Demo: The SABER System for Window-Based Hybrid Stream Processing with GPGPUs

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

Heterogeneous architectures that combine multi-core CPUs with many-core GPGPUs have the potential to improve the performance of data-intensive stream processing applications. Yet, a stream processing engine must execute streaming SQL queries with sufficient data-parallelism to fully utilise the available heterogeneous processors, and decide how to use each processor in the most effective way. Addressing these challenges, we demonstrate 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 and employs an adaptive scheduling strategy to balance the load on the different types of processors. To hide data movement costs, SABER pipelines the transfer of stream data between CPU and GPGPU memory. In this paper, we review the design principles of SABER in terms of its hybrid stream processing model and its architecture for query execution. We also present a web front-end that monitors processing throughput.

CCS Concepts

•**Information systems → Data management systems;**

Keywords

Stream processing, heterogeneous hardware, GPGPUs

1. INTRODUCTION

Stream processing systems found wide-spread application in domains such as credit fraud detection [11], urban traffic management [6], and click stream analytics [3]. These systems process continuous streams of input data in an online manner, aiming at maximising processing *throughput* while staying within acceptable latency bounds. Stream processing is often based on a streaming relational model [4]: a stream is a potentially infinite sequence of relational tuples. Streaming SQL queries define *windows* (finite subsequences of a stream) and *operators*, such as projection, selection, aggregation and join, that are executed per window.

Heterogeneous architectures that combine multi-core CPUs with many-core GPGPUs have the potential to improve the performance of stream processing engines. In particular, GPGPUs implement a throughput-oriented architecture that, unlike traditional CPUs, targets embarrassingly parallel workloads. GPGPUs feature thousands of simple cores, following the single-instruction, multiple-data (SIMD) model: in each cycle, a GPGPU core executes the same operation on different data. The potential of GPGPUs for accelerating query processing has been shown for traditional relational database engines [14] and for specific streaming algorithms, e.g., joins [17] and sorting [12]. Yet, the design of a *general-purpose* relational stream processing engine that can transparently take advantage of GPGPUs has been an open challenge.

This demonstration presents SABER [16], a **hybrid relational stream processing engine** in Java that executes streaming SQL queries on both a multi-core CPU and a many-core GPGPU. SABER adopts a hybrid execution model in which query operators can utilise CPU cores and the GPGPU interchangeably. Specifically, SABER incorporates the following innovations:

Hybrid stream processing model. SABER translates a streaming SQL query to an operator graph, which is then bundled with a batch of stream data to form a query task. SABER's hybrid stream processing model features two levels of data parallelism: (i) tasks run in parallel across multiple CPU cores and the GPGPU, and (ii) a task running on the GPGPU is further parallelised across its many cores. Instead of relying on offline performance models to select the processor on which to run a query operator, SABER employs an adaptive *heterogeneous lookahead scheduling* strategy to balance the load on the different types of processors.

Window-aware task processing. SABER executes query tasks while supporting sliding window semantics. A dispatcher splits the stream into fixed-sized batches that include multiple fragments of windows processed jointly by a task. 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. The speed-up achieved by GPGPUs may be bound by the data movement cost over the PCI express (PCIe) bus. Against this background, SABER introduces a *five-stage pipelining* mechanism that interleaves data movement and task execution on the GPGPU: it maintains high utilisation of the PCIe bandwidth by pipelining the transfers of batches to and from the GPGPU with task execution. It also hides the memory latency originating from copying batches to/from Java heap memory.

In the remainder of this paper, §2 introduces the hybrid stream processing model of SABER before §3 describes its architecture. We then give an overview of the demonstration (§4), and finish with related work (§5) and conclusions (§6).

2. THE HYBRID STREAM PROCESSING MODEL OF SABER

Exploiting heterogeneous architectures for stream processing raises the question of *when to use GPGPUs for streaming SQL queries* and *how to use GPGPUs with streaming window semantics*. The speed-up achieved by GPGPUs depends on the type of computation: executing an 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. Furthermore, a GPGPU processes a discrete amount of data in parallel, so physical batches of stream data need to be constructed for efficient processing.

SABER sidesteps the problem of "when" to offload query operators to a heterogeneous processor with a *hybrid stream processing model*. In this 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.

Query tasks. To achieve that each query can be scheduled on any heterogeneous processor, 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 with n input streams, a query task consists of (i) an n-ary operator function and (ii) a sequence of n *stream batches*, i.e. n finite sequences of tuples, one per input stream. The *query task size* is defined as the sum of the data volumes of stream batches. It is a system parameter that specifies how much stream data a query task must process, thus determining the computational cost of executing a task. In practice, the query task size is chosen independently of the query workload, simply based on the properties of the engine implementation and its underlying hardware.

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. This makes window handling in the hybrid model fundamentally different from existing approaches to data-parallel stream processing, such as Spark Streaming [19]. By decoupling the parallelisation level (i.e. the query task) from the specifics of the query (i.e. the window definition), SABER can support queries over fine-grained windows (i.e. windows with small slides) with full data-parallelism.

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. This has consequences for the realisation of the operator function of a query task: it must be decomposed into a *fragment* function that is applied per fragment and an *assembly* function that assembles the window result from window fragment results. This decomposition is operator-specific, but, for many associative and commutative functions (e.g. aggregation functions for count or max), both functions correspond to the original operator 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. SABER's hybrid model captures this optimisation through a *batch* function. It is applied to a query task as a whole, i.e., sequences of window fragments. Applied to a query task, it yields a sequence of window fragment results, which correspond to applying the respective fragment function to each fragment in the task. Again, the implementation of the batch function is operator-specific.

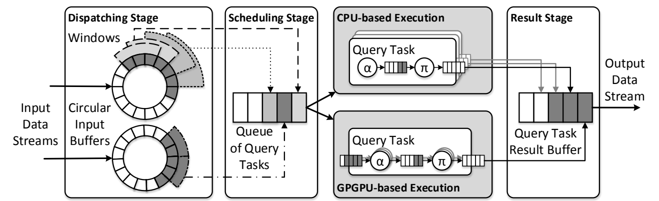


Figure 1: Overview of the SABER architecture

3. SABER ARCHITECTURE

SABER realises the above hybrid stream processing model with the architecture illustrated in Fig. 1. It consists of four stages that represent the lifecycle of a query task: from its creation, over to scheduling and execution, to the processing of the respective task results. The complete lifecycle of a query task is handled by a *worker thread*, a CPU thread that is bound to a physical core—one of them using the GPGPU for task execution. Below, we summarise the four stages of query processing in SABER.

Dispatching stage. Dispatching involves two steps: the storage of the incoming stream 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. 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 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, a query task is created.

Scheduling stage. Scheduling decides on the next task each processor should execute. SABER uses a *heterogeneous lookahead scheduling* (HLS) algorithm that achieves full utilisation of all heterogeneous processors, while accounting for the different performance characteristics of query operators. In essence, HLS tries to assign each task to the heterogeneous processor that, based on the 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. Making scheduling decisions based on the observed throughput per processor and query 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).

Execution stage. A query task is executed on a processor by evaluating the batch function on the input window fragments. A task executes either on one of the CPU cores or the GPGPU. To avoid any restriction of the performance of streaming operators by their memory accesses, SABER explicitly manages memory to avoid unnecessary data deserialisation and dynamic object creation.

To efficiently transfer data to and from the GPGPU, SABER employs 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 data movement operations in parallel to task execution: two CPU threads copy data from and to Java heap memory, and two GPGPU threads implement the data movement from and to GPGPU memory.

Result stage. The result stage reorders query task results that may arrive out-of-order due to their parallel execution. It also assembles window results from window fragment results by means of the assembly operator function. To reduce the synchronisation needed

```
-- Input: SegSpeedStr
--     long  timestamp
--     int   vehicle
--     float speed
--     int   highway
--     int   lane
--     int   direction
--     int   segment
select timestamp, highway, direction, segment,
       AVG(speed) as avgSpeed
from SegSpeedStr [range 300 slide 1]
group by highway, direction, segment
having avgSpeed < 40.0
```

Listing 1: Example streaming SQL query implemented in SABER

among the workers executing the query tasks, 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 circular 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.

4. DEMONSTRATION OVERVIEW

Streaming SQL queries. SABER supports the definition of streaming SQL queries with different types of windows and diverse operators. Queries may define count- or time-based windows. That is, the amount of enclosed data per window (the window size) and the difference between subsequent windows (the window slide) are either defined based on a fixed number of tuples or by a fixed timespan.

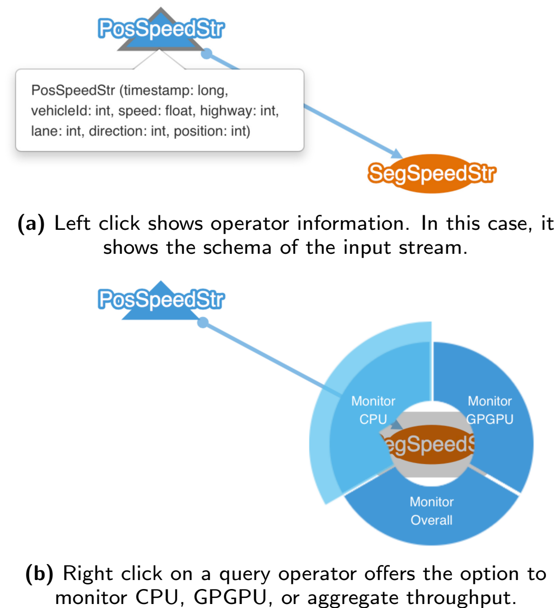
Furthermore, SABER supports all common relational operators, i.e. projection, selection, aggregation, and θ-join, each interpreting a window as a relation over which they are evaluated. For the aggregation operator, we also consider the use of `GROUP-BY` clauses. Operators in SABER may also be specified as *user-defined functions* (UDFs), which implement bespoke computation per window.

In Listing 1, we show one of the queries of the Linear Road Benchmark [5] implemented in SABER. It takes as input a stream of tuples that denote position events of vehicles on a driving on a highway segment. The query identifies highway segments for which the average speed of the respective vehicles in a sliding window of 5 minutes drops below a threshold.

Monitoring processing performance. SABER implements a RESTful API using Jetty to serve HTTP GET requests from its web frontend, the SABER *workbench*. When the browser loads the SABER workbench, it requests the dataflow graph of the query application that is currently running. Fig. 2 shows a dataflow graph that consists of a data source PosStreamStr and a query SegSpeedStr, the first query of the Linear Road Benchmark [5]. A left click on a graph node shows the SQL query statement or, in case of a data source, the input stream schema (Fig. 2a). A right click on a query displays the CPU, the GPGPU, or the aggregate throughput achieved by both processors for that query (Fig. 2b).

SABER monitors processing throughput of each query on a processor every 1 s. Measurements are stored in a fixed-length doubly linked list, one per query, configured to hold a snapshot of the last measurements: when the queue is full, for every new measurement added the oldest one is removed. If a user selects to display the throughput of a particular query operator, the SABER workbench sends periodic GET requests to flush the corresponding queue.

Fig. 3 shows the CPU, GPGPU, and hybrid throughput achieved by SABER when running a selection query over a synthetic input stream. A key feature of SABER is its ability to change the utilisation of each processor to maximise the aggregate processing throughput. In Fig. 3, the query workload changes such that the selectivity of the



(a) Left click shows operator information. In this case, it shows the schema of the input stream.

(b) Right click on a query operator offers the option to monitor CPU, GPGPU, or aggregate throughput.

Figure 2: Query 1 of the Linear Road Benchmark [5], as illustrated in the SABER workbench

query varies from 1% to 50%, making the selection predicate more expensive to evaluate. Before the change, the CPU alone can process stream data at approximately 6 GB/s, SABER's peak throughput. The GPGPU is this case is assigned approximately $1/20^{\text{th}}$ of the tasks, based on the switch threshold of the HLS algorithm. After the change in the input data distribution, however, the CPU struggles to handle the workload alone. At that point, HLS schedules more tasks on the GPGPU in order to sustain the overall throughput of the system just below 5 GB/s.

5. RELATED WORK

Stream processing engines for centralised infrastructures have long been limited to single-core execution. Only recently, systems such as Oracle CEP [1] and Microsoft StreamInsight [15] support multi-core architectures. Yet, this support comes at the expense of weakening stream ordering guarantees in the presence of window-based queries. Research prototypes such as S-Store [9] and Trill [10] 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, bu t it does not support hybrid query execution.

In recent years, many systems for data-parallel processing of streaming queries on a cluster of nodes have been developed. However, systems such as Storm [18] and SEEP [8] do not respect window semantics by default. Millwheel [2] provides strong window semantics, but it does not perform parallel computation on windows and instead assumes partitioned input streams. Spark Streaming [19] has a batched-stream model, and permits window definitions over this model, thus creating dependencies between window semantics, throughput and latency. Unlike Streaming Spark, SABER decouples window semantics from system performance.

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 [7, 13], 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 [20] executes streaming applications on GPGPU

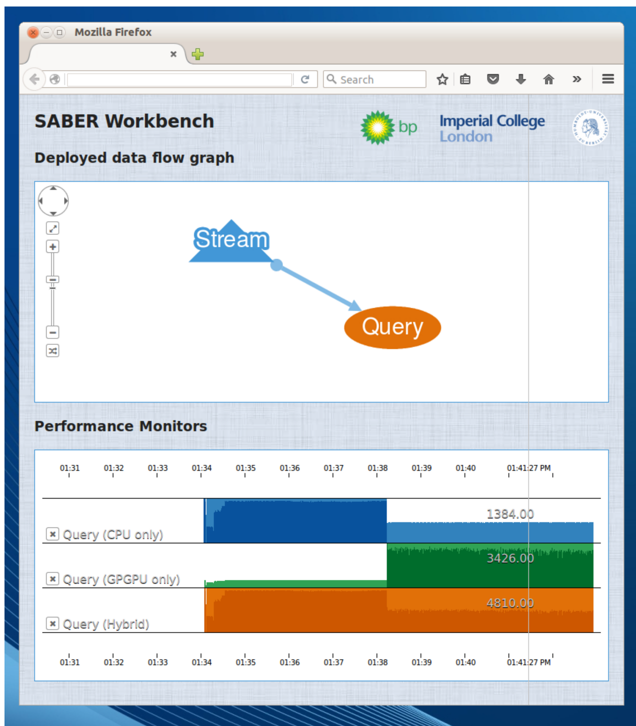


Figure 3: Front-end for monitoring the performance of SABER

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.

6. CONCLUSION

This work demonstrated SABER, a hybrid high-performance relational stream processing engine for CPUs and GPGPUs. SABER's major innovations are a hybrid execution model for window-based streaming SQL queries, an adaptive scheduling strategy to balance the load on heterogeneous processors, and pipelined data transfer between CPU and GPGPU memory. Specifically, we demonstrated the functionality of SABER in terms of supporting streaming SQL queries and presented a Web front-end for monitoring the performance of the engine.

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; a PhD CASE Award by EPSRC/BAE Systems; and BP UK.

7. REFERENCES

[1] Oracle® Stream Explorer. http://bit.ly/1L6tKz3. Last access: 30/05/16.

[2] 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, 2013.

[3] 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, 2009.

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

[5] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In VLDB '04, pages 480–491. Morgan Kaufmann, 2004.

[6] 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 EDBT '14, pages 712–723. OpenProceedings.org, 2014.

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

[8] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In SIGMOD '13, pages 725–736. ACM, 2013.

[9] 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, 2014.

[10] 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, 2014.

[11] feedzai.com. Modern Payment Fraud Prevention at Big Data Scale. http://bit.ly/1KCskD5, 2013. Last access: 30/05/16.

[12] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In SIGMOD '06, pages 325–336. ACM, 2006.

[13] 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, 2013.

[14] 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, 2013.

[15] 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, 2010.

[16] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. SABER: Window-based hybrid stream processing for heterogeneous architectures. In SIGMOD '16, 2016. To appear.

[17] J. Teubner and R. Mueller. How Soccer Players Would Do Stream Joins. In SIGMOD '11, pages 625–636. ACM, 2011.

[18] 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 SIGMOD '14, pages 147–156. ACM, 2014.

[19] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In SOSP '13, pages 423–438. ACM, 2013.

[20] Y. Zhang and F. Mueller. GStream: A General-Purpose Data Streaming Framework on GPU Clusters. In ICPP '11, pages 245–254. IEEE Press, 2011.