Hardware-Conscious Stream Processing: A Survey

Shuhao Zhang[1], Feng Zhang[2], Yingjun Wu[3], Bingsheng He[1], Paul Johns[1]
[1]National University of Singapore, [2]Renmin University of China,
[3]Amazon Web Services

ABSTRACT

Data stream processing systems (DSPSs) enable users to express and run stream applications to continuously process data streams. To achieve real-time data analytics, recent researches keep focusing on optimizing the system latency and throughput. Witnessing the recent great achievements in the computer architecture community, researchers and practitioners have investigated the potential of adoption hardware-conscious stream processing by better utilizing modern hardware capacity in DSPSs. In this paper, we conduct a systematic survey of recent work in the field, particularly along with the following three directions: 1) computation optimization, 2) stream I/O optimization, and 3) query deployment. Finally, we advise on potential future research directions.

1 Introduction

A large volume of data is generated in real time or near real time and has grown explosively in the past few years. For example, IoT (Internet-of-Things) organizes billions of devices around the world that are connected to the Internet. IHS Markit forecasts [3] that 125 billion such devices will be in service by 2030, up from 27 billion in 2018. With the proliferation of such high-speed data sources, numerous data-intensive applications are deployed in real-world use cases exhibiting latency and throughput requirements, that can not be satisfied by traditional batch processing models. Despite the massive effort devoted to big data research, many challenges remain.

A data stream processing system (DSPS) is a software system which allows users to efficiently run stream applications that continuously analyze data in real time. For example, modern DSPSs [5, 6] can achieve very low processing latency in the order of milliseconds. Many research efforts are devoted to improving the performance of DSPSs from the research community [45, 23, 98, 92] and leading enterprises such as SAP [102], IBM [37], Google [9] and Microsoft [19]. Despite the success of the last several decades, more radical performance demand, complex analysis, as well as intensive state access in emerging stream applications [21, 91] pose great challenges to existing DSPSs. Meanwhile, significant achievements have been made in the computer architecture community, which has recently led to various investigations of the potential of *hardware-conscious DSPSs*, which aim to exploit the potential of accelerating stream processing on modern hardware [104, 98].

Fully utilizing hardware capacity is notoriously challenging. A large number of studies have been proposed in recent years [19, 66, 57, 58, 45, 98, 103, 104]. This paper hence aims at presenting a systematic review of prior efforts on hardware-conscious stream processing. Particularly, the survey is organized along with the following three directions: 1) computation optimization, 2) stream I/O optimization, and 3) query deployment. We aim to show what has been achieved and reveal what has been largely overlooked. We hope that this survey will shed light on the hardware-conscious design of future DSPSs.

2 Background

In this section, we introduce the common APIs and runtime architectures of modern DSPSs.

2.1 Common APIs

A DSPS needs to provide a set of APIs for users to express their stream applications. Most modern DSPSs such as Storm [6] and Flink [5] express a streaming application as a directed acyclic graph (DAG), where nodes in the graph represent operators, and edges represent the data dependency between operators. Figure 1 (a) illustrates the *word count* (WC) as an example application containing five operators. A detailed description of a few more stream applications can be found in [103].

Some earlier DSPSs (e.g., Storm [6]) require users to implement each operator manually. Recent efforts from Saber [45], Flink [5], Spark-Streaming [96], and Trident [55] aim to provide

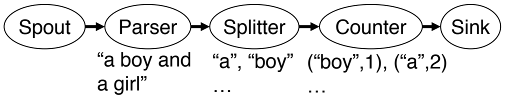

Figure 1: A stream processing example of word count.

declarative APIs (e.g., SQL) with rich built-in operations such as aggregation and join. Subsequently, many efforts have been devoted to improving the execution efficiency of the operations, especially by utilizing modern hardware (Section 4).

2.2 Common Runtime Architectures

Modern stream processing systems can be generally categorized based on their processing models including the Continuous Operator (CO) model and the Bulk Synchronous Parallel (BSP) model [87].

Continuous Operator Model: Under the CO model, the execution runtime treats each operator (a vertex of a DAG) as a single execution unit (e.g., a Java thread), and multiple operators communicate through message passing (an edge in a DAG). For scalability, each operator can be executed independently in multiple threads, where each thread handles a substream of input events with stream partitioning [43]. This execution model allows users to control the parallelism of each operator in a fine-grained manner [103]. This kind of design was adopted by many DSPSs such as Storm [6], Heron [49], Seep [17], and Flink [5] due to its advantage of low processing latency. Other recent hardware-conscious DSPSs adopt the CO model including Trill [19], BriskStream [104], and TerseCades [66].

Bulk-Synchronous Parallel Model: Under the BSP model, input stream is explicitly grouped into micro batches by a central coordinator and then distributed to multiple workers (e.g., a thread/machine). Subsequently, each data item in a micro batch is independently processed by going through the entire DAG (ideally by the same thread without any cross-operator communication). However, the DAG may contain synchronization barrier, where threads have to exchange their intermediate results (i.e., data shuffling). Taking WC as an example, the `Splitter` needs to ensure that the same word is always passed to the same thread of the `Counter`. Hence, a data shuffling operation is required before the `Counter`. As a result, such synchronization barriers break the DAG into multiple stages under the BSP model, and the communication between stages is managed by the central coordinator. This kind of design was adopted by Spark-streaming [96], Drizzle [87], and FlumeJava [18]. Other recent hardware-conscious DSPSs adopt the BSP model including Saber [45] and StreamBox [57].

Although there have been prior efforts to compare different models [82], it is still inconclusive that which model is more suitable for utilizing modern hardware – each model comes with its own advantages and disadvantages. For example, the BSP model naturally minimizes communication among operators inside the same DAG, but its single centralized scheduler has been identified with scalability limitation [87]. Moreover, its unavoidable data shuffling also brings significant communication overhead, as observed in recent research [104]. In contrast, CO model allows fine-grained optimization (i.e., each operator can be configured with different parallelisms and placements) but potentially incurs higher communication costs among operators. Moreover, the limitations of both models can potentially be addressed with more advanced techniques. For example, cross operator communication overhead (under both CO and BSP models) can be overcome by exploiting tuple batching [19, 103], high bandwidth memory [58, 70], data compression [66], InfiniBand [38] (Section 5), and architecture-aware query deployment [104, 98] (Section 6).

3 Survey Outline

The hardware architecture is evolving fast and provides a much higher processing capability than that traditional DSPSs were originally designed for. For example, recent *scale-up* servers can accommodate hundreds of CPU cores and terabytes of memory [4], providing abundant computing resources. Emerging network technologies such as Remote Direct Memory Access (RDMA) and 10Gb Ethernet significantly improve system ingress rate, making I/O no longer a bottleneck in many practical scenarios [57, 21]. However, prior studies [103, 98] have shown that existing data stream processing systems (DSPSs) severely underutilize hardware resources due to the unawareness of the underlying complex hardware architectures.

As summarized in Table 1, we are witnessing a revolution in the design of DSPSs that exploit emerging hardware capability, particularly along with the following three dimensions:

1) Computation Optimization: Contrary to conventional DBMSs, there are two key features in DSPSs that are fundamental to many stream applications and computationally expensive: *Windowing operation* [35] (e.g., windowing stream join) and *Out-of-order handling* [12]. The former

Table 1: Summary of the surveyed works

Research Dimensions	Key Concerns	Key Related Work
Computation Optimization	Synchronization overhead, work efficiency	CellJoin [23], FPGAJoin [47, 72], Handshake join [83, 73], PanJoin [65], HELLS-join [42, 41], Aggregation on GPU [45], Aggregation on FPGA [64, 24], Hammer Slide [84], StreamBox [57], Parallel Index Join [75]
Stream I/O Optimization	Time and space efficiency, data locality, and memory footprint	Batching [103], Stream with HBM [58, 70], TerseCades [66], Stream over InfiniBand [38], Stream on SSDs [51], and NVM-aware Storage [68]
Query Deployment	Operator interference, elastic scaling, and power constraint	Orchestrating [48, 22], StreamIt [16], SIMD [36], BitStream [8], Streams on Wires [60], HRC [88], RCM [89], CMGG [63], GStream [106], SABER [45], BriskStream [104]

deals with infinite stream, and the latter handles stream imperfection. The support for those expensive operations is becoming one of the major requirements for modern DSPSs and is treated as one of the key dimensions in differentiating modern DSPSs. Prior approaches use multicores [84, 57], heterogeneous architectures (e.g., GPUs and Cell processors) [23, 45], and Field Programmable Gate Arrays (FPGAs) [83, 73, 47, 72, 64, 24] for accelerating those operations.

2) Stream I/O Optimization: Cross-operator communication [103] is often a major source of overhead in stream processing. Recent work has revealed that the overhead due to cross-operator communication is significant, even without the TCP/IP network stack [103, 98]. Subsequently, research has been conducted on improving the efficiency of data grouping (i.e., output stream shuffling among operators) using High Bandwidth Memory (HBM) [58], compressing data in transmission with hardware accelerators and applying computation directly over compressed data [66], and leveraging InfiniBand for faster data flow [38]. Having said that, there are also cases where the application needs to temporarily store data for future usage [85] (i.e., state management [15]). Examples include stream processing with large window operation (i.e., workload footprint larger than memory capacity) and stateful stream processing with high availability (i.e., application states are kept persistently). To relieve the disk I/O overhead, recent work has investigated how to achieve more efficient state management, leveraging SSD [51] and non-volatile memory (NVM) [68].

3) Query Deployment: At an even higher point of view, researchers have studied launching a whole stream application (i.e., a query) into various hardware architectures. Similar to traditional database systems, the goal of query deployment in DSPS is to minimize operator interference/ cross-operator communication, balance hardware resource utilization, and so on. The major difference compared to traditional database systems lies in their different problem assumptions, and hence in their system architectures (e.g., infinite input stream [90], processing latency constraints [33], and unique cost function of streaming operators [102, 44]). To take advantage of modern hardware, prior works have exploited various hardware characteristics such as cache-conscious strategies [8], FPGA [60], and GPUs [88, 89, 63, 106]. Recent works have also looked into supporting hybrid architectures [45] and NUMA [104].

4 Computation Optimization

In this section, we review the literature on accelerating computationally expensive streaming operations using modern hardware.

4.1 Windowing Operation

In stream applications, the processing is mostly performed in the form of long-running queries known as continuous queries [11]. To handle potentially infinite data streams, continuous queries are typically limited to a window that limits the number of tuples to process at any point in time. The window can be defined based on the number of tuples (i.e., count based), function of time (i.e., time based) or sessions [86]. Window stream joins and window aggregation are two common expensive windowing operations in data stream processing.

4.1.1 Window Join

A common operation used in many stream analytical workloads is to *join* multiple data streams. Different from traditional join in relational databases [32], which processes a large batch of data at once, stream join has to produce results on the fly [39, 77, 31, 26]. By definition, the stream join operator performs over infinite streams. In practice, streams are cut into finite slices/windows [93]. In a two-way stream join, tuples from the left stream (R) are joined with tuples in the right stream (S) when the specified key attribute matches, and the timestamp of tuples from both streams falls within the same window.

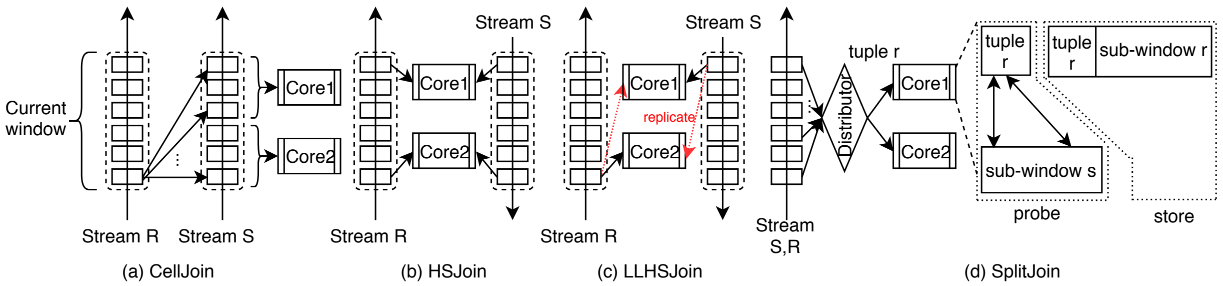

Figure 2: HW-conscious stream join algorithms (using two cores as an example).

Algorithm Overview. Kang et al. [39] described the first streaming join implementations. For each tuple r of stream R, 1) Scan the window associated with stream S and look for matching tuples; 2) Invalidate the old tuples in both windows; 3) Insert r into the window of R.

HW-Conscious Optimizations. The costly nature of stream join and the stringent response time requirements of stream applications have created significant interest in accelerating stream joining. Multicore processors that provide high processing capacity are ideal for executing costly windowed stream operators. However, fully exploiting the potential of a multicore processor is often challenging due to the complex processor microarchitecture, deep cache memory subsystem, and the unconventional programming model in general. Figure 2 illustrates the four representative studies on accelerating window stream joins described as follows.

CellJoin: An earlier work from Gedik et al. [23], called CellJoin, attempt to parallelize stream join on Cell processor, a heterogeneous multicore architecture. CellJoin generally follows Kang's [39] three-step algorithm. To utilize multicores, it re-partitions S, and each resulting partition is assigned to an individual core. In this way, the matching step can be performed in parallel on multiple cores. A similar idea has been adopted in the work by Karnagel et al. [42] to utilize the massively parallel computing power of GPU.

Handshake-Join (HSJoin): CellJoin essentially turns the join process into a scheduling and placement process. Subsequently, it is assumed that the window partition and fetch must be performed in global memory. The repartition and distribution mechanism essentially reveals that CellJoin generally follows the BSP model (see Section 2.2). This is later shown to be ineffective when the number of cores is large [83], and a new stream join technique called handshake join (i.e., HSJoin) was proposed. In contrast to CellJoin, HSJoin adopts the CO model. Specifically, both input streams notionally flow through the stream processing engine in opposite directions. As illustrated in Figure 2 (b), the two sliding windows are laid out side by side, and predicate evaluations are continuously performed along with the windows whenever two tuples encounter each other.

Low-Latency Handshake-Join (i.e., LLHSJoin): Despite its excellent scalability, the downside of HSJoin is that tuples may have to be queued for long periods of time before the match, resulting in high processing latency. In response, Roy et al. [73] propose a *low-latency* handshake-join (i.e., LLHSJoin) algorithm. The key idea is that, instead of sequentially forwarding each tuple through a pipeline of processing units, tuples are replicated and forwarded to all involved processing units (see the red dotted lines in Figure 2 (c)) before the join computation is carried out by one processing unit (called a home node).

SplitJoin: The state-of-the-art windowing join implementation called SplitJoin [62] parallelizes the join process via the CO model. Rather than forwarding tuples bidirectionally, as in HSJoin or LLHSJoin, SplitJoin broadcasts each newly arrived tuple t (from either S or R) to all processing units. In order to make sure that each tuple is processed only once, t is retained in exactly one processing unit chosen in a round-robin manner. Although SplitJoin [62] and HSJoin [83] can achieve the same concurrency theoretically without any central coordination, the former achieves a much lower latency due to the linear chaining delay of the HSJoin. While LLHSJoin [73] reduces the processing latency of HSJoin [83] by using a fast forwarding mechanism, it complicates the processing logic and reintroduces central coordination to the processing [62].

4.1.2 *Window Aggregation*

Another computationally heavy windowing operation is window aggregation, which summarizes the most recent information in a data stream. There are four workload characteristics [86] of stream aggregation including 1) window type, which refers to the logic based on which system derives finite windows from a continuous stream, such as tumbling, sliding, and session; 2) windowing measures, which refers to ways to measure windows, such as time-based, count-based, and any other arbitrary advancing measures; 3) aggregate functions with different algebraic properties [81] such as *invertible*, *associative*, *commutative*, and *order-preserving*; and 4) stream (dis)order, which shall be discussed in Section 4.2.

Algorithm Overview. The trivial implementation is to perform the aggregation calculation from scratch for every arrived data. The complexity is hence $O(n)$, where n is the window size. Intuitively, efficiently leveraging previous calculation results for future calculation is the key to reducing computation complexity, which is often called incremental aggregation. However, the effectiveness of incremental aggregation depends heavily on the aforementioned workload characteristics such as the property of the aggregation function. For example, when the aggregation function is invertible (e.g., sum), we can simply update (i.e., increase) the aggregation results when a new tuple is inserted into the window and evict with the time complexity of $O(1)$. For faster answering median like function, which has to keep all the relevant inputs, instead of performing a sort on the window for each newly inserted tuple, one can maintain an *order statistics tree* as auxiliary data structure [34], which has $O(logn)$ worst-case complexity of its insertion, deletion, and rank function. Similarly, the reactive aggregator (RA) [80] with $O(logn)$ average complexity only works for aggregation function with the associative property. Those algorithms also differ from each other at their capability of handling different window types, windowing measures, and stream (dis)order [86]. Traub et al. [86] recently proposed a generalization of the stream slicing technique to handle different workload characteristics for window aggregation. It may be an interesting future work to study how the proposed technique can be applied to better utilize modern hardware architectures (e.g., GPUs).

HW-Conscious Optimizations. There are a number of works on accelerating windowing aggregation in a hardware-friendly manner. An early work by Mueller et al. [59] described implementation for a sliding windowed median operator on FPGAs. This is an operator commonly used to, for instance, eliminate noise in sensor readings and in data analysis tasks [71]. The algorithm skeleton adopted by the work is rather conventional: it first sorts elements within the sliding window and then computes the median. Compared to the $O(logn)$ complexity of using an order statistics tree as an auxiliary data structure [34], Mueller's method has a theoretically much higher complexity due to the sorting step ($O(nlogn)$). Nevertheless, their key contribution is on how the sorting and computing steps can be efficiently performed on FPGAs. Mueller's implementation [59] focuses on efficiently processing one sliding window without discussing how to handle subsequent sliding windows. Mueller et al. hence proposed conducting multiple computations for each sliding window by instantiating multiple aggregation modules concurrently [60].

Recomputing from scratch for each sliding window is costly, even if conducted in parallel [60]. Hence, a technique called pane [52] was proposed and later verified on FPGAs [64] to address this issue. The key idea is to divide overlapping windows into disjoint panes, compute sub-aggregates over each pane, and "roll up" the partial-aggregates to compute final results. Pane was later improved [46] and covers more cases (e.g., to support non-periodic windows [14]). However, the latest efforts are mostly theoretical, and little work has been done to validate the effectiveness of these techniques on modern hardware, e.g., FPGA and GPUs.

Saber [45] is a relational stream processing system targeting heterogeneous machines equipped with CPUs and GPUs. To achieve high throughput, Saber also adopts incremental aggregation computations utilizing the commutative and associative property of some aggregation functions such as count, sum, and average. Theodorakis et al. [84] recently studied the trade-off between workload complexity and CPU efficient streaming window aggregation. To this end, they proposed an implementation that is both workload- and CPU-efficient. Gong et al. [27] proposed an efficient and scalable accelerator based on FPGAs, called ShuntFlow, to support arbitrary window sizes for both reduce- and index-like sliding window aggregations. The key idea is to partition aggregation with extremely large window sizes into sub-aggregations with smaller window sizes that can enable more efficient use of FPGAs.

4.2 Out-of-Order Handling

In a real production environment, out-of-order[1] input data are not uncommon. A stream operator is considered to be *order-sensitive* if it requires input events to be processed in a certain predefined

[1] Other issues such as delay and missing can be seen as special cases of out-of-order.

order (e.g., chronological order). Handling out-of-order input data in an order-sensitive operator often turns out to be a performance bottleneck, as there is a fundamental conflict between data parallelism and order-sensitive processing – the former seeks to improve the throughput of an operator by letting more than one thread operate on different events concurrently, possibly out-of-order.

Algorithm Overview. Currently, there are three general techniques to be applied together with the order-sensitive operator to handle out-of-order data streams. The first utilizes a buffer-based data structure [12] that buffers incoming tuples for a while before processing. The key idea is to keep the data as long as possible (within the latency/buffer size constraint) to avoid out-of-order inputs. The second technique relies on punctuation [53], which is a special tuple in the event stream indicating the end of a substream. Punctuations guarantee that tuples are processed in monotonically increasing time sequence across punctuations, but not within the same punctuation. The third approach is to use speculative techniques [74]. The main idea is to process tuples without any delay, and recompute the results in the case of order violation. There are also techniques specifically designed for handling out-of-order in a certain type of operator such as window aggregation [86].

HW-Conscious Optimizations. Gulisano et al. [28] are among the first to handle out-of-order for high-performance stream join on multi-core CPUs. The proposed algorithm, called *scalejoin* is illustrated in Figure 3 (a). It first merges all incoming tuples into one stream (through a data structure called *scalegate*) and then distributes them to processing threads (PTs) to perform join. The output also needs to be merged and sorted before exiting the system. The use of the *scalegate* makes this work fall into the category of buffer-based approach and have inherent limitation of higher processing latency. Scalejoin has been implemented in FPGA [47] and further improved in another recent work [72]. They both found that the proposed system outperforms the corresponding fully optimized parallel software-based solution running on a high-end 48-core multiprocessor platform.

StreamBox [57] handles out-of-order event processing by the punctuation-based technique on multicore processors. Figure 3 (b) illustrates the basic idea of taking the stream join operator as an example. Relying on a novel data structure called *cascading container* to track dependencies between epochs (a group of tuples delineated by punctuation), StreamBox is able to maintain the processing order among multiple concurrently executing containers that exploit the parallelism of modern multicore hardware.

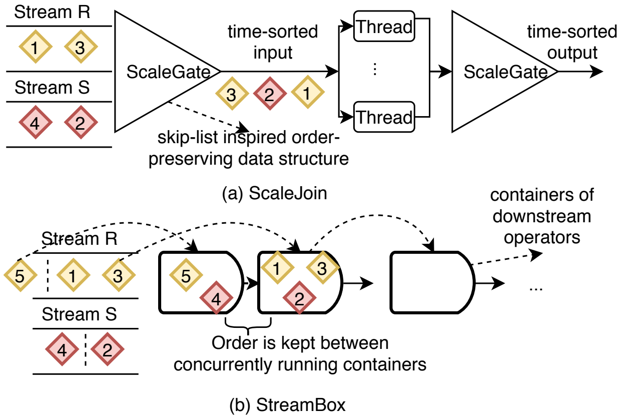
Stream R
1
3
Stream S
4
2
ScaleGate
time-sorted input
3
2
1
Thread
⋮
Thread
ScaleGate
time-sorted output
skip-list inspired order-preserving data structure
(a) ScaleJoin
containers of downstream operators
Stream R
5
1
3
Stream S
4
2
5
4
1
3
2
...
Order is kept between concurrently running containers
(b) StreamBox

Figure 3: Multicore-friendly out-of-order handling.

Kuralenok et al. [50] attempt to balance the conflict between order-sensitive and multicore parallelism with an optimistic approach falling in the third approach. The basic idea is to conduct the joining process without any regulations, but apologize (i.e., sending amending signals) when the processing order is violated. They show that the performance of the proposed approach depends on how often reorderings are observed during run-time. In the case where the input order is naturally preserved, there is almost no overhead. However, it leads to extra network traffic and computations when reorderings are frequent. To apply such an approach to practical use cases, it is hence necessary to predict the probability of reordering, which could be an interesting future work.

4.3 Remarks

From the above discussion, it is clear that the key to accelerating windowing operators are mainly two folds. On the one hand, we should minimize the operation complexity. There are two common approaches: 1) incremental computation algorithms [45], which maximize reusing intermediate results, and 2) rely on efficient auxiliary data structures (e.g., indexing the contents of sliding window [95]) for reducing data (and/or instruction) accesses, especially cache misses. On the other hand, we should maximize the system concurrency [84]. This requires us to distribute workloads among multiple cores and minimize synchronization overhead among them [57]. Unfortunately, these optimization techniques are often at odds with each other. For example, incremental computation algorithm is complexity efficient but difficult to parallelize due to inherent control dependencies in the CPU instruction [84]. Another example is that maintaining index structures for partial computing results may help to reduce data accesses, but it also brings maintenance overhead [54]. More investigation is required to better balance these conflicting aspects.

5 Stream I/O Optimization

In this section, we review the literature on improving the stream I/O efficiency using modern hardware.

5.1 Cross-operator Communication

Modern DSPSs [5, 6] are able to achieve very low processing latency in the order of milliseconds. However, excessive communication among operators [103] is still a key obstacle in further improving the performance of the DSPSs.

Kamburugamuve et al. [38] recently presented their findings on integrating Apache Heron [49] with InfiniBand and Intel OmniPath. The results show that both can be utilized to improve the performance of distributed streaming applications. Nevertheless, many optimization opportunities remain to be explored. For example, prior work [38] has evaluated Heron on InfiniBand with channel semantics but not remote direct memory access (RDMA) semantics [67]. The latter has shown to be very effective in other related works [76, 97].

Data compression is a widely used approach for reducing communication overhead. Pekhimenko et al. [66] recently examined the potential of using data compression in stream processing. Interestingly, they found that data compression does not necessarily lead to a performance gain. Instead, improvement can only be achieved through a combination of hardware accelerator (i.e., GPUs in their proposal) and new execution techniques (i.e., compute directly over compressed data).

As mentioned before, word count requires the same word to be transmitted to the same `Counter` operator (see Section 2.1). Subsequently, all DSPSs need to implement data grouping operations regardless of their processing model (i.e., the continuous operator model or bulk synchronous model). Data grouping involves excessive memory accesses that rely on hash-based data structures [103, 96]. Zeuch et al. [98] analyzed the design space of DSPSs optimized for modern multicore processors. In particular, they show that a queue-less execution engine based on query compilation, which replaces communication between operators with function calls, is highly suitable for modern hardware. Since data grouping can not be completely eliminated, they proposed a mechanism called "Upfront Partitioning with Late Merging", for efficient data grouping. Miao et al. [58] have exploited the possibility of accelerating data grouping using emerging 3D stacked memories such as high-bandwidth memory (HBM). By designing the system in a way that addresses the limited capacity of HBM and HBM's need for sequential-access and high parallelism, the resulting system can achieve several times of performance improvement over the baseline.

5.2 State Management

Emerging stream applications often require the underlying DSPS to maintain large application states so as to support complex real-time analytics [85, 15]. Representative example states required during stream processing include graph data structures [105] and transaction records [56].

The storage subsystem has undergone tremendous innovation in order to keep up with the ever-increasing performance demand. Wukong+S [105] is a recently proposed distributed streaming engine that provides real-time consistent query over streaming datasets. It is built based on Wukong [76], which leverages RDMA to optimize throughput and latency. Wukong+S also follows its pace to support stream processing while maintaining low latency and high throughput. Non-Volatile Memory (NVM) has emerged as a promising hardware and brings many new opportunities and challenges. Fernando et al. [68] has recently explored efficient approaches to support analytical workloads on NVM, where an NVM-aware storage layout for tables is presented based on a multidimensional clustering approach and a block-like structure to utilize the entire memory stack. As argued by the author, the storage structure designed on NVM may serve as the foundation for supporting features like transactional stream processing systems [29] in the future. Non-Volatile Memory Express (NVMe) -based solid-state devices (SSDs) are expected to deliver unprecedented performance in terms of latency and peak bandwidth. For example, the recently announced PCIe 4.0 based NVMe SSDs [1] are already capable of achieving a peak bandwidth of 4GB/s. Lee et al. [51] have recently investigated the performance limitations of current DSPSs on managing application states on SSDs and have shown that query-aware optimization can significantly improve the performance of stateful stream processing on SSDs.

5.3 Remarks

Hardware-conscious stream I/O optimization is still in its early days. Most prior work attempts at mitigating the problem through a purely software approach, such as I/O-aware query deployment [94]. The emerging hardware such as Non-Volatile Memory (NVM) and InfiniBand with RDMA open up new opportunities for further improving stream I/O performance [105]. Meanwhile, the usage of emerging hardware accelerators such as GPUs further brings new opportunities to trade-off computation and communication overhead [66]. However, a model-guided approach to balance the trade-off is still generally missing in existing work. We hence expect more work to be done in this direction in the near future.

6 Query Deployment

We now review prior works from a higher level of abstraction, the query/application dimension. We summarize them based on their deployment targets: multicore CPUs, GPUs, and FPGAs.

6.1 Multicore Stream Processing

Language and Compiler. Multicore architectures have become ubiquitous. However, programming models and compiler techniques for employing multicore features are still lagging behind hardware improvements. Kudlur et al. [48] were among the first to develop a compiler technique to map stream application to a multicore processor. By taking the Cell processor as an example, they study how to compile and run a stream application expressed in their proposed language, called `StreamIt`. The compiler works in two steps: 1) operator fission optimization (i.e., split one operator into multiple ones) and 2) assignment optimization (i.e., assign each operator to a core). The two-step mapping is formulated as an integer linear programming (ILP) problem and requires a commercial ILP solver. Noting its NP-Hardness, Farhad et al. [22] later presented an approximation algorithm to solve the mapping problem. Note that the mapping problem from Kudlur et al. [48] considers only CPU loads and ignores communications bandwidth. In response, Carpenter et al. [16] developed an algorithm that maps a streaming program onto a heterogeneous target, further taking communication into consideration. To utilize a SIMD-enabled multicore system, Hormati et al. [36] proposed vectorizing stream applications. Relying on high-level information, such as the relationship between operators, they were able to achieve better performance than general vectorization techniques. Agrawal et al. [8] proposed a cache conscious scheduling algorithm for mapping stream application on multicore processors. In particular, they developed the theoretical lower bounds on cache misses when scheduling a streaming pipeline on multiple processors, and the upper bound of the proposed cache-based partitioning algorithm called *seg_cache*. They also experimentally found that scheduling solely based on the cache effects can often be more effective than the conventional load-balancing (based on computation cost) approaches.

Multicore-aware DSPSs. Recently, there has been a fast growing amount of interest in building multicore-friendly DSPSs. Instead of statically compiling a program as done in `StreamIt` [48, 22, 16], these DSPSs provide better elasticity for application execution. They also allow the usage of general-purpose programming languages (e.g., Java, Scala) to express stream applications. Tang et al. [79] studied the data flow graph to explore the potential parallelism in a DSPS and proposed an *auto-pipelining* solution that can utilize multicore processors to improve the throughput of stream processing applications. For economic reasons, power efficiency has become more and more important in recent years, especially in the HPC domains. Kanoun et al. [40] proposed a multicore scheme for stream processing that takes power constraints into consideration. Trill [19] is a single-node query processor for temporal or streaming data. Contrary to most distributed DSPSs (e.g., Storm, Flink) adopting the continuous operator model, Trill runs the whole query only on the thread that feeds data to it. Such an approach has shown to be especially effective [98] when applications contain no synchronization barriers.

6.2 GPU-Enabled Stream Processing

GPUs are the most popular heterogeneous processors due to their high computing capacity. However, due to their unconventional execution model, special designs are required to efficiently adapt stream processing to GPUs.

Single-GPU. Verner et al. [88] presented a general algorithm for processing data streams with real-time stream scheduling constraints on GPUs. This algorithm assigns data streams to CPUs and GPUs based on their incoming rates. It tries to provide an assignment that can satisfy different requirements from various data streams. Zhang et al. [100] developed a holistic approach to building DSPSs using GPUs. They design a latency-driven GPU-based framework, which mainly focuses on real-time stream processing. Due to the limited memory capacity of GPUs, the window size of the stream operator plays an important role in system performance. Pinnecke et al. [69] studied the influence of window size and proposed a partitioning method for splitting large windows into different batches, considering both time and space efficiency. SABER [45] is a window-based hybrid stream processing framework aiming to utilize CPUs and GPUs concurrently.

Multi-GPU. Multi-GPU systems provide tremendous computation capacity, but also pose challenges like how to partition or schedule workloads among GPUs. Verner et al. [89] extend their method [88] to a single node with multiple GPUs. A scheduler controls stream placement and guarantees that the requirements among different streams can be met. GStream [106] is the first data streaming framework for GPU clusters. GStream supports stream processing applications in the form of a C++ library; it uses MPI to implement the data communication between different nodes and uses CUDA to conduct stream operations on GPUs. Alghabi et al. [10] first introduced the concept of stateful stream data processing on a node with multiple GPUs. Nguyen et al. [63] considered the scalability with the number of GPUs on a single node, and developed a GPU performance model for stream workload partitioning in multi-GPU platforms with high scalability. Chen et al. [20] proposed G-Storm, which enables Storm [6] to utilize GPUs and can be applied to various applications that Storm has already supported.

6.3 FPGA-Enabled Stream Processing

FPGAs are programmable integrated circuits whose hardware interconnections can be configured by users. Due to their low latency, high energy efficiency, and low hardware engineering cost, FPGAs have been explored in various application scenarios, including stream processing.

Hagiescu et al. [30] first elaborated challenges to implementing stream processing on FPGAs and proposed algorithms that optimize processing throughput and latency for FPGAs. Mueller et al. [60] provided *Glacier*, which is an FPGA-based query engine that can process queries on streaming data from networks. The operations in *Glacier* include selection, aggregation, grouping, and windows. Experiments show that using FPGAs helps achieve much better performance than using conventional CPUs. A common limitation of an FPGA-based system is its expensive synthesis process, which takes a significant time to compile the application into hardware designs for FPGAs. This makes FPGA-based systems inflexible in adapting to query changes. In response, Najafi et al. [61] demonstrated Flexible Query Processor (FQP), an online reconfigurable event stream query processor that can accept new queries without disrupting other queries in execution.

6.4 Remarks

Existing systems usually involve heterogeneous processors along with CPUs. Such heterogeneity opens up both new opportunities and poses challenges for scaling stream processing. From the above discussion, it is clear that both GPUs and FPGAs have been successfully applied for scaling up stream processing. FPGAs have low latency and are hardware configurable. Hence, they are suitable for special application scenarios, such as a streaming network.

7 System Design Requirements

In 2005, Stonebraker et al. [78] outlines eight requirements of real-time data stream processing. Since then, tremendous improvements have been made thanks to the great efforts from both industry and the research community. We now summarize how hardware-conscious optimization techniques mitigate the gap between DSPSs and requirements while highlighting the insufficiency.

Most DSPSs are designed with the principle of "*Keep the Data Moving*" [78], and hence aim to process input data "on-the-fly" without storing them. As a result, message passing is often a key component in the current DSPSs. To mitigate the overhead, researchers have recently attempted to improve the cross-operator communication efficiency by taking advantage of the latest advancement in network infrastructure [38], compression using hardware accelerator [66], and efficient algorithms by exploiting new hardware characteristics [58]. Going forward, we expect more work to be done for hardware-conscious stream I/O optimization.

Handling out-of-order input streams is relevant to both the *Handle Stream Imperfections* and *Generate Predictable Outcomes* [78] requirements. In real-time stream systems where the input data are not stored, the infrastructure must make provision for handling data that arrive late or are delayed, missing or out-of-sequence. Correctness can be guaranteed in some applications only if time-ordered and deterministic processing is maintained throughout the entire processing pipeline. Despite the significant efforts, existing DSPSs are still *far from ideal* for exploiting the potential of modern hardware. For example, as observed in a recent work [104], the same DSPS (i.e., StreamBox) delivers much lower throughput on modern multicore processors as a result of enabling ordering guarantees.

The state management in DSPSs is more

related to the *Integrate Stored and Streaming Data* [78] requirement. For many stream processing applications, comparing the "present" with the "past" is a common task. Thus, the system must provide careful management of the stored states. However, we observe that only a few related studies attempt to improve state management efficiency levering modern hardware [51]. There are still many open questions to be resolved, such as new storage formats, indexing techniques for emerging hardware architectures and applications [29, 101]. New media applications such as live audio streaming services [91] also challenge existing systems in terms of new processing paradigms.

The *Partition and Scale Applications Automatically* [78] requires a DSPS to be able to elastically scale up and down in order to process input streams with varying characteristics. However, based on our analysis, little work has considered scaling *down* the processing efficiently (and easily scaling up later) in a hardware-aware manner. A potential direction is adopting a serverless computing paradigm [13] with the help of novel memory techniques such as Non-Volatile Memory (NVM) into DSPSs. However, questions such as how to efficiently manage the partial computing state in GPUs or FPGAs still remain unclear.

The proliferation of high-rate data sources has accelerated the development of next-generation performance-critical DSPSs. For example, the new 5G network promises blazing speeds, massive throughput capability, and ultra-low latencies [2], thus bringing the higher potential for performance critical stream applications. In response, high-throughput stream processing is essential to keeping up with data streams in order to satisfy the *Process and Respond Instantaneously* [78] requirement. However, achieving high-throughput stream processing is challenging, especially when expensive windowing operations are deployed. By better utilizing modern hardware, researchers and practitioners have achieved promising results. For example, SABER processes 79 million tuples per second with eight CPU cores for Yahoo Streaming Benchmark, outperforming other DSPSs several times [7]. Nevertheless, current results also show that there is still room for improvement on a single node, and this constitutes an opportunity for designing the next-generation DSPSs [99].

Two requirements including *Query using SQL on Streams* and *Guarantee Data Safety and Availability* are overlooked by most existing HW-conscious optimization techniques in DSPSs. In particular, how to design HW-aware SQL statements for DSPSs, and how best to guarantee data safety and system availability when adopting modern hardware, such as NVM for efficient local backup and high-speed network for remote backup, remain an open question.

8 Conclusion

In this paper, we have discussed relevant literature from the field of hardware-conscious DSPSs, which aim to utilize modern hardware capabilities for accelerating stream processing. Those works have significantly improved DSPSs to better satisfy the design requirements raised by Stonebraker et al. [78]. In the following, we list some additional advice on future research directions.

Scale-up and -out Stream Processing. As emphasized by Gibbons [25], scaling both out and up is crucial to effectively improving the system performance. In situ analytics enable data processing at the point of data origin, thus reducing the data movements across networks; Powerful hardware infrastructure provides an opportunity to improve processing performance within a single node. To this end, many recent works have exploited the potential of high-performance stream processing on a single node [45, 57, 98]. However, the important question of how best to use powerful local nodes in the context of large distributed computation setting still remains unclear.

Stream Processing Processor. With the wide adoption of stream processing today, it may be a good time to revisit the design of a specific processor for DSPSs. GPUs [45] provide much higher bandwidth than CPUs, but it comes with larger latency as tuples must be first accumulated in order to fully utilize thousands of cores on GPU; FPGA [47] has its advantage in providing low latency, low power consumption computation but its throughput is still much lower compared to GPUs. The requirement for an ideal processor for stream processing includes *low latency*, *low power consumption*, and *high bandwidth*. On the other hand, components like complex control logic may be sacrificed as stream processing logic is usually predefined and fixed. Further, due to the nature of continuous query processing, it is ideal to keep the entire instruction set close to processor [103].

Acknowledgments. The authors would like to thank the anonymous reviewer and the associate editor, Pınar Tözün, for their insightful comments on improving this manuscript. This work is supported by a MoE Tier 1 grant (T1 251RES1824) and a MoE Tier 2 grant (MOE2017-T2-1-122) in Singapore. Feng Zhang's work was partially supported by the National Natural Science Foundation of China (Grant No. 61802412, 61732014).

9 References

[1] Corsair force series nvme ssd. `https://hothardware.com/news/corsair-mp600`.

[2] Ericsson mobility report november 2018, `https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-november-2018.pdf`.

[3] Number of connected iot devices will surge to 125 billion by 2030, ihs markit says. `https://en.ctimes.com.tw/DispNews.asp?O=HK1APAPZ546SAA00N9`.

[4] Sgi uv 300, `https://www.earlham.ac.uk/sgi-uv300`, 2015.

[5] Apache flink, `https://flink.apache.org/`, 2018.

[6] Apache storm, `http://storm.apache.org/`, 2018.

[7] Do we need distributed stream processing? `https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing`, 2019.

[8] K. Agrawal and et al. Cache-conscious scheduling of streaming applications. In *SPAA*, 2012.

[9] T. Akidau and et al. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 2013.

[10] F. Alghabi and et al. A scalable software framework for stateful stream data processing on multiple gpus and applications. In *GPU Computing and Applications*. 2015.

[11] A. Arasu and et al. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 2006.

[12] S. Babu and et al. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 2004.

[13] I. Baldini and et al. Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*, 2017.

[14] P. Carbone and et al. Cutty: Aggregate sharing for user-defined windows. In *CIKM*, 2016.

[15] P. Carbone and et al. State management in apache flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 2017.

[16] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES*, 2009.

[17] R. Castro Fernandez and et al. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *SIGMOD*, 2013.

[18] C. Chambers and et al. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[19] B. Chandramouli and et al. Trill: A high-performance incremental query processor for diverse analytics. Proc. VLDB Endow., Aug. 2015.

[20] Z. Chen and et al. G-Storm: GPU-enabled high-throughput online data processing in Storm. In *Big Data*, 2015.

[21] J. A. Colmenares and et al. Ingestion, indexing and retrieval of high-velocity multidimensional sensor data on a single node. volume abs/1707.00825, 2017.

[22] S. M. Farhad and et al. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *ASPLOS*, 2011.

[23] B. Gedik and et al. Celljoin: A parallel stream join operator for the cell processor. *The VLDB Journal*, 2009.

[24] P. R. Geethakumari and et al. Single window stream aggregation using reconfigurable hardware. In *ICFPT*, 2017.

[25] P. B. Gibbons. Big data: Scale down, scale up, scale out. In *IPDPS*, 2015.

[26] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. VLDB Endow.*, 2003.

[27] S. Gong and et al. Shuntflow: An efficient and scalable dataflow accelerator architecture for streaming applications. In *ADAC*, 2019.

[28] V. Gulisano and et al. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. In *Big Data*, 2015.

[29] P. Götze and et al. Query planning for transactional stream processing on heterogeneous hardware: Opportunities and limitations. In *BTW 2019*, 2019.

[30] A. Hagiescu and et al. A computing origami: folding streams in FPGAs. In *DAC*, 2009.

[31] M. A. Hammad and et al. Stream window join: tracking moving objects in sensor-network databases. In *SSDM*, 2003.

[32] J. He and et al. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 2013.

[33] T. Heinze and et al. Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems. *TPDS*, 2014.

[34] M. Hirzel and et al. Spreadsheets for stream processing with unbounded windows and partitions. In *DEBS*, 2016.

[35] M. Hirzel and et al. Sliding-window aggregation algorithms: Tutorial. In *DEBS*, 2017.

[36] A. H. Hormati and et al. Macross: Macro-simdization of streaming applications. In *ASPLOS*, 2010.

[37] N. Jain and et al. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *SIGMOD*, 2006.

[38] S. Kamburugamuve and et al. Low latency stream processing: Apache heron with infiniband & intel omni-path. In *UCC*, 2017.

[39] J. Kang and et al. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[40] K. Kanoun and et al. Low power and scalable many-core architecture for big-data stream computing. In *VLSI*, 2014.

[41] T. Karnagel and et al. The hells-join: A heterogeneous stream join for extremely large windows. In *DaMoN*, 2013.

[42] T. Karnagel and et al. Stream join processing on heterogeneous processors. In *BTW Workshops*, 2013.

[43] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *Proc. VLDB Endow.*, 2017.

[44] I. Kolchinsky and A. Schuster. Join query optimization techniques for complex event processing applications. *Proc. VLDB Endow.*, 2018.

[45] A. Koliousis and et al. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, 2016.

[46] S. Krishnamurthy and et al. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[47] C. Kritikakis and et al. An fpga-based high-throughput stream join architecture. In *FPL*, Aug. 2016.

[48] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 2008.

[49] S. Kulkarni and et al. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[50] I. E. Kuralenok and et al. An optimistic approach to handle out-of-order events within analytical stream processing. In *CEUR Workshop*, 2018.

[51] G. Lee and et al. High-performance stateful stream processing on solid-state drives. In *APSys*, 2018.

[52] J. Li and et a. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 2005.

[53] J. Li and et al. Out-of-order processing: A new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 2008.

[54] Q. Lin and et al. Scalable distributed stream join processing. In *SIGMOD*, 2015.

[55] N. Marz. Trident API Overview. `github.com/nathanmarz/storm/wiki/Trident-APIOverview`.

[56] J. Meehan and et al. S-store: streaming meets transaction processing. *Proc. VLDB Endow.*, 2015.

[57] H. Miao and et al. Streambox: Modern stream processing on a multicore machine. In *USENIX ATC*, 2017.

[58] H. Miao and et al. Streambox-hbm: Stream analytics on high bandwidth hybrid memory. *arXiv preprint arXiv:1901.01328*, 2019.

[59] R. Mueller and et al. Data processing on fpgas. *Proc. VLDB Endow.*, 2009.

[60] R. Mueller and et al. Streams on wires: a query compiler for FPGAs. *Proc. VLDB Endow.*, 2009.

[61] M. Najafi and et al. Flexible query processor on FPGAs. *Proc. VLDB Endow.*, 2013.

[62] M. Najafi and et al. Splitjoin: A scalable, low-latency stream join architecture with adjustable ordering precision. In *USENIX ATC*, 2016.

[63] D. Nguyen and J. Lee. Communication-aware mapping of stream graphs for multi-gpu platforms. In *CGO*, 2016.

[64] Y. Oge, M. Yoshimi, and et al. An efficient and scalable implementation of sliding-window aggregate operator on fpga. In *CANDAR*, 2013.

[65] F. Pan and H. Jacobsen. Panjoin: A partition-based adaptive stream join. *CoRR*, abs/1811.05065, 2018.

[66] G. Pekhimenko and et al. Tersecades: Efficient data compression in stream processing. In *USENIX ATC*, 2018.

[67] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[68] G. Philipp and et al. An nvm-aware storage layout for analytical workloads. In *ICDEW*, 2018.

[69] M. Pinnecke and et al. Toward GPU Accelerated Data Stream Processing. In *GvD*, 2015.

[70] C. Pohl. Stream processing on high-bandwidth memory. In *Grundlagen von Datenbanken*, 2018.

[71] L. Rabiner and et al. Applications of a nonlinear smoothing algorithm to speech processing. *TASSP*, 1975.

[72] C. Rousopoulos and et al. A generic high throughput architecture for stream processing. In *FPL*, 2017.

[73] P. Roy and et al. Low-latency handshake join. *Proc. VLDB Endow.*, May 2014.

[74] E. Ryvkina and et al. Revision processing in a stream processing engine: A high-level design. In *ICDE*, 2006.

[75] A. Shahvarani and H.-A. Jacobsen. Parallel index-based stream join on a multicore cpu. `https://arxiv.org/pdf/1903.00452.pdf`, 2019.

[76] J. Shi and et al. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *OSDI*, 2016.

[77] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. VLDB Endow.*, 2004.

[78] M. Stonebraker and et al. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 2005.

[79] Y. Tang and B. Gedik. Autopipelining for data stream processing. *TPDS*, 2013.

[80] K. Tangwongsan and et al. General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 2015.

[81] K. Tangwongsan and et al. *Sliding-Window Aggregation Algorithms*. Springer International Publishing, 2018.

[82] W. B. Teeuw and H. M. Blanken. Control versus data flow in parallel database machines. *TPDS*, 1993.

[83] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD*, 2011.

[84] G. Theodorakis and et al. Hammer slide: work-and cpu-efficient streaming window aggregation. In *ADMS*, 2018.

[85] Q.-C. To and et al. A survey of state management in big data processing systems. *The VLDB Journal*, 2018.

[86] J. Traub and et al. Efficient window aggregation with general stream slicing. In *EDBT*, pages 97–108, 2019.

[87] S. Venkataraman and et al. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[88] U. Verner and et al. Processing data streams with hard real-time constraints on heterogeneous systems. In *ICS*, 2011.

[89] U. Verner and et al. Scheduling processing of real-time data streams on heterogeneous multi-gpu systems. In *SYSTOR*, 2012.

[90] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.

[91] Z. Wen and et al. Rtsi: An index structure for multi-modal real-time search on live audio streaming services. In *ICDE*, 2018.

[92] Y. Wu and K. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *ICDE*, Apr. 2015.

[93] J. Xie and J. Yang. A survey of join processing in data streams. *Data Streams: Models and Algorithms*, 2007.

[94] J. Xu and et al. T-storm: Traffic-aware online scheduling in storm. In *ICDCS*, 2014.

[95] Y. Ya-xin and et al. An indexed non-equijoin algorithm based on sliding windows over data streams. *Wuhan University Journal of Natural Sciences*, 2006.

[96] M. Zaharia and et al. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[97] E. Zamanian and et al. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 2017.

[98] S. Zeuch and et al. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, 2019.

[99] S. Zeuch and et al.l. The nebulastream platform: Data and application management for the internet of things. In *CIDR*, 2020.

[100] K. Zhang and et al. A holistic approach to build real-time stream processing system with gpu. *JPDC*, 2015.

[101] S. Zhang and et al. Towards concurrent stateful stream processing on multicore processors, `https://arxiv.org/abs/1904.03800`.

[102] S. Zhang and et al. Multi-query optimization for complex event processing in sap esp. In *ICDE*, 2017.

[103] S. Zhang and et al. Revisiting the design of data stream processing systems on multi-core processors. In *ICDE*, 2017.

[104] S. Zhang and et al. Briskstream: Scaling Data Stream Processing on Multicore Architectures. In *SIGMOD*, 2019.

[105] Y. Zhang and et al. Sub-millisecond stateful stream querying over fast-evolving linked data. In *SOSP*, 2017.

[106] Y. Zhang and F. Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. In *ICPP*, 2011.