SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures

Alexandros Koliousis[†], Matthias Weidlich[†‡], Raul Castro Fernandez[†], Alexander L. Wolf[†], Paolo Costa[♯], Peter Pietzuch[†]

[†]Imperial College London [‡]Humboldt-Universität zu Berlin [♯]Microsoft Research

{akolious, mweidlic, rc3011, alw, costa, prp}@imperial.ac.uk

ABSTRACT

Modern servers have become heterogeneous, often combining multi-core CPUs with many-core GPGPUs. Such heterogeneous architectures have the potential to improve the performance of data-intensive stream processing applications, but they are not supported by current relational stream processing engines. For an engine to exploit a heterogeneous architecture, it must execute streaming SQL queries with sufficient data-parallelism to fully utilise all available heterogeneous processors, and decide how to use each in the most effective way. It must do this while respecting the semantics of streaming SQL queries, in particular with regard to window handling.

We describe SABER, a *hybrid* high-performance relational stream processing engine for CPUs and GPGPUs. SABER executes window-based streaming SQL queries in a data-parallel fashion using all available CPU and GPGPU cores. Instead of statically assigning query operators to heterogeneous processors, SABER employs a new adaptive *heterogeneous lookahead scheduling* strategy, which increases the share of queries executing on the processor that yields the highest performance. To hide data movement costs, SABER pipelines the transfer of stream data between CPU and GPGPU memory. Our experimental comparison against state-of-the-art engines shows that SABER increases processing throughput while maintaining low latency for a wide range of streaming SQL queries with both small and large window sizes.

1. INTRODUCTION

The recent generation of stream processing engines, such as S4 [45], Storm [52], Samza [1], Infosphere Streams [13], Spark Streaming [56] and SEEP [17], executes data-intensive streaming queries for real-time analytics and event-pattern detection over high volumes of stream data. These engines achieve high processing throughput through *data parallelism*: streams are partitioned so that multiple instances of the same query operator can process *batches* of the stream data in parallel. To have access to sufficient parallelism during execution, these engines are deployed on shared-nothing clusters, scaling out processing across servers.

An increasingly viable alternative for improving the performance of stream processing is to exploit the parallel processing capability offered by *heterogeneous* servers: besides multi-core CPUs with dozens of processing cores, such servers also contain accelerators such as GPGPUs with thousands of cores. As heterogeneous servers have become commonplace in modern data centres and cloud offerings [24], researchers have proposed a range of parallel streaming algorithms for heterogeneous architectures, including for joins [36,51], sorting [28] and sequence alignment [42]. Yet, we lack the design of a *general-purpose* relational stream processing engine that can transparently take advantage of accelerators such as GPGPUs while executing arbitrary streaming SQL queries with windowing [7].

The design of such a streaming engine for heterogeneous architectures raises three challenges, addressed in this paper:

(1) When to use GPGPUs for streaming SQL queries? The speed-up that GPGPUs achieve depends on the type of computation: executing a streaming operator over a batch of data in parallel can greatly benefit from the higher degree of parallelism of a GPGPU, but this is only the case when the data accesses fit well with a GPGPU's memory model. As we show experimentally in §6, some streaming operators accelerate well on GPGPUs, while others exhibit lower performance than on a CPU. A streaming engine with GPGPU support must therefore make the right scheduling decisions as to which streaming SQL queries to execute on the GPGPU in order to achieve high throughput for arbitrary queries.

(2) How to use GPGPUs with streaming window semantics? While a GPGPU can naturally process a discrete amount of data in parallel, streaming SQL queries require efficient support for *sliding windows* [7]. Existing data-parallel engines such as Spark Streaming [56], however, tie the size of physical batches of stream data, used for parallel processing, to that of logical windows specified in the queries. This means that the window size and slide of a query impact processing throughput; current data-parallel engines, for example, struggle to support small window sizes and slides efficiently. A streaming engine with GPGPU support must instead support arbitrary windows in queries while exploiting the most effective level of parallelism without manual configuration by the user.

(3) How to reduce the cost of data movement with GPGPUs? In many cases, the speed-up achieved by GPGPUs is bounded by the data movement cost over the PCI express (PCIe) bus. Therefore, a challenge is to ensure that the data is moved to the GPGPU in a continuous fashion so that the GPGPU is never idle.

We describe the design and implementation of SABER, a **hybrid relational stream processing engine** in Java that executes streaming SQL queries on both a multi-core CPU and a many-core GPGPU to achieve high processing throughput. The main idea behind SABER is that, instead of offloading a fixed set of query operators to an accelerator, it adopts a *hybrid* execution model in which all query operators can utilise all *heterogeneous processors* in-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

 ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882906

terchangeably. By "processor" we refer here to either an individual CPU core or an entire GPGPU. More specifically, SABER makes the following technical contributions:

Hybrid stream processing model. SABER takes a streaming SQL query and translates it to an operator graph. The operator graph is bundled with a batch of stream data to form a *query task* that can be scheduled on a heterogeneous processor. SABER's hybrid stream processing model then features two levels of data parallelism: (i) tasks run in parallel across the CPU's multiple cores and the GPGPU and (ii) a task running on the GPGPU is further parallelised across its many cores.

One approach for selecting the processor on which to run a given query operator is to build an offline performance model that predicts the performance of operators on each processor type. Accurate performance models, however, are hard to obtain, especially when the processing performance can change dynamically, e.g. due to skewed distributions of input data.

In contrast, SABER uses a new *heterogeneous lookahead scheduling (HLS)* algorithm: the scheduler tries to assign each task to the heterogeneous processor that, based on past behaviour, achieves the highest throughput for that task. If that processor is currently unavailable, the scheduler instead assigns a task to another processor with lower throughput that still yields an earlier completion time. HLS thus avoids the need for a complex offline model and accounts for changes in the runtime performance of tasks.

Window-aware task processing. When SABER executes query tasks on different heterogeneous processors, it supports sliding window semantics and maintains high throughput for small window sizes and slides. A *dispatcher* splits the stream into fixed-sized batches that include multiple fragments of windows processed jointly by a task. The computation of window boundaries is postponed until the task executes, thus shifting the cost from the sequential dispatching stage to the highly parallel task execution stage. For sliding windows, tasks perform incremental computation on the windows in a batch to avoid redundant computation.

SABER preserves the order of the result stream after the parallel, out-of-order processing of tasks by first storing the results in local buffers and then releasing them incrementally in the correct order as tasks finish execution.

Pipelined stream data movement. To avoid delays due to data movement, SABER introduces a *five-stage pipelining* mechanism that interleaves data movement and task execution on the GPGPU: the first three stages are used to maintain high utilisation of the PCIe bandwidth by pipelining the Direct Memory Access (DMA) transfers of batches to and from the GPGPU with the execution of associated tasks; the other two stages hide the memory latency associated with copying batches in and out of managed heap memory.

The remainder of the paper is organised as follows: §2 motivates the need for high-throughput stream processing and describes the features of heterogeneous architectures; §3 introduces our hybrid stream processing model; §4 describes the SABER architecture, focussing on its dispatching and scheduling mechanisms; and §5 explains how SABER executes query operators. The paper finishes with evaluation results (§6), related work (§7) and conclusions (§8).

2. BACKGROUND

2.1 High-throughput stream processing

Stream processing has witnessed an uptake in many application domains, including credit fraud detection [26], urban traffic management [9], click stream analytics [5], and data centre management [44]. In these domains, continuous streams of input data (e.g. transaction events or sensor readings) need to be processed in an online manner. The key challenge is to maximise processing *throughput* while staying within acceptable latency bounds.

For example, already in 2011, Facebook reported that a query for click stream analytics had to be evaluated over input streams of 9 GB/s, with a latency of a few seconds [49]. Credit card fraud detection systems must process up to 40,000 transactions per second and detect fraudulent activity within 25 ms to implement effective countermeasures such as blocking transactions [26]. In finance, the NovaSparks engine processes a stream of cash equity options at 150 million messages per second with sub-microsecond latency [46].

The queries for the above use cases can be expressed in a streaming relational model [7]. A stream is a potentially infinite sequence of relational *tuples*, and queries over a stream are window-based. A *window* identifies a finite subsequence of a stream, based on tuple count or time, which can be viewed as a relation. Windows may overlap—tuples of one window may also be part of subsequent windows in which case the window *slides* over the stream. Distinct subsequences of tuples that are logically related to the same set of windows are referred to as *panes* [41], i.e. a window is a concatenation of panes. Streaming relational query *operators*, such as projection, selection, aggregation and join, are executed per window.

To execute streaming relational queries with high throughput, existing stream processing engines exploit parallelism: they either execute different queries in parallel (*task parallelism*), or execute operators of a single query in parallel on different partitions of the input stream (*data parallelism*). The need to run expensive analytics queries over high-rate input streams has focused recent research efforts on data parallelism.

Current data-parallel stream processing engines such as Storm [52], Spark [56] and SEEP [17] process streams in a data-parallel fashion according to a *scale-out* model, distributing operator instances across nodes in a shared-nothing cluster. While this approach can achieve high processing throughput, it suffers from drawbacks: (i) it assumes the availability of compute clusters; (ii) it requires a high-bandwidth network that can transfer streams between nodes; (iii) it can only approximate time-based window semantics due to the lack of a global clock in a distributed environment; and (iv) the overlap between windows requires redundant data to be sent to multiple nodes or the complexity of a pane-based approach [11].

2.2 Heterogeneous architectures

In contrast to distributed cluster deployments, we explore a different source of parallelism for stream processing that is becoming increasingly available in data centres and cloud services: servers with *heterogeneous architectures* [10, 54]. These servers, featuring multi-core CPUs and accelerators such as GPGPUs, offer unprecedented degrees of parallelism within a single machine.

GPGPUs implement a *throughput-oriented architecture*. Unlike traditional CPUs, which focus on minimising the runtime of sequential programs using out-of-order execution, speculative execution and deep memory cache hierarchies, GPGPUs target embarrassingly parallel workloads, maximising total system throughput rather than individual core performance. This leads to fundamental architectural differences—while CPUs have dozens of sophisticated, powerful cores, GPGPUs feature thousands of simple cores, following the *single-instruction, multiple-data* (SIMD) model: in each cycle, a GPGPU core executes the same operation (referred to as a *kernel*) on different data. This simplifies the control logic and allows more functional units per chip: e.g. the NVIDIA Quadro K5200 has 2,304 cores and supports tens of thousands of threads [47].

GPGPU cores are grouped into streaming multi-processors (SM), with each SM supporting thousands of threads. Threads have their

own dedicated sets of registers and access to high-bandwidth, shared on-chip memory. While modern CPUs hide latency by using a deep cache hierarchy, aggressive hardware multi-threading in GPGPUs allows latency to be hidden by interleaving threads at the instruction-level with near zero overhead. Therefore, GPGPU cores have only small cache sizes compared to CPUs: e.g. the NVDIA Quadro K5200 has a 64 KB L1 cache per SM and a 1,536 KB L2 cache shared by all SMs.

A dedicated accelerator typically is attached to a CPU via a PCIe bus, which has limited bandwidth.[1] For example, a PCIe 3.0 $\times 16$ bus has an effective bandwidth of approximately 8 GB/s. This introduces a fundamental throughput limit when moving data between CPU and GPGPU memory. Especially when data movements occur at a high frequency, as in streaming scenarios, care must be taken to ensure that these transfers happen in a pipelined fashion to avoid stalling the GPGPU.

The potential of GPGPUs for accelerating queries has been shown for traditional relational database engines. For example, Ocelot [32] is an in-memory columnar database engine that statically offloads queries to a GPGPU using a single "hardware-oblivious" representation. GDB [29] and HyPE [16] accelerate SQL queries by offloading data-parallel operations to GPGPUs. All these systems, however, target one-off queries instead of continuous streaming queries with window semantics.

2.3 Using GPGPUs for stream processing

The dataflow model of GPGPUs purportedly is a good match for stream processing. In particular, performing computation on a group of tuples that is laid out in memory linearly should greatly benefit from the SIMD parallelism of GPGPUs. At the same time, the scarcity of designs for relational stream processing engines on GPGPUs is evidence of a range of challenges:

Different acceleration potential. Not all stream processing queries experience a speed-up when executed on a GPGPU compared to a CPU. Any speed-up depends on a range of factors, including the amount of computation per stream tuple, the size of the tuples, any memory accesses to temporary state and the produced result size. For example, a simple projection operator that removes attributes from tuples typically is bottlenecked by memory accesses, whereas an aggregation operator with a complex aggregation function is likely to be compute-bound. This makes it challenging to predict if the performance of a given query can be improved by a GPGPU.

Existing proposals for accelerating traditional database queries (e.g. [37, 55]) assume that a comprehensive analytical model can establish the relation between a query workload and its performance on a heterogeneous processor. Based on such a performance model, they offload queries on a GPGPU either statically [32] or speculatively [29]. Fundamentally, performance models are vulnerable to inaccuracy, which means that they may not make the most effective use of the available accelerators, given an unknown query workload.

The challenge is to devise an approach that makes effective decisions as to which queries can benefit from GPGPU offloading.

Support for sliding windows. Any data-parallel stream processing must split streams into *batches* for concurrent processing, but each streaming SQL query also includes a window specification that affects its result. The batch size should thus be independent from the window specification—the batch size is a physical implementation parameter that determines the efficiency of parallel processing, whereas the window size and slide are part of the query semantics.

[1] Despite recent advances in integrated CPU/GPGPU designs [22, 30, 31], discrete GPGPUs have a substantial advantage in terms of performance and power efficiency.

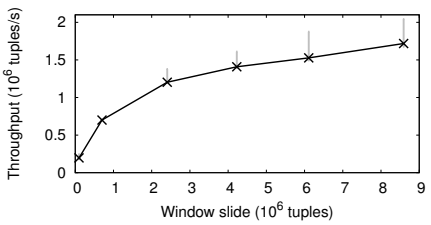


Figure 1: Performance of a streaming GROUP-BY query with a 5-second window and different window slides under Spark Streaming

Existing data-parallel engines, however, tie the definition of batches to that of windows because this simplifies window-based processing. For example, Spark Streaming [56] requires the window slide and the batch size to be a multiple of the window size, enabling parallel window computation to occur on different nodes in lockstep.

Such a coupling of batches and windows has a performance impact. In Fig. 1, we show the throughput of a streaming GROUP-BY query with a 5-second sliding window executing on Spark Streaming with a changing window slide. As the slide becomes smaller (i.e. there is more overlap between windows), the throughput decreases—when the batches are smaller, the parallel window processing becomes less efficient. This makes it infeasible to process queries with fewer than several million 64-byte tuples in each window slide while sustaining high throughput.

The challenge is how an engine can decouple batches and windows from each other and still ensure a consistently high throughput.

Cost of stream data movement. Since a stream processing engine processes continuous data, the bandwidth of the PCIe bus can limit the throughput of queries offloaded to the GPGPU. In particular, any data movement to and from the GPGPU causes a delay—a DMA-managed memory transfer from the CPU to the GPGPU takes around 10 microseconds [43], which is orders of magnitude slower than a CPU memory access.

The challenge is how an engine can amortise the delay of data movement to and from the GPGPU while processing data streams.

2.4 Stream data and query model

We assume a relational stream data model with window-based queries, similar to the *continuous query language* (CQL) [7].

Data streams. Let T be a universe of *tuples*, each being a sequence of values of primitive data types. A *stream* $S = \langle t_1, t_2, \ldots \rangle$ is an infinite sequence of such tuples, and T^* denotes the set of all streams. A tuple t has a timestamp, $\tau(t) \in \mathscr{T}$, and the order of tuples in a stream respects these timestamps, i.e. for two tuples t_i and t_j, $i < j$ implies that $\tau(t_i) \leq \tau(t_j)$. We assume a discrete, ordered time domain $\mathscr{T}$ for timestamps, given as the non-negative integers $\{0, 1, \ldots\}$. Timestamps refer to logical application time: they record when a given event occurred and not when it arrived in the system. There is only a finite number of tuples with equal timestamps.

Window-based queries. We consider queries that are based on *windows*, which are finite sequences of tuples. The set of all windows is $\hat{T}^* \subset T^*$, and the set of all, potentially infinite, sequences of windows is $(\hat{T}^*)^*$. A window-based query q is defined for n input streams by three components: (i) an n-tuple of *window functions* $\langle \omega_1^q, \ldots, \omega_n^q \rangle$, such that each function $\omega_i^q : T^* \to (\hat{T}^*)^*$, when evaluated over an input stream, yields a possibly infinite sequence of windows; (ii) a possibly compound n-ary *operator function* $f^q : (\hat{T}^*)^n \to \hat{T}^*$ that is applied to n windows, one per input stream, and produces a window result (a sequence of tuples); and (iii) a *stream function* $\phi^q : (\hat{T}^*)^* \to T^*$ that transforms a sequence of window results into a stream.

Common window functions define *count-* or *time-based* windows with a *window size s* and a *window slide l*: the window size determines the amount of enclosed data; the window slide controls the difference to subsequent windows. Let $S = \langle t_1, t_2, \ldots \rangle$ be a stream, and $\omega_{(s,l)}$ be a window function with size s and slide l that creates a sequence of windows $\omega_{(s,l)}(S) = \langle w_1, w_2, \ldots \rangle$. Given a window $w_i = \langle t_k, \ldots, t_m \rangle, i \in \mathbb{N}$, if ω is count-based, then $m = k + s - 1$ and the next window is $w_{i+1} = \langle t_{k+l}, \ldots, t_{m+l} \rangle$; if ω is time-based, then the next window is $w_{i+1} = \langle t_{k'}, \ldots, t_{m'} \rangle$ such that $\tau(t_m) - \tau(t_k) \leq s$, $\tau(t_{m+1}) - \tau(t_k) > s$, $\tau(t_{k'}) - \tau(t_k) \leq l$, and $\tau(t_{k'+1}) - \tau(t_k) > l$. This model thus supports *sliding* ($l < s$) and *tumbling* windows ($l = s$). We do not consider predicate-based windows in this work.

Query operators. The above model allows for the flexible definition of an operator function that transforms a window into a window result, both being sequences of tuples. In this work, we support relational streaming operators [7], i.e. projection (π), selection (σ), aggregation (α) and θ-join ($\bowtie$). These operators interpret a window as a relation (i.e. the sequence is interpreted as a multi-set) over which they are evaluated. For the aggregation operator, we also consider the use of `GROUP-BY` clauses.

Operator functions may also be specified as *user-defined functions* (UDFs), which implement bespoke computation per window. An example of a UDF is an n-ary partition join, which takes as input an n-tuple of windows, one per input stream, and first partitions all windows based on tuple values before joining the corresponding partitions of the windows. Despite its similarity, a partition join cannot be realised with a standard θ-join operator.

Stream creation. Window results are turned into a stream by a stream function. We consider stream functions that are based on relation-to-stream operators [7]. The *RStream* function concatenates the results of windows under the assumption that this order respects the tuple timestamps: with $\langle w_1, w_2, \ldots \rangle$ as a sequence of window results, and $w_i = \langle t_{k_i}, \ldots, t_{m_i} \rangle$ for $i \in \mathbb{N}$, the result stream is a sequence $S_R = \langle t_{k_1}, \ldots, t_{m_1}, t_{k_2}, \ldots, t_{m_2}, \ldots \rangle$. The *IStream* function considers only the tuples of a window result that were not part of the previous result: with $\langle \ldots, t_{j_{i+1}}, \ldots \rangle$ as the projection of w_{i+1} to all transitions $t_{j_{i+1}} \notin \{t_{k_i}, \ldots, t_{m_i}\}$ that are not part of w_i, the result stream is $S_R = \langle t_{k_1}, \ldots, t_{m_1}, \ldots, t_{j_2}, \ldots, t_{j_3}, \ldots \rangle$.

Typically, specific operator functions are combined with particular stream functions [7]: using the IStream function with projection and selection operators gives intuitive semantics, whereas aggregation and θ-join operators are used with the RStream function. In the remainder, we assume these default combinations.

3. HYBRID STREAM PROCESSING MODEL

Our idea is to sidestep the problem of "when" and "what" operators to offload to a heterogeneous processor by introducing a *hybrid stream processing model*. In this hybrid model, the stream processing engine always tries to utilise *all* available heterogeneous processors for query execution opportunistically, thus achieving the aggregate throughput. In return, the engine does not have to make an early decision regarding which type of query to execute on a given heterogeneous processor, be it a CPU core or a GPGPU.

Query tasks. The hybrid stream processing model requires that each query can be scheduled on any heterogeneous processor, leaving the scheduling decision until runtime. To achieve this, queries are executed as a set of data-parallel *query tasks* that are runnable on either a CPU core or the GPGPU. For a query q with n input streams, a query task $v = (f^q, B)$ consists of (i) the n-ary operator function f^q specified in the query and (ii) a sequence of n *stream batches* $B = \langle b_1, \ldots, b_n \rangle$, $b_i \in \hat{T}^*$, i.e. n finite sequences of tuples, one per input stream. We use $\iota(v) = q$ as a shorthand notation to refer to the query of task v (§4.2, Alg. 1).

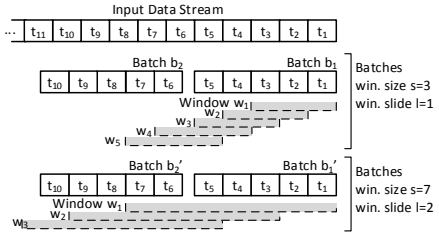


Figure 2: Stream batches under two different window definitions

The *query task size* φ is a system parameter that specifies how much stream data a query task must process, thus determining the computational cost of executing a task. Let $v = (f^q, B)$ with $B = \langle b_1, \ldots, b_n \rangle$ be a query task, then the query task size is the sum $\Sigma_{i=1}^{n} |b_i|$ of the *stream batch sizes*. The size of a stream batch $b_i = \langle t_1, \ldots, t_m \rangle$ is the data volume of its m tuples. The query task size equals the stream batch size in the case of single-input queries.

As we show experimentally in §6.4, a fixed query task size can in practice be chosen independently of the query workload, simply based on the properties of the stream processing engine implementation and its underlying hardware. There is a trade-off when selecting the query task size φ: a large size leads to higher throughput because it amortises scheduling overhead, exhibits more parallelism on the GPGPU and makes the transmission of query tasks to the GPGPU more efficient; on the other hand, a small size achieves lower processing latency because tasks complete more quickly.

Window handling. An important invariant under our hybrid model is that a stream batch, and thus a query task, are independent of the definition of windows over the input streams. Unlike pane-based processing [11, 41], the size of a stream batch is therefore not determined by the window slide. This makes window handling in the hybrid model fundamentally different from that of existing approaches for data-parallel stream processing, such as Spark Streaming [56] or Storm [52]. By decoupling the parallelisation level (i.e. the query task) from the specifics of the query (i.e. the window definition), the hybrid model can support queries over fine-grained windows (i.e. ones with small slides) with full data-parallelism.

Fig. 2 shows the relationship between the stream batches of a query task and the respective window definitions from a query. In this example, stream batches contain 5 tuples. With a window definition of $\omega_{(3,1)}$, stream batch b_1 contains 3 complete windows, w_1, w_2, and w_3, and fragments of two windows, w_4 and w_5. The remaining tuples of the latter two windows are contained in stream batch b_2. For the larger window definition $\omega_{(7,2)}$, the first stream batch b'_1 contains only window fragments: none of w_1, w_2, and w_3 are complete windows. Instead, these windows also span across tuples in stream batch b'_2.

Operators. A stream batch can contain complete windows or only window fragments. Hence, the result of the stream query for a particular sequence of windows (one per input stream) is assembled from the results of multiple query tasks. That is, a window $w = \langle t_1, \ldots, t_k \rangle$ of one of the input streams is partitioned into m *window fragments* $\{d_1, \ldots, d_m\}$, $d_i \in \hat{T}^*$, such that the concatenation of the window fragments yields the window.

As shown in Fig. 3, this has consequences for the realisation of the n-ary operator function f^q: it must be decomposed into a *fragment operator* function f_f^q and an *assembly operator* function f_a^q. The fragment operator function $f_f^q : (\hat{T}^*)^n \to \hat{T}^*$ defines the processing for a sequence of n window fragments, one per stream, and yields a sequence of tuples, i.e. a window fragment result.

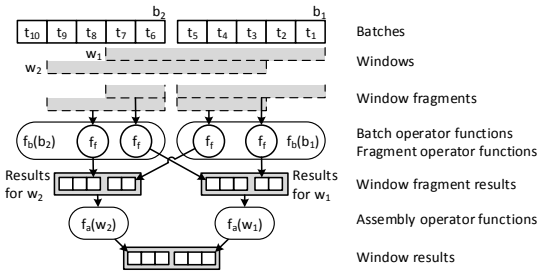


Figure 3: Example of computation based on window fragments

The assembly operator function $f_a^q : \hat{T}^* \rightarrow \hat{T}^*$ constructs the complete window result by combining the window fragment results from f_f^q. The composition of the fragment operator function f_f^q and the assembly operator function f_a^q yields the operator function f^q.

The decomposition of function f^q into functions f_f^q and f_a^q is operator-specific. For many associative and commutative functions (e.g. aggregation functions for count or max), both f_f^q and f_a^q are the original operator function f^q. For other functions, such as median, more elaborate decompositions must be defined [50].

The decomposition also introduces a synchronisation point in the data-parallel execution because the assembly function may refer to the results of multiple tasks. In §4.3, we describe how this synchronisation can be implemented efficiently by reusing threads executing query tasks to perform the assembly function.

Incremental computation. When processing a query task with a sliding window, it is more efficient to use *incremental computation*: the computation for a window fragment should exploit the results already obtained for preceding window fragments in the same batch.

The hybrid model captures this optimisation through a *batch operator* function $f_b^q : ((\hat{T}^*)^n)^m \rightarrow (\hat{T}^*)^m$ that is applied to a query task. The batch operator function takes as input m sequences of n window fragments, where n is the number of input streams and m is the maximum number of window fragments per stream batch in the task. It outputs a sequence of m window fragment results. The same result would be obtained by applying the fragment operator function f_f^q m-times to each of the m sequences of n window fragments.

Example. The effect of applying the hybrid stream processing model is shown in Fig. 3. Windows w_1 and w_2 span two stream batches, and they are processed by two query tasks. For each task, the batch operator function computes the results for the window fragments contained in that task. The assembly operator function constructs the window results from the window fragment results.

4. SABER ARCHITECTURE

We describe SABER, a relational stream processing engine for CPUs and GPGPUs that realises our hybrid stream processing model. We begin this section with an overview of the engine's architecture, and then explain how query tasks are dispatched (§4.1), scheduled (§4.2), and their results collected (§4.3).

Processing stages. The architecture of SABER consists of four stages controlling the lifecycle of query tasks, as illustrated in Fig. 4:

(i) A *dispatching* stage creates query tasks of a fixed size. These tasks can run on either type of processor and are placed into a single, system-wide queue.

(ii) A *scheduling* stage decides on the next task each processor should execute. SABER uses a new *heterogeneous lookahead scheduling* (HLS) algorithm that achieves full utilisation of all heterogeneous processors, while accounting for the different performance characteristics of query operators.

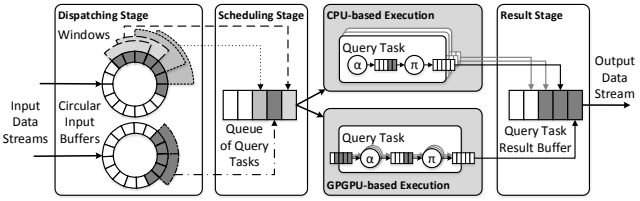


Figure 4: Overview of the Saber architecture

(iii) An *execution* stage runs the query tasks on a processor by evaluating the batch operator function on the input window fragments. A task executes either on one of the CPU cores or the GPGPU.

(iv) A *result stage* reorders the query task results that may arrive out-of-order due to the parallel execution of tasks. It also assembles window results from window fragment results by means of the assembly operator function.

Worker thread model. All four stages are executed by *worker threads*. Each worker is a CPU thread that is bound to a physical core, and one worker uses the GPGPU for the execution of query tasks. Worker threads handle the complete lifecycle of query tasks, which means that they are always fully utilised. This is in contrast to having separate threads for the dispatching or result stage, which could block due to out-of-order processing of query tasks.

Starting with the scheduling stage, the lifecycle of a worker thread is as follows: When a worker thread becomes idle, it invokes the scheduling algorithm to retrieve a task to execute from the system-wide task queue, indicating if it will execute the task on a CPU core or the GPGPU. The worker thread then executes the retrieved task on its associated heterogeneous processor. After task execution completes, the worker thread enters the result stage. In synchronisation with other worker threads, it orders window fragment results and, if possible, assembles complete window results from their fragments. Any assembled window results are then appended to the corresponding query's output data stream.

4.1 Dispatching of query tasks

Dispatching query tasks involves two steps: the storage of the incoming streaming data and the creation of query tasks. Query tasks are inserted into a system-wide queue to be scheduled for execution on the heterogeneous processors.

Buffering incoming data. To store incoming tuples, SABER uses a circular buffer per input stream and per query. By maintaining a buffer per query, processing does not need to be synchronised among the queries. Data can be removed from the buffer as soon as it is no longer required for query task execution.

The circular buffer is backed by an array, with pointers to the start and end of the buffer content. SABER uses byte arrays, so that tuples are inserted into the buffer without prior deserialisation. The implementation is lock-free, which is achieved by ensuring that only one worker thread inserts data into the buffer (worker threads are only synchronised in the result stage) and by explicitly controlling which data is no longer kept in the buffer. A query task contains a *start pointer* and an *end pointer* that define the respective stream batch. In addition, for each input buffer of a query, a query task contains a *free pointer* that indicates up to which position data can be released from the buffer.

A worker thread that executes a query task has only read-access to the buffer. Releasing data as part of the result stage, in turn, means moving the start pointer of the buffer to the position of the free pointer of a query task for which the results have been processed.

Query task creation. The worker thread that inserts data into a circular buffer also handles the creation of query tasks. Instead of first deserialising tuples and computing windows in the sequential dispatching stage, all window computation is handled by the highly parallel query execution stage.

In the dispatching stage, the data inserted into the buffer is added to the current stream batch and, as soon as the sum of stream batch sizes in all query input streams exceeds the query task size ϕ, a query task is created. A *query task identifier* is assigned to each task so that all tasks of a query are totally ordered. This permits the result stage to order the results of query tasks.

4.2 Scheduling of query tasks

The scheduling goal is to utilise all heterogeneous processors fully, yet to account for their different performance characteristics. The challenge is that the throughput of a given type of heterogeneous processor is query-dependent. Query tasks should therefore primarily execute on the processor that achieves the highest throughput for them. At the same time, overloading processors must be avoided because this would reduce the overall throughput. The scheduling decision must thus be query-specific and take the (planned) utilisation of the processors into account.

Instead of using a performance model, SABER observes the *query task throughput*, i.e. the number of query tasks executed per unit of time, defined per query and processor. Based on this, it estimates which processor type is preferred for executing a query task.

At runtime, the preferred processor may be busy executing other tasks. In such cases, SABER permits a task to execute on a non-preferred processor *as long as it is expected to complete before the preferred processor for that task would become available*.

Making scheduling decisions based on the observed task throughput has two advantages: (i) there is no need for hardware-specific performance models and (ii) the scheduling decisions are adaptive to changes in the query workload, e.g. when the selectivity of a query changes, the preferred processor for the tasks may also change. However, this approach assumes that the query behaviour is not too dynamic because, in the long term, the past execution of tasks must allow for the estimation of the performance of future tasks for a given query. Our experimental evaluation shows that this is indeed the case, even for workloads with dynamic changes.

Observing query task throughput. SABER maintains a matrix that records the relative performance differences when executing a query on a given type of processor. The *query task throughput*, denoted by $\rho(q,p)$, is defined as the number of query tasks of query q that can be executed per time unit on processor p. For the CPU, this value denotes the aggregated throughput of all CPU cores; for the GPGPU, it is the overall throughput including data movement overheads.

For a set of queries $Q = \{q_1, \ldots, q_n\}$ and two types of processors, $P = \{\text{CPU},\text{GPGPU}\}$, we define the *query task throughput matrix* as

$$\mathscr{C} = \begin{pmatrix} \rho(q_1,\text{CPU}) & \rho(q_1,\text{GPGPU}) \\ \vdots & \vdots \\ \rho(q_n,\text{CPU}) & \rho(q_n,\text{GPGPU}) \end{pmatrix}.$$

Intuitively, in each row of this matrix, the column with the largest value indicates the preferred processor (highest throughput) for the query; the ratio $r = \rho(q,\text{CPU})/\rho(q,\text{GPGPU})$ is the speed-up ($r > 1$) or slow-down ($r < 1$) of the CPU compared to the GPGPU.

The matrix is initialised under a uniform assumption, with a fixed value for all entries. It is then continuously updated by measuring the number of tasks of a query that are executed in a certain time span on a particular processor. Based on the average time over a fixed set of task executions, the throughput is computed and updated.

Algorithm 1: Hybrid lookahead scheduling (HLS) algorithm

input : $\mathscr{C}$, a query task throughput matrix over $Q = \{q_1, \ldots, q_n\}$ and $P = \{\text{CPU},\text{GPGPU}\}$,
$p \in \{\text{CPU},\text{GPGPU}\}$, the processor for which a task shall be scheduled,
$w = \langle v_1, \ldots, v_k \rangle$, $k > 1$, a queue of query tasks, $\iota(v_i) \in Q$ for $1 \leq i \leq k$,
$count : Q \times P \to \mathbb{N}$, a function assigning a number of executions per query and processor,
st, a switch threshold.
output: $v \in \{v_1, \ldots, v_k\}$ - the query task selected for execution on processor p

```
1  pos ← 1 ;                                              // Initialise position in queue
2  delay ← 0 ;                                            // Initialise delay
   // While end of queue is not reached
3  while pos <= k do
4      q ← ι(w[pos]) ;                                    // Select query of task at current position
5      p_pref ← arg max_{p' ∈ P} 𝒞(q, p') ;               // Determine preferred processor
       // If p is preferred or preference ignored due to delay or switch threshold
6      if (p = p_pref ∧ count(q,p) < st) ∨
       (p ≠ p_pref ∧ (count(q, p_pref) ≥ st ∨ delay ≥ 1/𝒞(q,p))) then
           // If preference was ignored, reset execution counter
7          if count(q, p_pref) ≥ st then count(q, p_pref) ← 0 ;
8          count(q,p) ← count(q,p) + 1 ;                  // Increment execution counter
9          return w[pos] ;                                // Select task at current position
10     delay ← delay + 1/𝒞(q, p_pref) ;                   // Update delay
11     pos ← pos + 1 ;                                    // Move current position in queue
12 return w[pos];
```

Hybrid lookahead scheduling (HLS). The HLS algorithm takes as input (i) a query task throughput matrix; (ii) the system-wide queue of query tasks; (iii) the processor type (CPU or GPGPU) on which a worker thread intends to execute a task; (iv) a function that counts how often query tasks have been executed on each processors; and (v) a switch threshold to ensure that tasks are not executed only on one processor. It iterates over the tasks in the queue and returns the task to be executed by the worker thread on the given processor.

At a high level, the decision as to whether a task v is selected by the HLS algorithm depends on the identified preferred processor and the outstanding work that needs to be done by the preferred processor for earlier tasks in the queue. Such tasks may have the same preferred processor and will be executed *before* task v, thereby delaying it. If this delay is large, it may be better to execute task v on a slower processor if task execution would complete before the preferred processor completes all of its other tasks.

To perform this *lookahead* and quantify the amount of outstanding work for the preferred processor, the HLS algorithm sums up the inverses of the throughput values (variable *delay*). If that delay is larger than the inverse of the throughput obtained for the current task with the slower processor, the slower processor leads to an earlier estimated completion time compared to the preferred processor.

Over time, the above approach may lead to the situation that tasks of a particular query are always executed on the same processor, rendering it impossible to observe the throughput of other processors. Therefore, the HLS algorithm includes a *switch threshold*, which imposes a limit on how often a given task can be executed on the same processor without change: a given task is not scheduled on the preferred processor if the number of tasks of the same query executed on the preferred processor exceeds this threshold.

The HLS algorithm is defined more formally in Alg. 1. It iterates over the tasks in the queue (lines 3–11). For each task, the query operator function of the task is selected (line 4). Then, the preferred processor for this query is determined by looking up the highest value in the row of the query task throughput matrix (line 5).

The current task is executed on the processor associated with the worker thread: (i) if the preferred processor matches the worker's processor, and the number of executions of tasks of this query on this processor is below the switch threshold or (ii) if the worker's

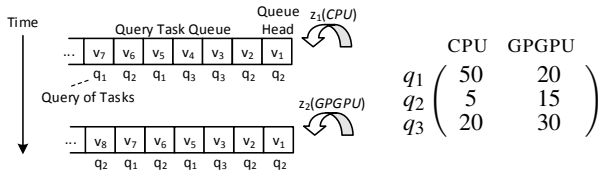


Figure 5: Example of hybrid lookahead scheduling

processor is not the preferred one but the accumulated delay of the preferred processor means that execution on the worker's processor is beneficial, or the number of tasks of this query on the preferred processor is higher than the switch threshold (line 6).

If a non-preferred processor is selected for the current task, the execution counter of the preferred processor is reset (line 7). The execution counter for the selected processor is incremented (line 8), and the current task is returned for execution (line 9). If the current task is not selected, the accumulated delay is updated (line 10), because the current task is planned to be executed on the other processor. Finally, the next query task in the queue is considered (line 11).

Example. Fig. 5 gives an example of the execution of the HLS algorithm. The query task queue contains tasks for three queries, q_1–q_3. Assuming that a worker thread z_1 for processor CPU becomes available, HLS proceeds as follows: the head of the queue, v_1, is a task of query q_2 that is preferably executed on GPGPU. Hence, it is not selected and the planned delay for GPGPU is set to $1/15$. The second and third tasks are handled similarly, resulting in an accumulated delay for GPGPU of $1/15+1/15+1/30=1/6$. The fourth task v_4 is also preferably executed on GPGPU. However, the estimation of the accumulated delay for GPGPU suggests that the execution of v_4 on CPU would lead to an earlier completion time, hence it is scheduled there. Next, if a worker thread z_2 that executes tasks on GPGPU becomes available, it takes the head of the queue because GPGPU is the preferred processor for tasks of query q_2.

4.3 Handling of task results

The result stage collects the results from all tasks that belong to a specific query and creates the output data stream. To maintain low processing latency, results of query tasks should be processed as soon as they are available, continuously creating the output data stream from the discretised input data streams. This is challenging for two reasons: first, the results of query tasks may arrive out-of-order due to their parallel execution, which requires reordering to ensure correct results; second, the window fragment results of a window may span multiple query tasks, which creates dependencies when executing the assembly operator function.

To address these challenges, the result stage reduces the synchronisation needed among the worker threads. To this end, processing of query task results is organised in three phases, each synchronised separately: (i) storing the task results, i.e. the window fragment results, in a buffer; (ii) executing the assembly operator function over window fragment results to construct the window results; and (iii) constructing the output data stream from the window results.

Result storage. When worker threads enter the result stage, they store the window fragment results in a circular *result buffer*, which is backed by a byte array. The order in which results are stored is determined by the query task identifiers assigned in the dispatch stage: a worker thread accesses a particular slot in the result buffer, which is the query task identifier modulo the buffer size.

To ensure that worker threads do not overwrite results of earlier query tasks stored in the result buffer, a *control buffer* records status flags for all slots in the result buffer. It is accessed by worker threads using an atomic compare-and-swap instruction. To avoid blocking a worker thread if a given slot is already populated, the result buffer has more slots than the number of worker threads. This guarantees that the results in a given slot will have been processed before this slot is accessed again by another worker thread.

Assembly of window results. After a worker thread has stored the results of a query task, it may proceed by assembling window results from window fragment results. In general, the results of multiple tasks may be needed to assemble the result of a single window, and assembly proceeds in a pairwise fashion. Given the results of two consecutive tasks, the windows are processed in their logical order, as defined by the query task identifier. The window result is calculated by applying the assembly operator function to all results of relevant fragments, which are contained in the two query task results. The obtained window results directly replace the window fragment results in the result buffer.

If additional fragment results are needed to complete the assembly for a window, i.e. the window spans more than two query task results, multiple assembly steps are performed. The assembly operator function is then applied to the result of the first assembly step and the fragment results that are part of the next query task result.

If a window spans more than two query tasks, and the assembly of window results is not just a concatenation of fragment results, the above procedure is not commutative—a window fragment result may be needed to construct the results of several windows, which are part of different evaluations of the assembly operator function. In that case, the pairwise assembly steps must be executed in the order defined by the query task identifiers. Such ordering dependencies can be identified by considering for each query task whether it contained windows that opened or closed in one of the stream batches. A query task result is ready for assembly either if it does not contain windows that are open or closed in one of the stream batches, or if the preceding query task result has already been assembled.

The information about task results that are ready for assembly and task results that have been assembled are kept as status flags in the control buffer. After a worker thread stores the result of a query task in the result buffer, it checks for slots in the result buffer that are ready for assembly. If there is no such slot, the worker thread does not block, but continues with the creation of the output stream.

Output stream construction. After window result assembly, the output data stream is constructed. A worker thread that finished assembly of window results (or skipped over it) checks if the slot of the result buffer that contains the next window result to be appended to the output stream is ready for result construction. If so, the worker thread locks the respective slot and appends the window results to the output stream. If there is no such slot, the worker thread leaves the result stage and returns to the scheduling stage.

5. QUERY EXECUTION

We now describe SABER's query execution stage in more detail. A challenge is that the performance of streaming operators may be restricted by their memory accesses. SABER thus explicitly manages memory to avoid unnecessary data deserialisation and dynamic object creation (§5.1). Whether the GPGPU can be utilised fully depends on how efficiently data is transferred to and from the GPGPU. SABER employs a novel approach to pipelined stream data movement that interleaves the execution of query tasks with data transfer operations (§5.2). Finally, we describe our implementation of streaming operators on the CPU (§5.3) and GPGPU (§5.4).

5.1 Memory management

Dynamic memory allocation is one of the main causes of performance issues in stream processing engines implemented in managed

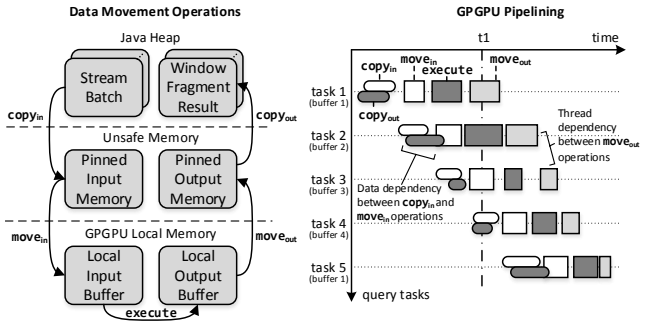


Figure 6: Data movement and pipelining for GPGPU-based execution of query tasks

languages with garbage collection, such as Java or C#. The SABER implementation therefore minimises the number of memory allocations by means of lazy deserialisation and object pooling.

Lazy deserialisation. To manage the data in input streams efficiently, SABER uses lazy deserialisation: tuples are stored in their byte representation and deserialised only if and when needed. Deserialisation is controlled per attribute, i.e. tuple values that are not accessed or needed to compute windows are not deserialised. Deserialisation only generates primitive types, which can be efficiently packed in byte arrays without the overhead of pointer dereferencing of object types. In addition, whenever possible, operators in SABER realise *direct byte forwarding*, i.e. they copy the byte representation of the input data to buffers that store intermediate results.

Object pooling. To avoid dynamic memory allocation on the critical processing path, SABER uses statically allocated pools of objects, for all query tasks, and of byte arrays, for storing intermediate window fragment results. To avoid contention for the byte arrays when many worker threads access it, each thread maintains a separate pool.

5.2 Pipelined stream data movement

Executing a query task on the GPGPU involves several data movement operations, as shown on the left side of Fig. 6: (i) the input data is copied from Java heap memory to pinned host memory ($\mathsf{copy}_{\mathsf{in}}$); (ii) using DMA, data is transferred from pinned host memory to GPGPU memory for processing ($\mathsf{move}_{\mathsf{in}}$); (iii) after the query task has executed (execute), the results are transferred back from the GPGPU memory to pinned host memory ($\mathsf{move}_{\mathsf{out}}$); and (iv) the results are copied back to Java heap memory ($\mathsf{copy}_{\mathsf{out}}$).

Performing these data movement operations sequentially would under-utilise the GPGPU. Given that stream processing handles small amounts of data in practice, the PCIe throughput becomes the bottleneck—executing the $\mathsf{copy}_{\mathsf{in}}$ and $\mathsf{copy}_{\mathsf{out}}$ operations sequentially would reduce the throughput to half of the PCIe bandwidth.

SABER therefore uses a *five-stage pipelining* mechanism, which interleaves I/O and compute operations to reduce idle periods and yield higher throughput. Pipelining is achieved by having dedicated threads execute each of the five data movement operations in parallel: two CPU threads copy data from and to Java heap memory ($\mathsf{copy}_{\mathsf{in}}$ and $\mathsf{copy}_{\mathsf{out}}$), and two GPGPU threads implement the data movement from and to GPGPU memory ($\mathsf{move}_{\mathsf{in}}$ and $\mathsf{move}_{\mathsf{out}}$). The remaining GPGPU threads execute query tasks (execute).

The right side of Fig. 6 shows the interleaving of data movement operations. Each row shows the five operations for a single query task. The interleaving of operations can be seen, for example, at time t1, when $\mathsf{move}_{\mathsf{out}}$ of task 1, execute of task 2, $\mathsf{move}_{\mathsf{in}}$ of task 3, and the copy operations of task 4 all execute concurrently.

The interleaving of operations respects data and thread dependencies. For a specific query task, all operations are executed sequentially because each operation relies on the data of the previous one. The CPU threads executing the copy operations, however, run in parallel so that the result data of the i-th task is copied by the $\mathsf{copy}_{\mathsf{out}}$ operation of the $(i+4)$-th task: as shown in Fig. 6, task 5's $\mathsf{copy}_{\mathsf{out}}$ operation returns the results of task 1. If task 1's $\mathsf{move}_{\mathsf{out}}$ operation has not completed, task 5 will be blocked until a notification from the dedicated GPGPU thread performing $\mathsf{move}_{\mathsf{out}}$.

The execution of each data movement operation by a thread also results in the sequential execution of the same operation of different tasks: in the example, task 2's $\mathsf{move}_{\mathsf{out}}$ operation must complete before the respective thread executes task 3's $\mathsf{move}_{\mathsf{out}}$ operation.

5.3 CPU operator implementations

To execute a query task on a CPU core, a worker thread applies the window batch function of a given query operator to the window fragments of the stream batches for this task. SABER implements the operator functions as follows.

Projection and selection operators are both stateless, and their batch operator function is thus a single scan over the stream batch of the corresponding query task, applying the projection or selection to each tuple. The assembly operator function then concatenates the fragment results needed to obtain a window result according to the stream creation function (see §2.4).

Aggregation operators, such as *sum*, *count* and *average*, over sliding windows exploit *incremental computation* [12, 50]. In a first step, the batch operator function computes window fragments by partitioning the stream batch of the query task into panes, i.e. distinct subsequences of tuples. To compute the aggregation value per window fragment, the batch operator function reuses the aggregated value—the result—of previous window fragments and incorporates tuples of the current fragment.

For an aggregation operator, a window result cannot be constructed by simply concatenating window fragment results, but the operator function must be evaluated for a set of fragment results. The assembly of window results from window fragment results is done stepwise, processing the window fragment results of two query tasks at a time. As described in §4.3, SABER keeps track for each fragment whether it is part of a window that opened, closed, is pending (i.e. neither opened nor closed), or fully contained in the stream batch of the query task. This is achieved by storing window fragment results in four different buffers (byte arrays) that contain only fragments that belong to windows in specific states.

SABER also supports the definition of HAVING and GROUP-BY clauses. The implementation of the HAVING clause reuses the selection operator. For the GROUP-BY clause, the batch operator function maintains a hash table with an aggregation value per group. To avoid dynamic memory allocation, SABER uses a statically allocated pool of hash table objects, which are backed by byte arrays.

Join operators implement a streaming θ-join, as defined by Kang et al. [35]. The execution of query tasks for the join is sequential because, unlike other task-parallel join implementations [27], SABER achieves parallelism by the data-parallel execution of query tasks.

5.4 GPGPU operator implementations

GPGPU operators are implemented in OpenCL [40]. Each operator has a generic code template that contains its core functionality. SABER populates these templates with tuple data structures, based on the input and output stream schemas of each operator, and query-specific functions over tuple attributes (e.g. selection predicates or aggregate functions). Tuple attributes are represented both as primitive types and vectors, which allows the implementation to use vectorised GPGPU instructions whenever possible.

Each GPGPU thread evaluates the fragment operator function. To exploit the cache memory shared by the synchronised threads on a single GPGPU core, tuples that are part of the same window are assigned to the same work group. The assembly operator function that derives the window results from window fragment results (see §4.3) is evaluated by one of the CPU worker threads.

Projection and selection. To perform a projection, the threads in a work group load the tuples into the group's cache memory, compute the projected values, and write the results to the global GPGPU memory. Moving the data to the cache memory is beneficial because the deserialisation and serialisation of tuples becomes faster. For the selection operator, the result from a fragment operator function is stored as a binary vector in the group's cache memory. Here, each vector entry indicates whether a tuple has been selected.

In a second step, SABER uses a prefix-sum (or scan) operation [14] to write the output data to a continuous global GPGPU memory region: it scans the binary vector and obtains continuous memory addresses to which selected tuples are written. These addresses are computed per window fragment and offset based on the start address of each fragment in order to compress the results for all fragments in a stream batch.

Aggregation first assigns a work group to each of the window fragments in the stream batch of the query task. For the implemented commutative and associative operator functions, each thread of the work group repeatedly reads two tuples from the global GPGPU memory and aggregates them. The threads thus form a reduction tree from which the window fragment result is derived. As in CPU-based execution (§5.3), the results are stored while keeping track for each fragment whether it is part of a window that opens, closes, is pending, or is fully contained in the stream batch.

To implement aggregations with `HAVING` clauses, SABER relies on the aforementioned approach for selection, which is applied to the aggregation results. For a `GROUP-BY` clause, every work group populates an open-addressing hash table. The table uses a linear probing approach combined with a memory-efficient representation. The size of the table and the hash function used on the CPU and GPGPU are the same to ensure that, given a tuple in the CPU table, it can be searched in the GPGPU table, and vice versa.

An additional challenge, compared to the CPU implementation, is to ensure that the hash table for a window fragment is thread-safe. We reserve an integer in the intermediate tuple representation to store the index of the first input tuple that has occupied a table slot. Threads try to atomically compare-and-set this index. When a collision occurs, a thread checks if the key stored is the same as the current key, in which case it atomically increments the aggregate value, or it moves to the next available slot.

If the window fragment is incomplete, the hash table is output as is—this is necessary because the result aggregation logic is the same for both CPU and GPGPU. Otherwise, all sparsely populated hash tables are compacted at the end of processing.

Join operators exploit task-parallelism, similar to the *cell join* [27]. To avoid quadratic memory usage, however, SABER adopts a technique used in join processing for in-memory column stores [32] and performs a join in two steps: (i) the number of tuples that match the join predicate is counted and (ii) the results are compressed in the global GPGPU memory, as for selection.

6. EVALUATION

We evaluate SABER to show the benefits of its hybrid stream processing model. First, we demonstrate that such a model achieves higher performance for both real-world and synthetic query benchmarks (§6.2). We then explore the trade-off between CPU and GPGPU execution for different queries (§6.3). After that, we investigate the impact of the query task size on performance (§6.4) and show that SABER's CPU operator implementations scale (§6.5). Finally, we evaluate the effectiveness of HLS scheduling (§6.6).

Datasets		Queries			
Name	# Attr.	Name	Windows	Operators	Values
Synthetic (Syn)	7	$\mathsf{PROJ_m}$	*various*	$\pi_{a_1,\ldots,a_m}$	$1 \leq m \leq 10$
		$\mathsf{SELECT_n}$	*various*	$\sigma_{p_1,\ldots,p_n}$	$1 \leq n \leq 64$
		$\mathsf{AGG_f}$	*various*	α_f	$f \in \{\text{avg}, \text{sum}\}$
		$\mathsf{GROUP\text{-}BY_o}$	*various*	$\gamma_{g_1,\ldots,g_o}$	$1 \leq o \leq 64$
		$\mathsf{JOIN_r}$	*various*	$\bowtie_{p_1,\ldots,p_r}$	$1 \leq r \leq 64$
Cluster Monitoring (CM)	12	$\mathsf{CM_1}$	$\omega_{60,1}$	$\pi, \gamma, \alpha_{\text{sum}}$	
		$\mathsf{CM_2}$	$\omega_{60,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$	
Smart Grid (SG)	7	$\mathsf{SG_1}$	$\omega_{3600,1}$	π, α_{avg}	
		$\mathsf{SG_2}$	$\omega_{3600,1}$	$\pi, \gamma, \alpha_{\text{avg}}$	
		$\mathsf{SG_3}$	$\omega_{1,1}, \omega_{1,1}$	$\pi, \sigma, \bowtie$	
Linear Road Benchmark (LRB)	7	$\mathsf{LRB_1}$	ω_{∞}	π	
		$\mathsf{LRB_2}$	$\omega_{30,1}, \omega_{\text{part}}$	$\pi_{\text{distinct}}, \bowtie$	
		$\mathsf{LRB_3}$	$\omega_{300,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$	
		$\mathsf{LRB_4}$	$\omega_{30,1}$	$\pi, \gamma, \alpha_{\text{count}}$	

Table 1: Summary of evaluation datasets and workloads

6.1 Experimental set-up and workloads

All experiments are performed on a server with 2 Intel Xeon E5-2640 v3 2.60 GHz CPUs with a total of 16 physical CPU cores, a 20 MB LLC cache and 64 GB of memory. The server also hosts an NVIDIA Quadro K5200 GPGPU with 2,304 cores and 8 GB GDDR 5 memory, connected via a PCIe 3.0 ($\times$16) bus. When streaming data to SABER from the network, we use a 10 Gbps Ethernet NIC. The server runs Ubuntu 14.04 with Linux kernel 3.16 and NVIDIA driver 346.47. SABER uses the Oracle JVM 7 with the *mark-and-sweep* garbage collector.

Table 1 summarises the query workloads and datasets in our evaluation. We use both synthetic queries, for exploring query parameters, and application queries from real-world use cases. The CQL representations of the application queries are listed in Appendix A.

Synthetically-generated workload (Syn). We generate synthetic data streams of 32-byte tuples, each consisting of a 64-bit timestamp and six 32-bit attribute values drawn from a uniform distribution—the default being integer values, but for aggregation and projection the first value being a float. We then generate a set of queries using the operators from §2.4, and vary key parameters: the number of attributes and arithmetic expressions in a projection; the number of predicates in a selection or a join; the function in an aggregation; and the number of groups in an aggregation with `GROUP-BY`.

Compute cluster monitoring (CM). Our second workload emulates a cluster management scenario. We use a trace of time-stamped tuples collected from an 11,000-machine compute cluster at Google [53]. Each tuple is a monitoring event related to the tasks of compute jobs that execute on the cluster, such as the successful completion of a task, the failure of a task, or the submission of a high-priority task for a production job.

We use two queries, $\mathsf{CM_1}$ and $\mathsf{CM_2}$, that express common cluster monitoring tasks [23, 38]: $\mathsf{CM_1}$ combines a projection and an aggregation with a `GROUP-BY` clause to compute the sum of the requested share of CPU utilisation per job category; and $\mathsf{CM_2}$ combines a projection, a selection, and an aggregation with a `GROUP-BY` to report the average requested CPU utilisation of submitted tasks.

Anomaly detection in smart grids (SG). This workload focuses on anomaly detection in energy consumption data from a smart electricity grid. The trace is a stream of smart meter readings from different electrical devices in households [34].

We execute three queries, $\mathsf{SG_{1–3}}$, that analyse the stream to detect outliers. $\mathsf{SG_1}$ is a projection and aggregation that computes the

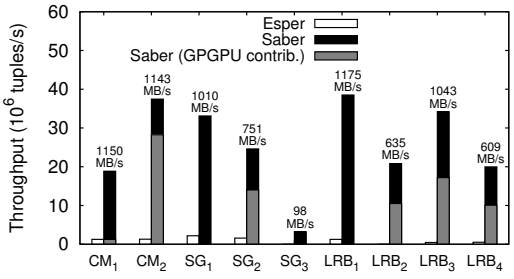


Figure 7: Performance for application benchmark queries

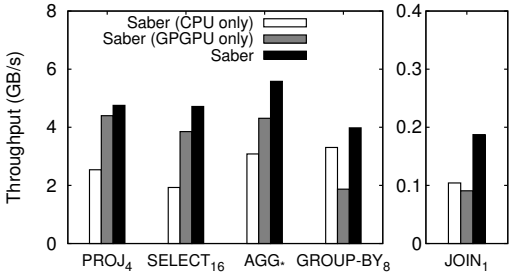


Figure 8: Performance for synthetic queries. AGG_* evaluates all aggregate functions and GROUP-BY evaluates cnt and sum.

sliding global average of the meter load; SG_2 combines a projection and aggregation with a GROUP-BY to derive the sliding load average per plug in a household in a house; and SG_3 uses a join to compare the global load average with the plug load average by joining the results of SG_1 and SG_2, and then counts the houses for which the local average is higher than the global one.

Linear Road Benchmark (LRB). This workload is the Linear Road Benchmark [8] for evaluating stream processing performance. The benchmark models a network of toll roads, in which incurred tolls depend on the level of congestion and the time-of-day. Tuples in the input data stream denote position events of vehicles on a highway lane, driving with a specific speed in a particular direction.

6.2 Is hybrid stream processing effective?

We first explore the performance of SABER for our application benchmark queries. To put the achieved processing throughput into perspective, we also compare to an implementation of the queries in Esper [2], an open-source multi-threaded stream processing engine. The input streams for the queries are generated by a separate machine connected through a 10 Gbps network link.

Fig. 7 shows the processing throughput for SABER and Esper for the application queries. For SABER, we also report the split of the contribution of the CPU (upper black part) and the GPGPU (lower grey part) to the achieved throughput. SABER manages to saturate the 10 Gbps network link for many queries, including CM_1, CM_2 and LRB_1. In contrast, the performance of Esper remains two orders of magnitude lower due to the synchronisation overhead of its implementation and the lack of GPGPU acceleration.

Different queries exhibit different acceleration potential on the CPU versus the GPGPU, which SABER automatically selects according to its hybrid execution model. For CM_1, the aggregation executes efficiently on the CPU, whereas the selection in CM_2 on the GPGPU benefits from its high throughput, leaving aggregation over sliding windows to the CPU. SG_1 and LRB_1 do not benefit from the GPGPU because the CPU workers can keep up with the task generation rate. SG_2 and LRB_3 utilise both heterogeneous processors equally because they can split the load.

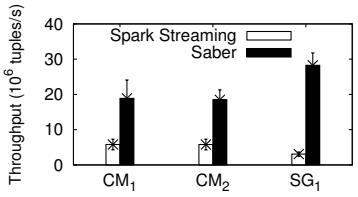
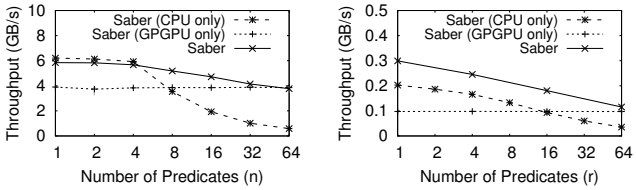


Figure 9: Performance compared to Spark Streaming



(a) $SELECT_n$ with $\omega_{32KB,32KB}$ **(b)** $JOIN_r$ with $\omega_{4KB,4KB}$

Figure 10: Performance impact of query parameters

Next, we use our synthetic operator queries to compare the performance achieved by SABER to CPU-only and GPGPU-only execution, respectively. Fig. 8 shows that the hybrid approach always achieves higher throughput than only using a single type of heterogeneous processor. The contention in the dispatching and result stages in SABER means that the combined throughput is less than the sum of the throughput achieved by the CPU and the GPGPU only.

Fig. 9 shows the performance of SABER and Spark Streaming with the CM_1, CM_2 and SG_1 queries. Since Spark does not support count-based windows, we change the queries to use 500 ms tumbling windows based on system time for comparable results. For CM_1 and CM_2, SABER's throughput is limited by the network bandwidth, while, for SG_1, SABER saturates 90% of the network bandwidth, showing a throughput that is 6× higher than Spark Streaming, which is limited due to scheduling overhead.

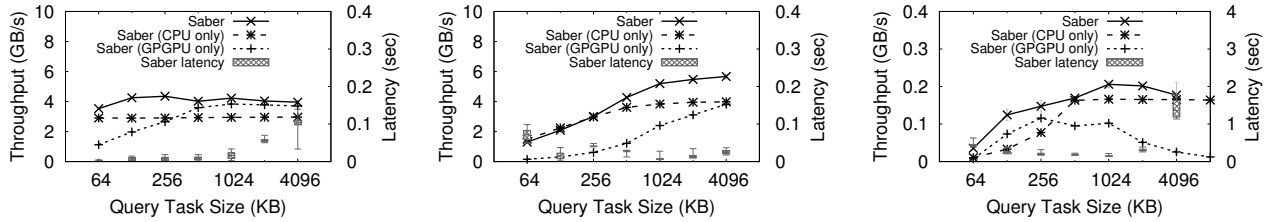
We put SABER's performance into perspective by comparing it against MonetDB [33], an efficient in-memory columnar database engine. We run a θ-join query over two 1 MB tables with 32-byte tuples. The tables are populated with synthetic data such that the selectivity of the query is 1%. In SABER, we emulate the join by treating the tables as streams and applying a tumbling window of 1 MB. In MonetDB, we partition the two tables and join the partitions pairwise so that the engine can execute each of the partial θ-joins in parallel. The output is the union of all partial join results.

When the query output has only two columns—those on which we perform the θ-join—MonetDB and SABER exhibit similar performance: with 15 threads, MonetDB runs the query in 980 ms and SABER in 1,088 ms.[2] When the query output includes all columns (`select *`), MonetDB is 2× slower than SABER because, as a columnar engine, it spends 40% of the time reconstructing the output table after the join evaluation. However, when we change the query to an equi-join with the same selectivity (rather than a θ-join), MonetDB is 2.7× faster because it is highly-optimised for such queries.

6.3 What is the CPU/GPGPU trade-off?

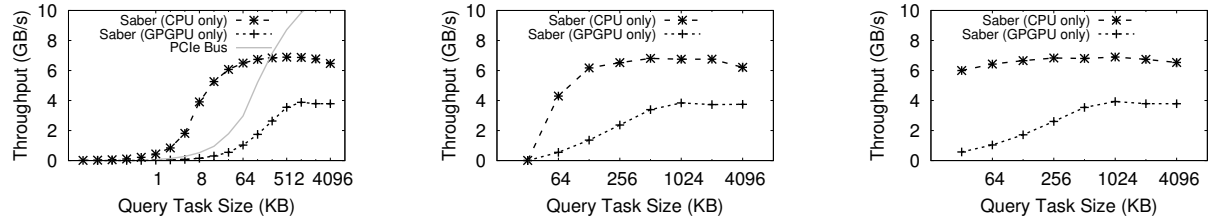
As we increase the complexity of some query operators, the CPU processing throughput degrades. Keeping the number of worker threads fixed to 15, Fig. 10 shows this effect for $SELECT_n$ and $JOIN_r$ when varying the number of predicates (the other operators yield similar results). For $SELECT_n$, we further observe that, for few predicates, the throughput is bound by the rate at which the dispatcher can generate tasks.

[2]We report the average of 5 runs. In all SABER runs, the GPGPU consumes approximately $1/3$ of the dispatched tasks.



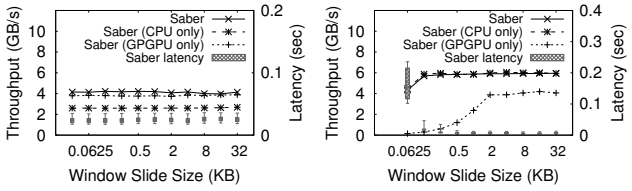
(a) SELECT_{10} with $\omega_{32\text{KB},32\text{KB}}$ **(b)** $\mathsf{AGG}_{\mathsf{avg}}\mathsf{GROUP\text{-}BY}_{64}$ with $\omega_{32\text{KB},32\text{KB}}$ **(c)** JOIN_4 with $\omega_{32\text{KB},32\text{KB}}$

Figure 12: Performance impact of query task size ϕ for different query types



(a) SELECT_1 with $\omega_{32\text{B},32\text{B}}$ **(b)** SELECT_1 with $\omega_{32\text{KB},32\text{B}}$ **(c)** SELECT_1 with $\omega_{32\text{KB},32\text{KB}}$

Figure 13: Performance impact of query task size ϕ for different window sizes and slides



(a) SELECT_{10} with $\omega_{32\text{KB},x}$ **(b)** $\mathsf{AGG}_{\mathsf{avg}}$ with $\omega_{32\text{KB},x}$

Figure 11: Performance impact of window slide

We compare the CPU throughput with that obtained from using just the GPGPU. Fig. 10 shows that there are regions where the GPGPU is faster than the CPU, and vice versa. The GPGPU is bound by the data movement overhead, and any performance benefit diminishes when the query is not complex enough. Except for the case in which the dispatcher is the bottleneck because of a simple query structure (SELECT_n, $1 \leq n \leq 4$), hybrid execution is beneficial for both queries, achieving nearly additive throughput.

Fig. 11 shows the impact of the window slide on the CPU/GPGPU trade-off for two sliding window queries, SELECT_{10} and $\mathsf{AGG}_{\mathsf{avg}}$, under a fixed task size of 1 MB. The window size is 32 KB, while the slide varies from 32 bytes (1 tuple) to 32 KB. As expected, the slide does not influence the throughput or latency on any processor for SELECT_{10} because the selection operator does not maintain any state. For $\mathsf{AGG}_{\mathsf{avg}}$, the CPU uses incremental computation, which increases the throughput until it is bound by the dispatcher. On the GPGPU, an increase of the window slide reduces the number of windows processed, which increases throughput. At some point, however, the GPGPU throughput is limited by the PCIe bandwidth.

6.4 How does task size affect performance?

Next, we investigate the impact of the query task size on performance. Intuitively, larger tasks improve throughput but negatively affect latency. We first consider three queries, SELECT_{10}, $\mathsf{AGG}_{\mathsf{avg}}\mathsf{GROUP\text{-}BY}_{64}$, and JOIN_4, with $\omega_{32\text{KB},32\text{KB}}$. In each experiment, we measure the throughput and latency when varying query task sizes from 64 KB to 4 MB.

The results in Fig. 12 confirm our expectation. While the absolute throughput values vary across the different queries, they exhibit a similar trend: initially, the throughput grows linearly with the query task size and then plateaus at around 1 MB. The only exception is the GPGPU-only implementation of the JOIN query for which throughput collapses with query task sizes beyond 512 KB. This is due to a limitation of our current implementation: the computation of the window boundaries is always executed on the CPU.

To validate our claim that the batch size is independent from the window size and slide, we use a single type of query, SELECT_1, and vary the window size and slide: from a window size of just 1 tuple ($\omega_{32\text{B},32\text{B}}$), to a slide of just 1 tuple ($\omega_{32\text{KB},32\text{B}}$), and to a large tumbling window ($\omega_{32\text{KB},32\text{KB}}$).

Fig. 13 shows that SABER achieves a comparable performance in all three cases, with the throughput growing approximately until 1 MB. This demonstrates that the batch size is independent from the executed query but only depends on the hardware.

6.5 Does Saber scale on the CPU?

We now turn our attention to the scalability of SABER's CPU operator implementations. We measure the throughput achieved by each operator in isolation when varying the number of worker threads. Fig. 14 shows the results for a PROJ_6 query with $\omega_{32\text{KB},32\text{KB}}$ windows. We omit the results for the other operators because they exhibit similar trends. The results show that the CPU operator implementation scales linearly up to 16 worker threads. Beyond this number, the performance plateaus (or slightly deteriorates for some operators) due to the context-switching overhead when the number of threads exceeds the number of available physical CPU cores.

6.6 What is the effect of HLS scheduling?

We finish by investigating the benefit of hybrid lookahead scheduling (HLS). For this, we consider a workload W_1 with two queries, Q_1=$\mathsf{PROJ}_6{}^*$ (i.e. PROJ_6 with 100 arithmetic expressions for each attribute) with $\omega_{32\text{KB},32\text{KB}}$, and Q_2=$\mathsf{AGG}_{\mathsf{cnt}}\mathsf{GROUP\text{-}BY}_1$ with $\omega_{32\text{KB},16\text{KB}}$, executed in sequence. These queries exhibit opposite performance: when run in isolation, Q_1 has higher throughput on the GPGPU (1,475 MB/s vs. 292 MB/s), while Q_2 has higher throughput on the CPU (2,362 MB/s vs. 372 MB/s).

We compare the performance achieved by HLS against two baselines: a "first-come, first-served" (FCFS) scheduling policy and a static scheduling policy (Static), in which Q_1's tasks are scheduled

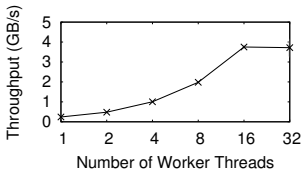


Figure 14: Scalability of CPU operator implementation

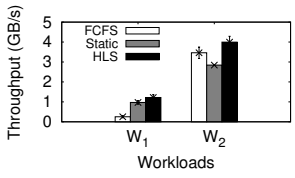


Figure 15: Performance impact of HLS scheduling

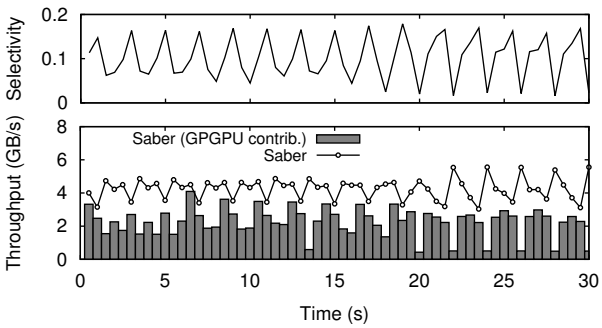


Figure 16: Adaptation of HLS scheduling

on the GPGPU and Q_2's task are scheduled on the CPU. Note that this policy is infeasible in practice with dynamic workloads.

Fig. 15 shows that, as expected, FCFS has the worst performance as tasks are not matched with the preferred processor. More interesting is the comparison between Static and HLS: although Static significantly improves over FCFS, HLS achieves higher throughput because it exploits all available resources.

Next, we consider a scenario in which Static would under-perform. Workload W_2 consists of two queries, Q_3=PROJ$_1$ with $\omega_{32\text{KB},32\text{KB}}$ and Q_4=AGG$_{\text{sum}}$ with $\omega_{32\text{KB},32\text{KB}}$, executed in sequence. Statically assigning Q_3 to the CPU and Q_4 to the GPGPU yields full utilisation of the GPGPU, but CPU cores are under-utilised. Reversing the assignment has the opposite effect. In Fig. 15, we show the latter assignment due to its higher throughput. Splitting the workload with FCFS, both CPU and GPGPU are fully utilised with a CPU/GPGPU split of approximately 1:1.5 for both Q_3 and Q_4. HLS saturates both processors too, but with peak throughput by converging to a better CPU/GPGPU split of 1:2.5 for Q_3 and 1:0.5 for Q_4. The results for W_1 and W_2 thus show the ability of HLS to support heterogeneous processors without *a priori* knowledge of the workload.

Fig. 16 shows how HLS adapts to workload changes. We run a SELECT$_{500}$ query over the cluster management trace, filtering task failure events. The selection predicate has the form $p_1 \wedge (p_2 \vee \cdots \vee p_{500})$ such that when a failure event is selected (p_1) all other predicates are evaluated too, making query tasks with high selectivity expensive to run. The trace contains a period with a surge of task failure events, which we repeat. The resulting selectivity is shown in the upper part of Fig. 16. To enable HLS to react to frequent changes, we update the query task throughput matrix every 100 ms.

Fig. 16 shows how HLS adapts: when selectivity is low, the CPU is faster than the GPGPU and monopolises the task queue—the GPGPU contribution is limited to tasks permitted to run by HLS's switch threshold rule. When the selectivity increases, the GPGPU is the faster processor. As the query task throughput for both processors changes, HLS adapts by scheduling more tasks on the GPGPU to sustain a high throughput.

7. RELATED WORK

Centralised stream processing engines have existed for decades, yet engines such as STREAM [6], TelegraphCQ [20], and NiagaraCQ [21] focus on single-core execution. Recent systems such as Esper [2], Oracle CEP [3], and Microsoft StreamInsight [39] support multi-core architectures at the expense of weakening stream ordering guarantees in the presence of windows. Research prototypes such as S-Store [18] and Trill [19] have strong window semantics with SQL-like queries. However, S-Store does not perform parallel window computation. Trill parallelises window processing through a map/reduce model, but it does not support hybrid query execution.

Distributed stream processing systems such as Storm [52], Samza [1] and SEEP [17] execute streaming queries with data-parallelism on a cluster of nodes. They do not respect window semantics by default. Millwheel [4] provides strong window semantics, but it does not perform parallel computation on windows and instead assumes partitioned input streams. Spark Streaming [56] has a batched-stream model, and permits window definitions over this model, thus creating dependencies between window semantics, throughput and latency. Recent work on adapting the batch size for Spark Streaming [25] automatically tunes the batch size to stay within a given latency bound. Unlike Streaming Spark, SABER decouples window semantics from system performance.

Window computation. Pane-based approaches for window processing [41] divide overlapping windows into panes. Aggregation occurs at the pane level, and multiple panes are combined to provide the final window result. In contrast to our hybrid stream processing model, the goal is to partition windows so as to avoid redundant computation, not to achieve data parallelism. Balkesen and Tatbul [11] propose pane processing for distribution, which incurs a different set of challenges compared to SABER due to its shared-nothing model.

Recent work [12, 50] has focused on the problem of avoiding redundant computation with sliding windows aggregation. This involves keeping efficient aggregation data structures, which can be maintained with minimum overhead [12], even for general aggregation functions that are neither invertible nor commutative [50]. Such approaches are compatible with the batch operator functions that SABER uses to process multiple overlapping window fragments.

Accelerated query processing. In-memory databases have explored co-processing with CPUs and GPGPUs for accelerating database queries for both in-cache and discrete systems. All such systems [15, 16, 22, 30, 48, 55], however, target one-off and not streaming queries, and they therefore do not require parallel window semantics or efficient fine-grained data movement to the accelerator.

GStream [57] executes streaming applications on GPGPU clusters, and provides a new API that abstracts away communication primitives. Unlike GStream, SABER supports SQL window semantics and deployments on single-node hybrid architectures.

8. CONCLUSIONS

We have presented SABER, a stream processing system that employs a new hybrid data parallel processing model to run continuous queries on servers equipped with heterogeneous processors, namely a CPU and a GPGPU. SABER does not make explicit decisions on which processor a query should run but rather gives preference to the fastest one, yet enabling other processors to contribute to the aggregate query throughput. We have shown that SABER achieves high processing throughput (over 6 GB/s) and sub-second latency for a wide range of streaming queries.

Acknowledgements. Research partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 318521, the German Research Foundation (DFG) under grant agreement number WE 4891/1-1, and a PhD CASE Award by EPSRC/BAE Systems.

9. REFERENCES

[1] Apache Samza. http://samza.apache.org/. Last access: 09/02/16.

[2] Esper. http://www.espertech.com/esper/. Last access: 09/02/16.

[3] Oracle® Stream Explorer. http://bit.ly/1L6tKz3. Last access: 09/02/16.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[5] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP Server and Online Behavioral Targeting. *Proc. VLDB Endow.*, 2(2):1558–1561, Aug. 2009.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.*, 26(1):19–26, Mar. 2003.

[7] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[8] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[9] A. Artikis, M. Weidlich, F. Schnitzler, I. Boutsis, T. Liebig, N. Piatkowski, C. Bockermann, K. Morik, V. Kalogeraki, J. Marecek, A. Gal, S. Mannor, D. Gunopulos, and D. Kinane. Heterogeneous Stream Processing and Crowdsourcing for Urban Traffic Management. In *Proceedings of the 17th International Conference on Extending Database Technology*, EDBT '14, pages 712–723. OpenProceedings.org, 2014.

[10] aws.amazon.com. Announcing Cluster GPU Instances for Amazon EC2. http://amzn.to/1RrDflL, Nov. 2010. Last access: 09/02/16.

[11] C. Balkesen and N. Tatbul. Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams. In *Proceedings of the 8th International Workshop on Data Management for Sensor Networks*, DMSN '11, 2011.

[12] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental Sliding Window Analytics. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 61–72. ACM, 2014.

[13] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1093–1104. ACM, 2010.

[14] G. E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, 1990.

[15] S. Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[16] S. Breß and G. Saake. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *Proc. VLDB Endow.*, 6(12):1398–1403, Aug. 2013.

[17] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736. ACM, 2013.

[18] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. Zdonik. S-Store: A Streaming NewSQL System for Big Velocity Applications. *Proc. VLDB Endow.*, 7(13):1633–1636, Aug. 2014.

[19] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.*, 8(4):401–412, Dec. 2014.

[20] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668. ACM, 2003.

[21] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *SIGMOD Rec.*, 29(2):379–390, May 2000.

[22] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a Coupled CPU-GPU Architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11. IEEE Computer Society Press, 2012.

[23] X. Chen, C.-D. Lu, and K. Pattabiraman. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *Proceedings of the 25th International Symposium on Software Reliability Engineering*, ISSRE '14, pages 167–177. IEEE Computer Society Press, 2014.

[24] S. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. Walters. Heterogeneous Cloud Computing. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 378–385. IEEE Press, 2011.

[25] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive Stream Processing Using Dynamic Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13. ACM, 2014.

[26] feedzai.com. Modern Payment Fraud Prevention at Big Data Scale. http://bit.ly/1KCskD5, 2013. Last access: 09/02/16.

[27] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellJoin: A Parallel Stream Join Operator for the Cell Processor. *The VLDB Journal*, 18(2):501–519, Apr. 2009.

[28] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336. ACM, 2006.

[29] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[30] J. He, M. Lu, and B. He. Revisiting Co-processing for Hash

Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[31] J. He, S. Zhang, and B. He. In-cache Query Co-processing on Coupled CPU-GPU Architectures. *Proc. VLDB Endow.*, 8(4):329–340, Dec. 2014.

[32] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious Parallelism for In-memory Column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[33] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[34] Z. Jerzak and H. Ziekow. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 266–269. ACM, 2014.

[35] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, ICDE '03, pages 341–352. IEEE Press, 2003.

[36] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The HELLS-join: A Heterogeneous Stream Join for Extremely Large Windows. In *Proceedings of the 9th International Workshop on Data Management on New Hardware*, DaMoN '13, pages 2:1–2:7. ACM, 2013.

[37] T. Karnagel, M. Hille, M. Ludwig, D. Habich, W. Lehner, M. Heimel, and V. Markl. Demonstrating Efficient Query Processing in Heterogeneous Environments. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 693–696. ACM, 2014.

[38] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGrid '10, pages 94–103. IEEE Computer Society, 2010.

[39] S. J. Kazemitabar, U. Demiryurek, M. Ali, A. Akdogan, and C. Shahabi. Geospatial Stream Query Processing Using Microsoft SQL Server StreamInsight. *Proc. VLDB Endow.*, 3(1-2):1537–1540, Sept. 2010.

[40] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*, 2012.

[41] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[42] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming Algorithms for Biological Sequence Alignment on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 18(9):1270–1281, Sept. 2007.

[43] D. Lustig and M. Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture*, HPCA '13, pages 354–365. IEEE Computer Society, 2013.

[44] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, July 2004.

[45] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177. IEEE Computer Society, 2010.

[46] Novasparks™. NovaSparks Releases FPGA-based Feed Handler for OPRA. `http://bit.ly/1kGbQ0D`, Aug. 2014. Last access: 09/02/16.

[47] NVIDIA®. Quadro K5200 Data Sheet. `http://bit.ly/1O3D0Yx`. Last access: 09/02/16.

[48] H. Pirk, S. Manegold, and M. Kersten. Waste not... Efficient co-processing of relational data. In *Proceedings of the 30th IEEE International Conference on Data Engineering*, ICDE '14, pages 508–519. IEEE Press, 2014.

[49] Z. Shao. Real-time Analytics at Facebook. Presented at the *Fifth Conference on Extremely Large Databases*, XLDB5, Oct. 2011. `http://stanford.io/1HqxPmw`. Last access: 09/02/16.

[50] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General Incremental Sliding-window Aggregation. *Proc. VLDB Endow.*, 8(7):702–713, Feb. 2015.

[51] J. Teubner and R. Mueller. How Soccer Players Would Do Stream Joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 625–636. ACM, 2011.

[52] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156. ACM, 2014.

[53] J. Wilkes. More Google Cluster Data. Google Research Blog, `http://bit.ly/1A38mfR`, Nov. 2011. Last access: 09/02/16.

[54] wired.com. Google Erects Fake Brain With... Graphics Chips? `http://wrd.cm/1CyGIYQ`, May 2013. Last access: 09/02/16.

[55] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.*, 6(10):817–828, Aug. 2013.

[56] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438. ACM, 2013.

[57] Y. Zhang and F. Mueller. GStream: A General-Purpose Data Streaming Framework on GPU Clusters. In *Proceedings of the International Conference on Parallel Processing*, ICPP '11, pages 245–254. IEEE Press, 2011.

APPENDIX

A. BENCHMARK QUERIES

A.1 Cluster monitoring

```
-- Query 1
--
-- Input:  TaskEvents
--         long  timestamp
--         long  jobId
--         long  taskId
--         long  machineId
--         int   eventType
--         int   userId
--         int   category
--         int   priority
--         float cpu
--         float ram
--         float disk
--         int   constraints
-- Output: CPUusagePerCategory
```

```
--         long  timestamp
--         int   category
--         float totalCpu
--
select timestamp, category, sum(cpu) as totalCpu
from TaskEvents [range 60 slide 1]
group by category
--
-- Query 2
--
-- Input:  TaskEvents
-- Output: CPUusagePerJob
--         long  timestamp
--         long  jobId
--         float avgCpu
--
select timestamp, jobId, avg(cpu) as avgCpu
from TaskEvents [range 60 slide 1]
where eventType == 1
group by jobId
```

A.2 Smart grid

```
-- Query 1
--
-- Input:  SmartGridStr
--         long  timestamp
--         float value
--         int   property
--         int   plug
--         int   household
--         int   house
--        [int   padding]
-- Output: GlobalLoadStr
--         long  timestamp
--         float globalAvgLoad
--
select timestamp, AVG (value) as globalAvgLoad
from SmartGridStr [range 3600 slide 1]
--
-- Query 2
--
-- Input:  SmartGridStr
-- Output: LocalLoadStr
--         long  timestamp
--         int   plug,
--         int   household
--         int   house
--         float localAvgLoad
--
select timestamp, plug, household, house,
       AVG (value) as localAvgLoad
from SmartGridStr [range 3600 slide 1]
group by plug, household, house
--
-- Query 3
--
-- Input:  GlobalLoadStr, LocalLoadStr
-- Output: Outliers
--         long  timestamp
--         int   house
--         float count
--
(
select L.timestamp, L.plug, L.household,
       L.house
from LocalLoadStr  [range 1 slide 1] as L,
     GlobalLoadStr [range 1 slide 1] as G
 where L.house == G.house and
       L.localAvgLoad > G.globalAvgLoad
) as R
--
select timestamp, house, count(*)
from R
group by house
```

A.3 Linear Road Benchmark

```
-- Query 1
--
-- Input:  PosSpeedStr
--         long  timestamp
--         int   vehicle
--         float speed
--         int   highway
--         int   lane
--         int   direction
--         int   position
-- Output: SegSpeedStr
--         long  timestamp
--         int   vehicle
--         float speed
--         int   highway
--         int   lane
--         int   direction
--         int   segment
--
select timestamp, vehicle, speed,
       highway, lane, direction,
       (position/5280) as segment
from SegSpeedStr [range unbounded]
--
-- Query 2
--
-- Input:  SegSpeedStr
-- Output: VehicleSegEntryStr
--         long  timestamp
--         int   vehicle
--         float speed
--         int   highway
--         int   lane
--         int   direction
--         int   segment
--
select distinct
   L.timestamp, L.vehicle, L.speed, L.highway, L.lane,
   L.direction, L.segment
from SegSpeedStr [range 30 slide 1] as A,
     SegSpeedStr [partition by vehicle rows 1] as L
where A.vehicle == L.vehicle
--
-- Query 3
--
-- Input:  SegSpeedStr
-- Output: CongestedSegRel
--         long  timestamp
--         int   highway
--         int   direction
--         int   segment
--         float avgSpeed
--
select timestamp, highway, direction, segment,
       AVG(speed) as avgSpeed
from SegSpeedStr [range 300 slide 1]
group by highway, direction, segment
having avgSpeed < 40.0
--
-- Query 4
--
-- Input:  SegSpeedStr
-- Output: SegVolRel
--         long  timestamp
--         int   highway
--         int   direction
--         int   segment
--         float numVehicles
--
(
select timestamp, vehicle, highway, direction, segment,
       count(*)
from SegSpeedStr [range 30 slide 1]
group by highway, direction, segment, vehicle
) as R
--
select timestamp, highway, direction, segment,
       count(vehicle) as numVehicles
from R
group by highway, direction, segment
```