative window aggregations append new data into internal states. For this pattern, the system pre-allocates space on SSDs for each key in advance for the appended data in near future. Through the optimization, we can avoid huge write amplification in naive append implementations, where an append operation is performed by copying the original data and rewriting the data elsewhere with the appended values. In addition, we can avoid huge indexes and a large number of random reads as in NoVoHT [3], which maintains multiple index entries for the fragmented appended values.

Range-read Intensive. To support range reads efficiently, the system should maintain the internal states sorted by the event time. By understanding the query semantics, the system can also prefetch the data likely to be read in the near future in advance.

4.3 Runtime Optimization

When stream queries with user-defined operators are submitted, it is hard to accurately anticipate their state access patterns without user hints. This makes the system choose sub-optimal state access strategies, which degrades the performance of the stream pipelines. To solve this problem, we propose a runtime optimization that builds the state management plan for UDF operators based on the monitored state access patterns during runtime.

Our runtime optimization is done through the following steps. When initially deployed, the operators with no confident anticipation on their state access patterns store their states in main memory. In the early stage of their execution, the size of their states is small enough to be stored in main memory. The state observer monitors the state access operations (e.g., point/range read, update, append) and gathers the statistics. When the state size becomes large, the state management optimizer builds a state management plan based on the statistics gathered during the runtime. After that, the states stored in main memory are migrated to SSDs and are stored according to the optimized plan. The migration is performed by background threads, to avoid blocking the event processing during the migration. While the migration process is ongoing, the states are temporarily stored both in main memory and SSDs.

5 DISCUSSION

In this section, we discuss additional research directions regarding stateful stream processing on SSDs. We focus on how we integrate emerging storage technologies with stateful stream processing, such as non-volatile memory express over fabrics (NVMf), remote direct memory access (RDMA), and near-data processing (NDP).

5.1 Peer-to-peer Checkpointing via NVMf

Fault tolerance is an important problem in stream processing, because stream processing systems should deliver timely results even when failures occur. In stream processing, checkpointing internal states to remote stable storages (e.g., HDFS, S3) and recovering the states by replaying the events from the checkpoint time is a popular strategy for fault recovery [8, 11, 21]. NVMf enables direct peer-to-peer (P2P) among NVMe SSDs connected via PCI-express and RDMA-supported networks [13]. By applying this technology to stream processing, we can accelerate the checkpointing process of internal states with little additional CPU overhead. One candidate design is checkpointing the internal state of each node to another node via P2P RDMA. This enables efficient checkpointing without copying the internal states to distributed file systems [24].

5.2 Leveraging Near-Data Processing

Near-data processing (NDP) accelerates the data processing pipeline by placing the computing power near data. NDP has already proven its effectiveness for processing database workloads [15, 16]. NDP can also be applied to stateful stream processing on SSDs, because all the states are already stored in persistent storages. Before loading the states into the main memory, we can pre-aggregate the data in SSDs via NDP and load only the aggregated data. By doing this, we can save the cost of loading data from the persistent storage to the main memory.

6 RELATED WORK

Stateful Stream Processing. Stateful stream processing is an important topic in the field of big data analytics and has been studied in many previous works [8, 10, 11, 18, 26, 27]. Flink [10] and Samza [18] support stateful stream processing on SSDs with RocksDB. We have shown the inefficiencies of their approaches in Section 3. Samza mitigates this problem by adding in-memory caching layers between RocksDB and Samza. However, as shown in the evaluations of the Samza paper, Samza does not handle workloads with poor data localities efficiently.

Wukong+S [27] processes a large number of concurrent stream queries on linked data with low latency by integrating the connected graph storage and stream processing layer in a novel way. Different from their work, our work focuses on efficiently processing a small number of stateful queries by understanding stream access patterns of diverse stream queries.

SummaryStore [7] handles colossal data streams with low latency by gracefully degrading the fidelity of the stored data over time. Their techniques are orthogonal to our work and can help improve the performance of our system with low loss in accuracy.

Persistent KV Stores. There are many existing studies on persistent KV stores [2, 4, 9, 12, 14, 17, 23]. Our optimization leverages various optimization techniques used in those systems, such as append-only writes on SSDs [2, 4] or leveraging non-sorted data structures [9, 14, 23]. Faster [12] further optimizes the performance of in-place updates that are common in stream processing workloads. To achieve its goal, Faster adopts highly-concurrent hybrid store that spans both on memory and disk.

However, as persistent KV stores are unaware of stream query semantics, they cannot automatically change their state management strategies according to diverse state access patterns of the queries. This could result in potential performance degradation.

NVMe over Fabrics and Near-Data Processing. Emerging technologies such as NVMf and NDP are becoming popular in data analytics systems. NVMf offers fast remote access to NVMe SSD drives connected in RDMA-enabled networks. By exploiting NVMf, Apache Crail [22] optimizes Spark [25] on NVMe SSDs. NDP accelerates data processing pipelines by moving computation close to storage. Biscuit [16] improves the performance of processing the TPC-H workload leveraging NDP. In this paper, we discuss potential research directions for adopting NVMf and NDP to accelerate stateful stream processing on SSDs.

7 CONCLUSION

As the size of internal states in stateful stream processing becomes large, efficiently handling the states on SSDs becomes an important problem. In this paper, we look at the performance problem of current stateful stream processing on SSDs resulting from the lack of awareness of the stream query semantics. In addition, we show that the problem can be mitigated by adopting an optimized state management strategy that considers the state access patterns of the stream processing applications. We propose a system design that automatically optimizes stateful stream processing on SSDs, and also discuss how we can integrate emerging technologies in storage systems with stateful stream processing.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. We also thank Dr. Nae Young Song, Sanha Lee, Junwen Yang, and the members of SNU Software Platform Lab for their help to improve the paper. This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1503-01.

REFERENCES

[1] Flink State Backends. https://ci.apache.org/projects/flink/flink-docs-master/ops/state/state_backends.html.
[2] LevelDB. http://leveldb.org/.
[3] NoVoHT. https://github.com/kev40293/NoVoHT.
[4] RocksDB. http://rocksdb.org/.
[5] Stream Processing Hard Problems Part II: Data Access. https://engineering.linkedin.com/blog/2016/08/stream-processing-hard-problems-part-ii--data-access.
[6] A Tutorial of RocksDB SST formats. https://github.com/facebook/rocksdb/wiki/A-Tutorial-of-RocksDB-SST-formats.
[7] Nitin Agrawal and Ashish Vulimiri. 2017. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *SOSP*. ACM.
[8] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *VLDB* 6, 11 (2013), 1033–1044.
[9] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*. ACM.
[10] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *VLDB* 10, 12 (2017), 1718–1729.
[11] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *SIGMOD*. ACM.
[12] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*. ACM.
[13] Rob Davis and Idan Burstein. 2016. NVMe Over Fabrics—High Performance Flash Moves to Ethernet. *Storage Developer Conference* (2016).
[14] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash. In *SIGMOD*. ACM.
[15] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *PACT*. IEEE.
[16] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *ISCA*. IEEE.
[17] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*. ACM.
[18] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *VLDB* 10, 12 (2017), 1634–1645.
[19] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[20] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.
[21] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *EuroSys*. ACM.
[22] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017), 38–49.
[23] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *ATC*. USENIX.
[24] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *ICDE*. IEEE.
[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. USENIX.
[26] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*. ACM.
[27] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *SOSP*. ACM.