Grizzly: Efficient Stream Processing Through Adaptive Query Compilation

Philipp M. Grulich[1]   Sebastian Breß[1]   Steffen Zeuch[1,2]   Jonas Traub[1]

Janis von Bleichert[1]   Zongxiong Chen[2]   Tilmann Rabl[3]   Volker Markl[1,2]

[1]Technische Universitat Berlin   [2]DFKI GmbH   [3]HPI, University of Potsdam

ABSTRACT

Stream Processing Engines (SPEs) execute long-running queries on unbounded data streams. They follow an interpretation-based processing model and do not perform runtime optimizations. This limits the utilization of modern hardware and neglects changing data characteristics at runtime.

In this paper, we present *Grizzly*, a novel adaptive query compilation-based SPE, to enable highly efficient query execution. We extend query compilation and task-based parallelization for the unique requirements of stream processing and apply adaptive compilation to enable runtime re-optimizations. The combination of light-weight statistic gathering with just-in-time compilation enables Grizzly to adjust to changing data-characteristics dynamically at runtime. Our experiments show that Grizzly outperforms state-of-the-art SPEs by up to an order of magnitude in throughput.

ACM Reference Format:

Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318464.3389739

1 INTRODUCTION

Over the last decades, the requirements of data processing changed significantly. Real-time analytics require the execution of long-running queries over unbounded, continuously changing, high-velocity data streams. Common SPEs
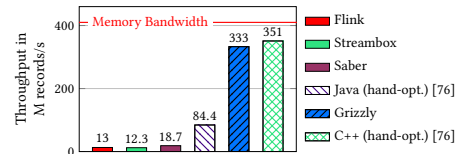
Figure 1: Yahoo! Streaming Benchmark (8 Threads).

such as Flink [15] and Storm [66] scale-out executions to achieve high throughput and low-latency. However, recent research revealed that these SPEs do not fully utilize modern hardware [76, 78]. The authors identified three main reasons for this. First, these systems cause many instruction cache misses because they use an interpretation-based processing model. Second, they cause many data cache misses because they rely on managed runtimes. Third, they utilize sub-optimal parallelization strategies on single nodes because they optimize for a scale-out environment. By eliminating these bottlenecks in hand-written implementations of stream processing queries, Zeuch et al. [76] showed a significant performance improvement is possible. However, to hand-code query plans is impractical and cumbersome in practice. For database systems, Neumann [55] introduced query compilation as a technique to achieve the performance of hand-written code for general query processing. However, no state-of-the-art SPE exploits query compilation to achieve similar performance improvements for streaming workloads.

In this paper, we introduce Grizzly[1], the first adaptive, query compilation approach for stream processing on multi-core systems. Grizzly combines the generality and ease-of-use functionality of SPEs with the efficient hardware utilization of hand-written code. To reach this goal, we tackle three fundamental challenges: First, the semantics of stream processing are fundamentally different from relational algebra. Data streams are conceptually unbound and need to be discretized into finite windows. Windowing semantics are extremely diverse and cover different window types (e.g., tumbling and sliding windows), window measures (e.g., time-based and count-based windows), and window functions (e.g., aggregations). The cyclic control flow between these components makes it hard to apply state-of-the-art

---

[1]https://github.com/TU-Berlin-DIMA/grizzly-prototype

P. M. Grulich, S. Breß, S. Zeuch, J. Traub, J. v. Bleichert, Z. Chen, T. Rabl, and V. Markl

query compilation approaches directly. More specifically, Grizzly extends classic query compilation by supporting cyclic compile-time dependencies between the windowing components (assignment, trigger, and functions). Second, stream processing queries are inherently long-running, while the input stream constantly changes. As a consequence, the optimal plan and code efficiency changes over time too. To handle this, Grizzly establishes a feedback loop between code-generation and execution. Our key idea is to use query compilation to inject low-overhead profiling code into the compiled query. The instrumented query collects profiling information, which we use to optimize the query at run time adaptively. Third, in contrast to relational operators, stream processing requires ordering between records. This introduces additional challenges for the concurrent processing of state-full operations (e.g., aggregations). SPEs, such as Flink, apply key-by partitioning to mitigate this problem. In contrast, we apply task-based parallelism to utilize multi-core systems efficiently. To this end, Grizzly generates specialized code to address the ordering requirements of a particular query and to take the underlying hardware into account.

By tackling these challenges, Grizzly closes the gap between state-of-the-art SPEs and the performance of hand-written code. In Figure 1, we compare the performance of Grizzly to a scale-out SPE (Flink[15]), two scale-up SPEs (Saber [45], Streambox [53]), and two hand-optimized implementations of the Yahoo! Streaming Benchmark [21]. Only Grizzly and the hand-written C++ implementation fully utilize the available hardware. Grizzly outperforms state-of-the-art SPEs by up to an order of magnitude without losing generality. In summary, our contributions are as follows:

(1) We present an adaptive query compilation approach for stream processing that generates efficient code.
(2) We extend query compilation to support common window types, window measures, and window functions.
(3) We introduce adaptive optimizations to react to changing data characteristics.
(4) We utilize order-preserving task-based parallelization and introduce light-weight coordination among threads.
(5) We demonstrate Grizzly's performance in comparison to state-of-the-art SPEs on diverse workloads.

The remainder of this paper is structured as follows. First, we discuss foundational background (Sec. 2). Second, we introduce the architecture of Grizzly (Sec. 3), its code generation approach (Sec. 4), its parallelization technique (Sec. 5), and its adaptive optimizations (Sec. 6). Finally, we evaluate Grizzly (Sec. 7) and discuss related work (Sec. 8).

2 BACKGROUND

In this section, we provide an overview of window semantics (Sec. 2.1) and introduce query compilation (Sec. 2.2).

2.1 Window Semantics

Stream processing has been formally defined by multiple authors [11, 16, 45]. Following Carbone et al. [16] we define a data stream $\bar{s}$ as a sequence of records and denote, $s_i = \bar{s}(i)$ as the *ith* element in $\bar{s}$ and $\bar{s}([a, b]) = \bar{s}(R) = \{s_i | i \in R\}$ as a sub-stream of $\bar{s}$. The window operator discretizes a data stream $\bar{s}$ into a sequence of potentially overlapping windows $w_i = \bar{s}([b_i, e_i])$. Windows are characterized by a window type, a window measure, and a window function [68].

Window Types. The window type is formally defined by an assignment function $f_a(s_i) \rightarrow w_i$ that assigns a record s_i to a window w_i. Common window types are tumbling, sliding, and session windows [68]. Tumbling and sliding windows discretize a stream into windows of fixed length l. Additionally, sliding windows define a slide step l_s that declare how often new windows start. Thus, records are assigned to multiple concurrent overlaps sliding windows if $l_s < l$. In contrast, session windows end if no record is received for a time l_g (session gap) after a period of activity.

Window Measures. The window measure defines the progress of windows. Common window measures are time and count [11]. Time-based windows utilize a monotonic increasing timestamp ts and trigger as soon as the time passes the window end $ts > w_i.e$. In contrast, the length l of count-based windows corresponds to the number of assigned records. Thus, a count window ends if $i > w_i.e$. Note for keyed aggregations, time-based windows trigger for all keys at the same time, but the trigger decision of count-based windows has to be managed per key.

Window-Functions. Window functions execute arbitrary computations on assigned records. For aggregation functions, we differentiate between decomposable and non-decomposable functions as proposed by Jesus et al. [40]. Decomposable aggregate functions (e.g., *sum*, *avg*) are computed incrementally; thus, only a partial aggregate has to be stored. In contrast, non-decomposable aggregation functions (e.g., holistic functions), require access to all records of a window.

2.2 Query Compilation

Over the last decade, query compilation for data-at-rest processing was extensively studied [46, 55, 61] and implemented in several systems [44, 46, 55]. To generate code for a query, many of these systems apply the produce/consume [55] model. In this approach, a query compiler segments a query plan into pipelines whenever a materialization of intermediate results is required (e.g., for *Aggregation* or *Join* operators). All operations inside a pipeline are fused to one combined operator that performs a single pass over the data such that data stays in CPU registers [55]. To implement the produce/consume model, the compiler requires each operator to implement two functions. First, the produce function is called

on the root operator and navigates the query plan from the root to the leaves and segments the query in pipelines. Second, the consume function is called from the leaf nodes, navigates to the root node, and generates the code for each pipeline. This results in a very compact code fragment which combines the processing of all pipeline operators.

3 GRIZZLY

In this section, we introduce Grizzly, our novel adaptive, compilation-based SPE. Grizzly's primary goal is to provide a high-level query interface for end-users while at the same time achieving the performance of hand-optimized code. In the remainder of this section, we discuss the major challenges of compilation-based SPEs (Sec. 3.1), present how Grizzly's core principles address them (Sec 3.2) and explain Grizzly's execution model (Sec. 3.3).

3.1 Challenges for compilation-based SPEs

Similar to query compilation for data-at-rest, a compilation-based SPE, segments queries into multiple pipelines and fuses operators within pipelines. However, stream processing workloads introduces several new challenges.

Challenge 1: Stream processing semantics. To the best of our knowledge, there is no SPE that is able to fuse stream processing queries involving windowing. The main challenges are three-fold. First, the window triggering depends on the window assignment and is order-sensitive. Second, the window function needs to be performed after the windowing, but defines the state that needs to be stored in windows. Third, triggering involves a final aggregation step (e.g., to compute the average). The cyclic control flow between these three tasks makes it hard to apply state-of-the-art query compilation techniques to an SPE because they assume only linear compile-time dependencies between operators.

Challenge 2: Order preserving semantics. In contrast to relational algebra, the outcome of stream processing operators depends on the order of records in the data stream. Thus, data-parallel execution requires coordination among processing threads before the next pipeline can process window results. A compilation-based SPE has to take this requirement into account during code generation. As a result, a compilation-based SPE has to adjust the coordination among threads depending on the query to ensure correct processing results while enabling efficient processing.

Challenge 3: Changing data characteristics. Stream processing queries are deployed once and executed for a long time, while the input stream may change. In particular, they may face unpredictable changes in the data characteristics at runtime, e.g., a changing number of distinct values or a changing data distribution of keys. As a consequence, the efficiency of generated code may change over
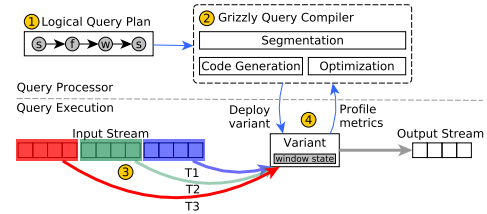

Figure 2: Query Execution Workflow in Grizzly.

time. Thus, a compilation-based SPE has to re-evaluate the applied optimizations and, if required, generate new code during runtime.

3.2 Core Principles of Grizzly

Grizzly addresses the challenges introduced in Section 3.1, by applying query compilation, enabling task-based parallelization, and adaptively optimizing the generated code with regards to hardware and data characteristics.

Query Compilation. Grizzly introduces query compilation for stream processing and handles the complexity of windowing. Within pipelines, Grizzly fuses operations to compact code fragments and performs all operations of a pipeline in one single pass over a chunk of input records without invoking functions. Thus, data remains in CPU registers as long as possible without loading records repeatedly. To improve data locality in contrast to managed run-times, Grizzly avoids serialization and accesses all data via raw memory pointer. As a result, query compilation in Grizzly increases code and data locality significantly.

Order preserving task-based parallelization. To exploit multi-core CPUs efficiently, Grizzly executes pipelines concurrently in a task-based fashion on a global state. This eliminates the overhead of data pre-partitioning and state merging. However, it requires coordination between threads to fulfill the order requirements of stream processing. Grizzly addresses these by introducing a light-weight, lock-free window-processing approach based on atomics.

Adaptive optimizations. Grizzly introduces a feedback loop between code-generation and query execution to exploit dynamic workload characteristics. Grizzly continuously monitors performance characteristics, detects changes, and generates new code variants. As a result, Grizzly performs speculative optimizations and assumptions about the incoming data. If an assumption is invalidated, Grizzly re-optimizes a code variant. To reduce the performance overhead, Grizzly combines light-weight but coarse-grained performance counters with fine-grained code instrumentalization.

3.3 Compilation-based Query Execution

In Figure 2, we present the architecture of Grizzly's compilation-based query execution model, which consists of four phases. From the logical query plan ① to the continuous adaption to changing data characteristics ④.

3.3.1   Logical Query Plan. In the first phase ①, Grizzly offers a high-level Flink-like API and translates each query to a logical query plan. This plan contains a chain of operators that consumes a stream with a static source schema. Grizzly supports traditional relational operators, e.g., selection and map, and stream processing specific operators for windowing. Window definitions consist of a window type, a window measure, and a window function, as introduced in Section 2.1. Furthermore, Grizzly supports global windows that create one aggregate over the whole stream and keyed windows that created partitioned aggregations per key. Based on these operators, Grizzly supports common stream processing queries.

3.3.2   Query Compiler. In the second phase ②, Grizzly segments the logical query plan into pipelines, performs optimizations, and generates code for each pipeline.

Segmentation. Query compilers for data-at-rest fuse operators until they reach a *pipeline-breaker*, which requires a full materialization of intermediate results (e.g., joins or aggregations). However, the unbounded nature of data streams prevents the full materialization of intermediate results. To this end, Grizzly separates pipelines at operators that require partial materialization, similar to soft-pipeline-breakers [76]. In particular, non-blocking operators (e.g., map or filter) are fused. In contrast, all blocking operations in stream processing are computed over windows (e.g., aggregations or joins) and terminate pipelines. Thus the support of windowed operations is crucial for a compilation-based SPE.

Optimization. After query segmentation, Grizzly optimizes the individual pipelines. To this end, Grizzly exploits static information, e.g., the hardware configuration, as well as dynamic data characteristics. To collect data characteristics, Grizzly introduces fine-grained instrumentation into the generated code. This enables Grizzly to derive assumptions about the workload, e.g., predicate selectivity and the distributions of field values. Based on these assumptions, Grizzly chooses particular physical operators.

Code-Generation. In the last step, Grizzly translates each physical pipeline to C++ code and compiles it to an executable code variant. Note that all variants of the same pipeline are semantically equivalent, but execute different instructions and access different data-structures. For code generation, Grizzly follows the produce/consume model and extends it with support for rich stream processing semantics. In particular, we consider code generation and operator fusion for the window operator.

3.3.3   Execution. In the third phase ③, Grizzly executes the generated pipeline variant. Each variant defines an open and close function to manage the state of the variant. Depending on the physical operators, state is completely pre-allocated or dynamically allocate during execution. For the input stream,
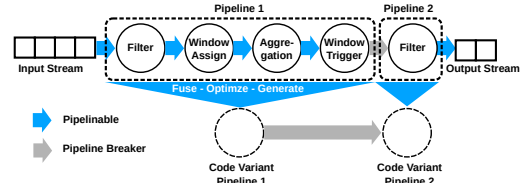

Figure 3: Compilation stream processing query.

Grizzly exploits the fact that input records physically arrive in batches over the network and schedules each batch as a task for an individual thread to utilize multi-core CPUs. Thus, pipelines and their associated state are accessed concurrently by multiple threads. This introduces challenges for window processing, as all threads have to pass the window-end before one thread outputs the result. To this end, Grizzly introduces a lock-free data structure, such that multiple threads can concurrently process a window without starvation.

3.3.4   Profiling & Adaptive Optimization. In the final phase ④, Grizzly continuously collects profiling information and re-optimizes the query in two steps. During query execution, Grizzly collects hardware performance counters, e.g., number of cache misses, to detect changing data characteristics. Hardware performance counters have a negligible performance impact [23, 77], but give a coarse-grained intuition about the evolution of data-characteristics. If the collected counters indicate a change, Grizzly collects more fine-grained profiling information, via code instrumentation. Based on this information, Grizzly re-optimizes the query and generates a new code variant.

4   QUERY COMPILATION

In this section, we detail Grizzly's query compilation approach and address the special challenges of stream processing. In particular, we focus on window aggregations as they are the primary operator requiring materialization and consequently break pipelines. Figure 3, illustrates how Grizzly segments an example query in two pipelines. Each pipeline begins with an arbitrary number of non-blocking pipeline operators (e.g., filter). Finally, each pipeline is terminated by the window operator, which Grizzly performs in three steps. First, the *window assigner* assigns input records, depending on the window type, to one or more corresponding windows. Second, the *window aggregator* updates the window aggregate. Third, the *window trigger* checks if an active window is complete and invokes the next pipeline to forward the window result. As a result, Grizzly supports a diverse set of window characteristics, which require specialized code generation for all windowing aspects, e.g., assignment, aggregation, and trigger. In Figure 4(a), we present a mapping of a generic query plan to an abstract code template. Note that we use the templates only for representation reasons, internally each physical operator produces C++ code depending
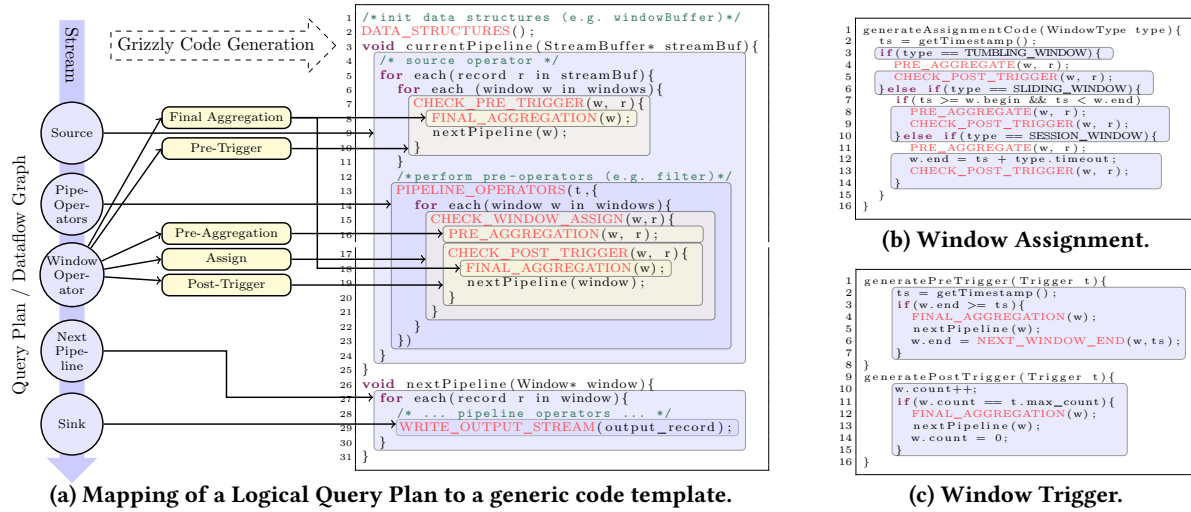
(a) Mapping of a Logical Query Plan to a generic code template.

(b) Window Assignment.

(c) Window Trigger.

Figure 4: Overview of Code Generation.

on its particular properties. As shown, a stream processing query may consist of four types of operators: source, pipeline operators, window operators, and sinks. In the following, we first provide an overview of the operators that are supported by Grizzly (Sec. 4.1). After that, we describe the code generation for the windowing operator as the main building block in streaming queries (Sec. 4.2).

4.1 Operator Overview

In this section, we discuss the individual operators of Figure 4(a) and illustrate how Grizzly generates code for them.

Source Operator. The input stream arrives as a sequence of input buffers containing records. Each pipeline receives one input buffer at a time (Line 3) and the source operator iterates in a tight for loop over all records (Line 5). To avoid the deserialization of data from the input buffer, Grizzly casts the data from the raw buffer directly into complex event types. Then, the loop body executes all fused pipeline operators and ends with the window operator.

Pipeline-Operators. Pipeline-operators apply arbitrary non-blocking computation (e.g., filters, maps). Consequently, pipeline-operators could generate arbitrary output records per input record. Thus, all succeeding operations (e.g., window assignment and triggers) must be nested inside the pipeline-operators (Line 13).

Window Operator. Grizzly divides the window operator into three sub operators: assigner, aggregation, and trigger.

Assigner. During window assignment, Grizzly determines the target windows for the current record. The code iterates over all active windows and assigns the current record to its corresponding window(s) (Line 15).

Aggregation. After assigning a record to a window, Grizzly updates the window aggregate. Depending on the window

function, Grizzly pre-aggregates records to minimize memory consumption (Line 16). After the window is triggered, Grizzly computes the final aggregate (Line 8 and Line 18).

Trigger. Depending on the window measure (count-based or time-based), it is required to perform the trigger check before (Line 7) or after the window assignment (Line 17).

Next-Pipeline. After triggering a window, the next pipeline starts processing window results (Line 26). The next pipeline can again contain arbitrary pipeline operators and ends with a window operator or a sink. As a result, Grizzly supports queries with multiple windows.

Sink Operator. The sink operator terminates a pipeline and writes records to an output stream (Line 29).

4.2 Window Operator

In this section, we discuss window operator-specific query compilation aspects. To this end, we present the code generation approach for window assignment (Sec. 4.2.1), window aggregation (Sec. 4.2.2), and the window trigger (Sec. 4.2.3). Finally, we discuss the handling of window joins (Sec. 4.2.4).

4.2.1 Window Assignment. The window assigner maps incoming records to windows. To this end, Grizzly keeps track of active windows and generates specialized code depending on the window type, illustrated in Figure 4(b). To keep track of active windows, Grizzly stores metadata for each window (e.g., start and end timestamps) in a compact array. During window assignment, Grizzly checks all windows and assigns the record to a window aggregate if the assignment condition is true. Depending on the window type, Grizzly generates different code (blue background for generated code). For tumbling and session windows, each record belongs to exactly one window (Line 3 and Line 10). In contrast, for sliding windows the generated code iterates over all active windows

(Line 6) and selects matching windows based on the current timestamp (`ts`) (Line 7). In contrast, session windows expand with each assigned record and Grizzly shifts the window end if a record is assigned (Line 12).

4.2.2 Window Aggregation. After assigning records to windows, Grizzly adds the record to the window aggregate. Grizzly differentiates between decomposable (e.g., *sum*, *avg*) and non-decomposable (e.g., *median*) aggregation functions as introduced in Section 2.1. For non-decomposable aggregation functions, Grizzly stores all assigned records in a separate window buffer and computes the final aggregation after the window is triggered. For decomposable aggregation functions, Grizzly computes the window aggregate incrementally. Thus, Grizzly only stores a partial aggregate instead of all assigned records, which reduces memory requirements. Furthermore, primitive partial aggregates can be updated much more efficiently using atomic operations. For keyed (grouped) aggregations, Grizzly maintains a partial aggregate per key.

4.2.3 Window Trigger. The window trigger finalizes windows and passes them to the next pipeline. Grizzly evaluates trigger conditions before the processing of a record (pre-triggers) or after the window aggregation (post-triggers). We illustrate the code generation algorithm in Figure 4(c).

Pre-Trigger. The pre-trigger checks active windows before processing the current record. This is necessary to support time-based window measures. Time triggers only depend on the progress of time for the trigger decision and are independent of the individual record. Thus, time-based triggers replace the CHECK_PRE_TRIGGER macro in the generic code template (see Figure 4(a), Line 7). For time-based triggers, the generated code compares the current timestamp `ts` to the end time of each active window (Line 3). If the window end timestamp is passed, the window is triggered. In this case, Grizzly computes the final window aggregate (Line 4) and calls the next pipeline to process the window result (Line 5). Finally, Grizzly clears the window state, calculates a new window end timestamp, and updates the window metadata (Line 6). Note that an additional trigger is necessary if the arrival rate of new records is too slow to guarantee a constant evaluation of the trigger function.

Post-Trigger. The post-trigger is executed after assigning a record to a window. It replaces the CHECK_POST _TRIGGER macro in the generic code template (see. Figure 4(a), Line 17). The post-trigger only evaluates the assigned window instead of all active windows. Post-triggers are necessary to support count-based windows, which directly trigger a window after the last record is assigned. In contrast, a count trigger maintains a counter to keep track of the number of assigned records to each active window (Line 10). If the number of items has reached the maximal window count (Line 11), the trigger calculates the final aggregate (Line 12)

and invokes the next pipeline (Line 13). Finally, Grizzly clears the window state and sets the window count to zero (Line 14).

4.2.4 Windowed Join. Grizzly supports windowed equal joins following the semantics of Flink [15]. For each input stream, Grizzly generates one code pipeline that maintains an intermediate join table. Grizzly reuses the window trigger code to discard the intermediate state as soon as the window ends. During execution, each pipeline concurrently assigns records to its local join table and probes the record to the join table of the other join side. Consequently, the stream join is fully pipelined and non-blocking.

5 PARALLELIZATION

To utilize modern multi-core processors efficiently, Grizzly applies data-centric parallelization. This paradigm is reflected in both the classic exchange operator [35] and morsel-based operators [48]. Grizzly extends these ideas and introduces light-weight coordination primitives to address the unique ordering requirements of stream processing. During runtime, Grizzly creates tasks for each incoming buffer and its target processing pipeline. Worker threads execute the pipeline and operate on a shared global state, e.g., for window aggregations. This approach eliminates the data shuffling step of systems like Flink and provides robustness for skewed key distributions and heavy hitters. In the remainder of this section, we present how Grizzly coordinates window processing to address the order semantics of stream processing (Sec. 5.1) and how it specializes generated code with regards to NUMA hardware (Sec. 5.2).

5.1 Lock-Free Window Processing

In general, a dynamic, task-based parallelization can lead to wrong processing results for streaming queries. Thus, Grizzly has to prevent that windows are passed to the next pipeline, while other threads still assign records to them. A naïve approach would introduce a barrier at window ends to synchronize all processing threads. However, this limit performance due to the introduced waiting time. To overcome this limitation, we introduce a lock-free window processing technique that allows threads to process different windows concurrently. In particular, Grizzly maintains multiple window-aggregates in a ring buffer (depending on the window type), similar to the technique proposed by Zeuch et al. [76]. Furthermore, each thread maintains a pointer to its current window-aggregate and the value of the next window end. This technique enables Grizzly to support important properties. First, every thread can decide independently to which window it assigns incoming records. Second, only the last thread that modifies a window creates the final window aggregate and invokes the next pipeline.
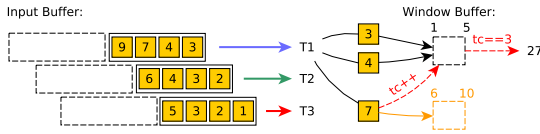
Figure 5: Example for Lock-Free-Window Trigger.

Figure 5, illustrates an example of Grizzly's log-free window implementation for time-based windows. Each thread processes its input buffer and checks per record if the window should trigger. If the window end is reached (at record ⑦ in Figure 5) the thread triggers the window *locally*. To this end, the thread calculates the next window end and shifts its current window pointer to the next position in the window-buffer. After that, the thread will assign all succeeding records to the next window-aggregate. In addition, each thread increments atomically a global trigger counter `tc`. If `tc` is equal to the degree of parallelism (tc==3 in Figure 5 it is guaranteed that all threads have triggered the window locally, and no thread will modify the window anymore. In this case, Grizzly creates the final window aggregate and invokes the next pipeline.

5.2 NUMA-aware Stream Processing

Research in the area of multi-core query execution shows that its crucial to take NUMA effects into account to enable scalability across multiple CPU sockets [42, 48]. Especially data accesses across NUMA regions reduce bandwidth by 2x [50]. In Grizzly, we minimize the inter-NUMA node communication and specialize the code generation to the underlying NUMA configuration. During query compilation, Grizzly detects the NUMA configuration and deploys a two-phase strategy for window aggregations. In the first phase, processing threads pre-aggregate values into a hash-map inside the local NUMA region. In the second phase, Grizzly merges the aggregates of the local states at the window end. During execution, Grizzly pins all processing threads to a specific NUMA region and only process local input buffers. Overall, this design reduces cross-numa communication to a minimum and enables efficient sharing inside one socket.

6 ADAPTIVE QUERY OPTIMIZATION

Research in the area of adaptive and progressive optimization demonstrates that the reaction to changing data characteristics improves performance significantly [8]. This specifically affects streaming queries, which are commonly deployed once and run virtually forever. In Grizzly, we detect and react to changing data characteristics at runtime and perform adaptive optimizations using JIT compilation. In the remainder of this section, we detail Grizzly's adaptive query compilation approach (Sec. 6.1) and present three optimizations that exploit specific data characteristics (Sec. 6.2).

6.1 Adaptive Query Compilation

Grizzly follows an explore/exploit approach to enable adaptive optimizations. At run time, Grizzly continuously performs optimization and deoptimization [31, 38]. Depending on assumptions about the workload (e.g., data- or hardware-characteristics), Grizzly generates specialized code variants. If assumptions become invalid, Grizzly deoptimizes and migrates back to a generic code variant. In the remainder of this section, we detail the individual steps of this process.

6.1.1 Execution Stages. The following three execution stages reflect the adaptive compilation process.

First Stage: Generic Execution. In the first stage, Grizzly executes a generic code variant and performs static optimization. For instance, Grizzly utilizes knowledge about the data schema to optimize comparisons to constant values.

Second Stage: Instrumented Execution. In the second stage, Grizzly introduces code instrumentation to collect fine-grained data-characteristics. Thus, each operator can generate arbitrary profiling instructions to track statistics (e.g., predicate selectivity, or the domain of a value). To reduce overhead, Grizzly applies sampling by executing profiling code only with a subset of threads and on a subset of records.

Third Stage: Optimized Execution. In the third stage, Grizzly utilizes the profiling information to make assumptions about the underlying data characteristics. Based on this, Grizzly performs speculative optimizations and specializes code as well as data structures.

6.1.2 Deoptimization. Deoptimization migrates from the optimized code variant back to the generic one. The causes of this are two-fold. First, during execution, Grizzly detects that an assumption is violated. For instance, if a key exceeds the assumed range (*assuming $x < 5$ but actualy $x = 10$*). In this case, the current processing thread continuous with the generic code variant. Second, Grizzly continuously monitors hardware performance counters to identify changes in data-characteristics, e.g., number of cache misses. If Grizzly detects a change, it schedules the deoptimization of the current code variant. If the frequency of deoptimizations is low, Grizzly directly migrates to stage two.

6.1.3 Variant Migration. For the migration between code variants, Grizzly ensures correct query results while minimizing processing overhead. To this end, Grizzly lazily invalidates code variants such that multiple threads can operate on different variants concurrently. All processing threads determine the switch individually and switch to the next variant after the current task. If all threads have discarded the old variant, Grizzly triggers state migration. In the case of windows, this requires the merging of a specialized state representation with the generic representation of the same

state. Furthermore, to ensure correctness, Grizzly triggers no windows before the migration is completed.

6.2 Adaptive Optimizations

In the following, we discuss three examples of adaptive optimization implemented in Grizzly. Beyond this, Grizzly's adaptive optimization approach is able to detect and react to a wide range of different characteristics (e.g., ingestion rate, value distribution, selectivity), and to perform a wide range of optimizations (e.g., operator re-order, algorithm selection, data-structure specialization).

6.2.1 Exploiting Predicate Selectivity. Optimizing selection operators has been studied extensively in the database [13, 14, 26, 30, 62, 70, 75] as well as the compiler community [6, 7]. In Grizzly, we utilize profiling information to determine the optimal order of selection predicates inside a query plan. In particular, conjunctions over multiple predicates benefit if the most selective predicate is evaluated first, as the CPU can skip the evaluation of all other branches. Additionally, predicates with a selectivity of around 50% cause miss-prediction and introduce a high-performance overhead. During *instrumentalization*, Grizzly generates one counter per predicate to measures the individual selectivity. In comparison to measuring the combined operator selectivity with performance counters [77], this allows to choose the optimal predicate order directly. During *optimized execution*, Grizzly executes the optimized code variant and monitors the number of mis-predictions for taken and not taken branches by applying the cost model of Zeuch et al. [75]. An increasing number of mispredictions indicates that the selectivity of a predicate changed and that the current predicate order becomes inefficient. Thus, Grizzly initiates a new profiling phase to re-optimize the predicate order.

6.2.2 Exploiting Value Ranges. In the general case, Grizzly maintains window aggregates in an Intel TBB concurrent hash-map [39]. This hash-map accepts any data type for keys and values and grows dynamically with the number of keys. As a result, Grizzly supports any number of input keys as long as the hash-map fits into memory. However, this flexibility introduces a substantial overhead [51]. To mitigate this overhead, Grizzly speculates on the value range. During instrumentalization, Grizzly injects code to identify the maximal and minimal key value that is inserted into the map. During *optimized execution*, Grizzly replaces the dynamic hash-map with a static memory buffer, which only stores window aggregates. This prevents hash-collisions and eliminates overhead for resizing the state. To prevent out-of-bound accesses, Grizzly de-optimizes the code variant if a key lies outside of the assumed value range. This additional check introduces a negligible overhead as the condition is

false as long the assumption is valid. Thus, the CPU branch predictor can predict the branch always correctly.

6.2.3 Exploiting Value Distributions. The efficiency of window aggregations highly depends on the hash-map implementation and the key distribution in the workload [22]. A global shared hash-map is beneficial for uniformly distributed keys, as concurrent accesses to the same key are less frequent. In contrast, skewed workloads with heavy hitters benefit from an independent hash-map per thread. This eliminates concurrent accesses and synchronization overhead but requires merging and reduces memory efficiency as aggregates are stored multiple times. Grizzly adaptively chooses between both strategies depending on the data characteristics. During instrumentalization, Grizzly creates a histogram over the keyspace to monitors the distribution. If Grizzly can assume that the majority of accesses could hit at least the L3 cache, Grizzly uses the independent hash-map. During *optimized execution*, Grizzly monitors the performance counters of the cache coherence protocol, to detect if the selected strategy is still appropriate. For instance, an increasing number of exclusive accesses to a cache line that another thread has in exclusive access indicates that the uniform distribution shifts to a more skewed distribution.

7 EVALUATION

In this section, we experimentally evaluate Grizzly. In Section 7.1, we introduce our experimental setup. After that, we conduct four sets of experiments. First, we evaluate the throughput and latency of Grizzly and state-of-the-art SPEs for different workloads (Sec. 7.2). Second, we highlight the throughput impact of different workload characteristics (Sec. 7.3). Third, we showcase the advantages of Grizzly's adaptive optimizations (Sec. 7.4). Finally, we analyze resource utilization and system efficiency to reveal the reasons why Grizzly's utilize modern hardware more efficiently compared to state-of-the-art SPEs (Sec. 7.5).

7.1 Experimental Setup

In the following section, we present the hardware and software configurations (Sec. 7.1.1) and the workloads of our experiments (Sec. 7.1.2).

7.1.1 Hardware and Software. We execute experiments on two machines: a commodity, single-socket server (Server A) and a high-end, multi-socket server (Server B) (to isolate the effects of NUMA). Server A has one Intel Core i7-6700K processor with four physical cores (in total 8 logical cores) and contains 32GB main memory. Server B has two Intel Xeon 6126 with 12 physical cores each (in total 48 logical cores) and contains 1.48TB main memory. Both CPUs have a dedicated 32 KB L1 cache for data and instructions per core.
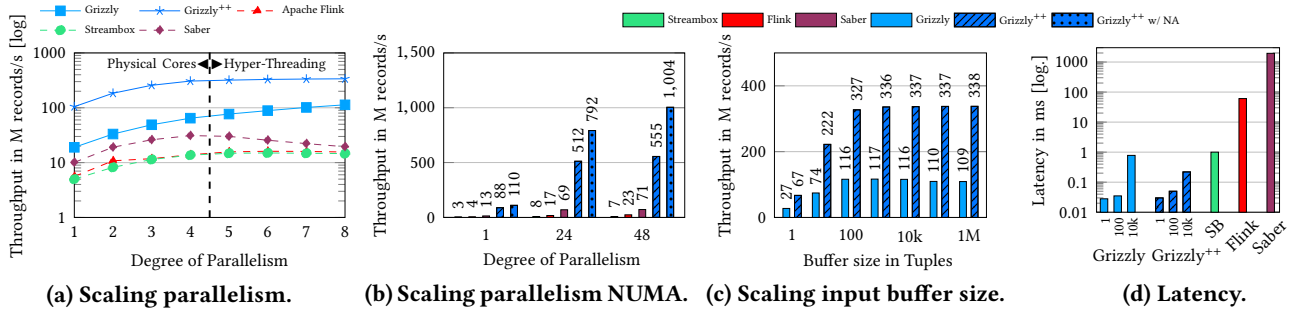
(a) Scaling parallelism.   **(b) Scaling parallelism NUMA.**   **(c) Scaling input buffer size.**   **(d) Latency.**

Figure 6: System Comparison.

Additionally, Server A has 256 KB L2 cache per core and 8 MB L3 cache per CPU, and Server B has 1MB L2 cache per core and 19.25 MB L3 cache per CPU. If not stated otherwise, we execute all experiments on Server A using all logical cores.

The C++ implementations are compiled with GCC 6.5 and O3 optimization, as well as the mtune flags to produce specific code for the underlying CPU. We measure hardware performance counters using PAPI [65] version 5.5.1. The Java implementations run on the HotSpot VM in version 1.8.0 201. We use Apache Flink [15] in version 1.8.0 as a representative scale-out SPE and disable fault-tolerance mechanisms to minimize overhead. As representative scale-up SPEs, we use Streambox [53] (C++ based) and Saber [45] (JVM-based). In the following evaluation, we examine two versions of Grizzly. Grizzly refers to a version that does not exploit any knowledge about the data characteristics and thus applies no adaptive, data-driven optimizations. Grizzly^{++} refers to a version that is aware of data characteristics and thus applies the adaptive, data-driven optimizations from Section 6.2.

7.1.2 Workload. If not stated other, we base our experiments on variations of the Yahoo! Streaming Benchmark (YSB) to simulate real-world stream processing workloads. We follow the YSB implementation of Grier et al. [36] and Saber [58], which processes all data directly inside the SPE to prevent the overhead of external systems such as Apache Kafka or Redis. The YSB query consists of two processing steps. First, the YSB query evaluates if the event type matches the string *"view"* (33% of the records qualify). Second, the YSB query aggregates the qualifying records by their campaign id into a processing-time tumbling window of 10 seconds. We ingest data with 10k distinct keys and process a SUM aggregation.

7.2 System Comparison

In this section, we study the system throughput under the impact of parallelism (Sec. 7.2.1 and 7.2.2), compare processing latencies (Sec. 7.2.3), evaluate queries from the Nexmark benchmark (Sec. 7.2.4), and discuss all findings (Sec 7.2.5).

7.2.1 Scaling on a single socket. In this experiment, we evaluate the scalability of Flink, Streambox, Saber, and Grizzly

on Server A. We execute the default YSB query and study the throughput for an increasing degree of parallelism.

Results. In Figure 6(a), we scale the execution of the YSB benchmark using different degrees of parallelism. Flink and Streambox scale up similar and achieve a throughput of up to 16M records/s. In contrast, Saber outperforms Flink and Streambox by 2.2x (31M records/s). Saber's throughput increases up to four cores. Beyond that, the throughput decreases due to hyper-threading. Hyper-threading (HT) introduces two logical cores for each physical core, which share caches, branch prediction units, and functional units [32]. HT is beneficial if multiple threads execute different types of work (e.g., computation and I/O accesses) [80]. Therefore, the results for Saber indicate that multiple threads compete for the same shared CPU resources, which limits the performance improvements of HT [32]. Note that the results are in line with numbers published by the original authors [57]. As shown, both versions of Grizzly outperform all other SPEs. In particular, Grizzly achieves near-linear speedup and exploits HT efficiently. In contrast, by exploiting adaptive optimizations, Grizzly^{++} achieves the highest throughput, which is over an order of magnitude higher compared to Flink, Saber, and Streambox. Furthermore, Grizzly^{++} becomes memory bound for a degree of parallelism of four (all physical cores), and thus HT does not improve throughput significantly. Overall, without adaptive optimizations, Grizzly outperforms Saber by an average factor of 2.9 (min 1.7x, max 5.8x) and Flink/Streambox by a factor of 5.3 (min 3.7x, max 7.7x). With adaptive optimizations, Grizzly^{++} achieves an average speedup of 4.2x (min 2.9x, max 5.4x) over the generic Grizzly version (due to its more dense memory layout). As a result, Grizzly^{++} outperforms all evaluated SPEs on average by at least one order of magnitude (Saber 11.5x, Streambox 21.4x, Flink 21.5x).

7.2.2 NUMA Scaling. In this experiment, we evaluate the scalability of all SPEs on Server B. For Grizzly, we differentiate between a NUMA-aware version as outlined in Section 5.2 (Grizzly^{++} w/ NA) and a NUMA-unaware version

(Grizzly^{++} w/o NA). We execute the YSB query and compare the throughput for parallelism of 1, 24, and 48.

Results. Figure 6(b) highlights the impact of NUMA for the individual systems. Overall, this experiment highlights the impact of NUMA-awareness. Already, for parallelism of one, Grizzly^{++} w/ NA leads to a speedup of 1.3x as it guarantees that all data is located on the same numa node as the processing thread. By increasing the degree of parallelism to 24, all systems improve throughput. In this case, Grizzly^{++} w/ NA results in a speedup of 1.5x compared to Grizzly^{++} w/o NA. If we further increase the degree of parallelism to 48, we observe that the throughput of all numa-unaware systems stagnates. In contrast, Grizzly^{++} w/ NA optimizations result in an additional speedup of 1.8x.

7.2.3 Latency. In this experiment, we examine the processing latency. First, we study the dependency between the buffer size and the latency for Grizzly. Additionally, we compare the latency of Grizzly, Saber, Streambox, and Flink. We define latency as the duration between the ingestion time of the last record that contributes to a window aggregate and the output of the aggregate of that window [41]. We execute the YSB on all systems with a parallelism of eight.

Results. In Figure 6(c), we observe that both Grizzly versions reach peak performance for a buffer size larger than 100 records as the run time overhead becomes neglectable [74]. For the dependency between buffer size and latency, Figure 6(d) highlights two aspects. First, the buffer size has a high impact on the processing latency of Grizzly. For a buffer size of one, Grizzly achieves an average latency of 0.035ms (± 0.014ms) that increases up to 0.91ms (± 0.26ms) for a buffer size of 10k records. This characteristic is independent of the Grizzly version. Second, the code optimizations of Grizzly^{++} lead to lower latencies and smaller variances for large buffer sizes (avg. latency 0.22ms ± 0.15ms). The main reason for this is the higher complexity of the TBB hash-map in the default Grizzly version. Streambox is the only SPEs that is also able to reach average latencies in the range of 1ms (± 0.4ms). In contrast, Flink has on average a latency of 60ms (± 4ms) and Saber 1.9s (± 49ms). The higher latency of Saber is caused by its micro-batch processing model [45]. The micro-batching approach trades higher throughput for higher latency and is one of the reasons why Saber's throughput is higher compared to Flink and Streambox. Overall, both versions of Grizzly achieves up to an order of magnitude lower latencies and smaller latency variance than all other SPEs.

7.2.4 Nexmark Benchmark. In this set of experiments, we evaluate five queries of the Nexmark benchmark on Grizzly^{++} and Flink. In particular, we use a tumbling window of 10s for Q7 and Q8, a sliding window of 10s with a slice of 1s for Q5, and a Sum aggregation for Q5 and Q7. In contrast, Q1 and Q2 are window-less.
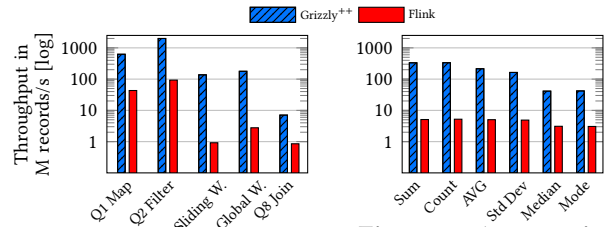

Figure 7: Nexmark.

Figure 8: Aggregation functions.

Results. In Figure 7, we present the throughput of queries with different workloads on both systems. For the stateless Map (Q1) and Filter (Q2) queries, Grizzly^{++} outperforms Flink by at least 14x. Flink and Grizzly perform these queries without any coordination between threads and process every input record only once. However, Grizzly^{++} benefits from eliminating any data serialization overhead. For the stateful queries Q5 and Q7, Grizzly^{++} outperforms Flink by at least a factor of 60x due to its more compact state representation, which improves cache locality. Additionally, Grizzly's task-based parallelization technique is beneficial for Q7. In contrast, Flink cannot parallelize the processing of global windows. The stream join of Q8 is highly resource-intensive, as both systems have to materialize the complete input data stream until the window triggers. Grizzly^{++} concurrently builds and probes the join tables across all processing threads, which introduce additional coordination overhead. However, Grizzly^{++} still outperforms Flink by at least a factor 8x on Q8. Overall, we observe that Grizzly++ provides similar throughput improvements among all Nexmark queries and outperforms Flink by at least 8x. As our selected set of queries covers basic building blocks of queries, we expect similar performance improvements for other streaming workloads.

7.2.5 Discussion. Across all experiments, we observed, that Grizzly outperforms all evaluated systems by up to one order of magnitude in throughput as well as latency on commodity hardware as well as high-end NUMA servers. The code specialization based on data characteristics (Grizzly^{++}) increases the throughput by up to 5.4x compared to the version without code specialization (Grizzly). Starting from small buffer sizes of 100 elements, Grizzly^{++} reaches peak throughput (337 million records/s) and achieves sub-millisecond latencies. Therefore, Grizzly mitigates the trade-off between latency and throughput. This experiment highlights two important aspects of stream processing on modern hardware. First, both versions of Grizzly exploit the cores of the CPU efficiently and code generation leads up to an order of magnitude performance improvement. Second, the code specializations of Grizzly^{++} induce an additional speed up and are crucial to fully utilize modern hardware efficiently. Furthermore, we highlight that Grizzly supports a wide range of workloads
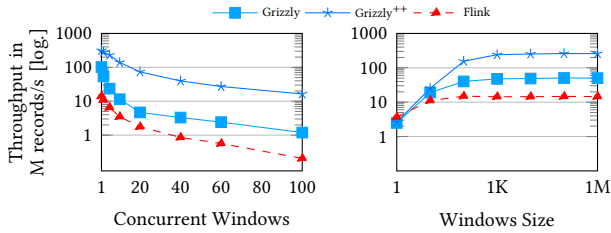
Figure 9: Concurrent windows.

Figure 10: Count-based windows.

Figure 11: Throughput for scaling the state size.

and reaches high performance on complex operators such as joins or aggregations.

7.3 Workload Characteristics

In the following set of experiments, we study the impact of particular workload characteristics on the throughput. To this end, we study the impact of state size (Sec. 7.3.4), the number of concurrent windows (Sec. 7.3.2), and count-based windows (Sec. 7.3.3).

7.3.1 Impact of Aggregation Type. In this experiment, we evaluate six window aggregation functions with a tumbling window of 10s on Grizzly^{++} and Flink. We evaluate four decomposable (i.e., *Sum*, *Count*, *AVG*, *StdDev*) and two non-decomposable aggregation functions (i.e., *Median*, *Mode*).

Results. The results in Figure 8 highlight the dependency between the complexity of the aggregation function and processing throughput. For decomposable aggregation functions, Flink reaches an average throughput of 5M records per second. In contrast, Grizzly^{++} outperforms Flink by a factor of up to 64x. Depending on the number of atomic state variables, Grizzly^{++}'s throughput varies up to a factor of 2x (e.g., SUM requires one atomic update, and Std Dev requires three updates per record). In the case of non-decomposable aggregation functions, the throughput of both systems decreases as they must materialize all records until the window ends. However, Grizzly^{++} is still able to outperform Flink by a factor of 13x. This is mainly due to its light-weight, in-memory state representation.

7.3.2 Impact of Concurrent Windows. In this experiment, we study the throughput of over overlapping sliding windows. In particular, the efficient support of sliding windows is crucial as the ratio between size and slide could lead to high numbers of concurrent windows, e.g., a sliding window of one hour with a slice of one-minute results in 60 concurrent windows. We use the YSB query with a sliding window and scale the number of concurrent windows from 1 to 100.

Results. Figure 9 shows that the throughput of Flink and Grizzly is highly dependent on the number of concurrent windows. The overhead of concurrent sliding windows was demonstrated in previous work [64, 67, 68]. Both Flink and Grizzly use buckets to maintain window aggregates. Thus,
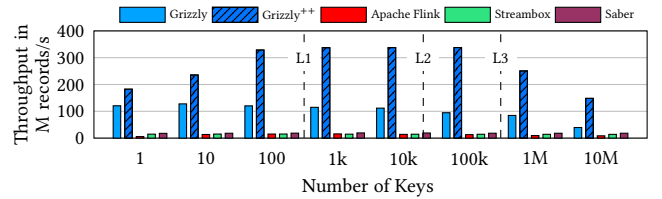
both systems have to assign each record to multiple windows. As a result, the performance decreases with an increasing number of concurrent windows. However, Grizzly outperforms Flink on average by a factor of 4.2x (Grizzly) and 44x (Grizzly^{++}). Grizzly^{++} achieves a higher throughput as it represents state in a dense fixed-size array. This simplifies data access and reduces cache misses also in the case of concurrent windows.

7.3.3 Impact of Window Measure. In the previous experiments, we studied time-based windows. In contrast to time-based windows, the triggering logic of count-based windows is fundamentally different and more complex as it requires updating a global counter after each record assignment. In the following experiment, we study the impact of the size of a count window on the throughput. We execute the YSB query with a count-based window and vary the window size, which directly determines the window trigger frequency.

Results. Figure 10 reveals that the trigger overhead dominates the throughput for small window sizes (1-100 records) across all SPEs. Starting from a window size of 1k records, the overhead gets negligible, and the throughput becomes independent of the window size. For windows larger than 1k records, Grizzly outperforms Flink by a factor of 3.4x (generic Grizzly) and 17.7x (Grizzly^{++}). In comparison to time-based windows, count-based windows reduce the throughput by a factor of two. This is mainly due to the more complex window trigger logic required for count-based windows (see Section 4.2.3). In particular, Grizzly maintains a counter per key and window, which has to be incremented atomically for each assigned value.

7.3.4 Impact of State Size. In this experiment, we scale the state size by adjusting the number of distinct keys (8byte) in the input data stream of the YSB query (campaign ids). In particular, we execute the default YSB query and scale the number of distinct campaign ids (keys) from 1 to 10 million. Because the YSB query aggregates by key, the number of keys directly impacts the intermediate state size.

Results. Figure 11 highlights the dependency between throughput and state size (number of keys) across the examined systems. Streambox and Saber achieve, on average, a throughput of 15M and 19M record/s, respectively. Both systems outperform Flink that reaches the lowest average throughput with 11M records/s. If the stream only consists
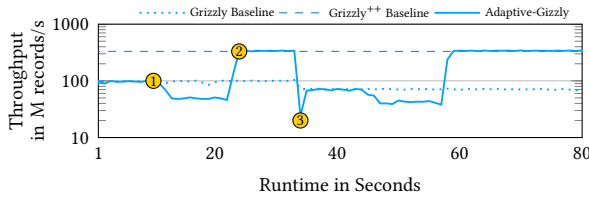
Figure 12: Changing value range.

of one key, Flink's throughput decreases to 6M record/s, which is equal to its single-thread performance (see evaluation in Section 7.2.1). This demonstrates the disadvantage of key-partitioning based parallelization as only one thread performs computations per distinct key. In contrast, Grizzly outperforms all other SPEs for all state sizes.

In general, increasing key ranges of generated records induces only a small impact on throughput for Flink, Saber, and Streambox. If the intermediate state exceeds the L3 Cache (more then 130k keys), the throughput slightly decreases (e.g., for Flink 0.6x). This result indicates that all three SPEs do not exploit modern hardware, in particular, CPU caches, efficiently. In comparison to the best performing SPE (Saber), Grizzly reaches an average speedup of 5.9x (min 4.4x, max 7.0x) and Grizzly^{++} reaches an average speedup of 15.3x (min 10.2x, max 18.4x). For small state sizes (1-100 keys), Grizzly^{++} induces a high overhead. This overhead is mainly caused by concurrent accesses on a small number of keys that result in a significant synchronization overhead. Between 100 and 100k keys, Grizzly^{++} reaches peak performance (338M records/s). For more than 100k keys, the performance of both Grizzly versions decreases as the state size exceeds the L3 Cache.

7.3.5 Discussion. In these experiments, Grizzly outperformed Streambox, Saber, and Flink across all tested workload configurations. Depending on the query workload, the performance improvements differ. In particular, the aggregation type, the window type, and the number of concurrent windows impact performance significantly. However, we proved that the adaptive optimizations of Grizzly^{++} exploit the cache hierarchy and the capabilities of modern hardware most efficiently.

7.4 Adaptive Optimizations

In the following set of experiments, we evaluate Grizzly's adaptive optimization techniques (Sec. 6.2).

7.4.1 Compilation Stages. In this experiment, we study the performance impact of Grizzly's three compilation stages. We execute the YSB query and configure the duration of each compilation stage to 10 seconds. After 30 seconds, the number of distinct keys increases by 10x.

Results. Figure 12 illustrates the system throughput of Grizzly over time. At the beginning, Grizzly deploys the generic code variant and reaches a throughput of 100M records/s. At ①, Grizzly migrates to the instrumented code
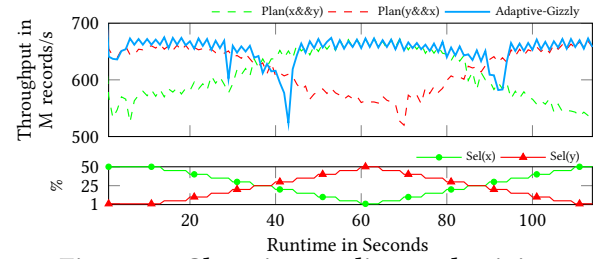

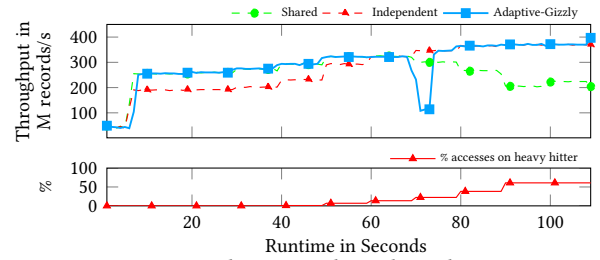**Figure 13: Changing predicate selectivity.**

Figure 14: Changing key distribution.

variant. The profiling instructions introduce an overhead of 50% such that the throughput decreases to 50M records/s. At ②, Grizzly utilizes the collected profiling information to deploy an optimized code variant. This results in a speedup of 3.3x over the generic baseline (330M records/s). At ③, the number of distinct keys increases and Grizzly de-optimizes the pipeline variant as discussed in Section 6.1. After deoptimization, the throughput drops shortly to 24M records/s, before a new optimization circle starts.

7.4.2 Selectivity Profiling. In this experiment, we study the performance impact of optimizing predicate reordering in queries containing selections (see Sec. 6.2.1). To this end, we introduce five greater equal predicates into the YSB query such that 120 different predicate orders are possible. During execution, we vary the selectivity of two predicates (x and y). All other predicates have a fixed selectivity of 50%.

Results. Figure 13 compares the throughput of Grizzly with two specific plans, which either first evaluate the x or y predicate. At the beginning, predicate y is very selective. Thus it is more efficient to evaluate y first. Starting from second 40, the predicate x becomes more selective then y. Grizzly detects the crossing point, and changes the operator order to evaluate x first. As a result, this adaptive optimization leads to a throughput difference of up to 150M records/s.

7.4.3 Heavy-Hitter-Profiling. In this experiment, we study the impact of Grizzly's adaptive optimization for detecting the distribution of keys in window aggregations (see Sec. 6.2.3). We execute the standard YSB query with 1M distinct keys. Over time, we shift the distribution of keys, starting with a nearly uniform distribution towards a scenario where 60% of records access the same key.

Table 1: Resource utilization per Record on YSB query.

	Grizzly	Grizzly^{++}	Streambox	Saber	Flink
Branches/rec	18.2	7	706	184	701
Branch Mispred./rec	0.78	0.38	5.5	0.25	2.23
L1-D Misses/rec	3.5	1.39	50.6	11.9	23.8
L2-D Misses/rec	11.2	3.59	132	22.6	43.8
LLC Misses/rec	1.8	1.92	46.6	10.9	18.7
TLB-D Misses/rec	0.01934	0.01234	6.7	0.20	0.45
Instructions/rec	139.4	41.6	3440	1157	4162
L1-I Misses/rec	0.00026	0.00011	13.8	1.1	14.4
L2-I Misses/rec	0.00023	0.00010	2.7	0.26	1.2
TLB-I Misses/rec	0.00012	0.00006	0.091	0.017	0.081

Results. After the initial profiling phase, Grizzly detects that the keyspace is nearly uniformly distributed and chooses a shared hash-map. After 60 seconds runtime, the performance of the shared hash-map significantly decreases, as more than 10% of all records access the same key. Grizzly detects the increasing cache contention with performance counters and triggers re-optimizes by migrating to an independent hash-map. For highly skewed distributions, the independent hash-map achieves a speedup of up to 2x.

7.4.4 Discussion. The experiments showed that Grizzly is able to detect and exploit changing data-characteristics adaptively at runtime. Depending on the scenario, an optimized code variant can result in a performance gain of up to 3x. Thus, it is important to limit the execution time of unoptimized pipeline variants (e.g., by profiling only small buffers. Leis et al. already showed that 10k records are enough to identify join orders [49]).

7.5 Analysis of Resource Utilization

In this section, we evaluate the resource utilization of Grizzly, Streambox, Flink, and Saber. The resource utilization enables us to explain the different performance characteristics observed in previous experiments. In Table 1, we show performance counters for the default YSB query. These results reflect the pure execution workload per record without any preprocessing. We divide the collected counter into three blocks: Control Flow, Data Locality, and Code Locality.

Control Flow In the first block, Table 1 shows the number of executed branches and branch mispredictions per record. These counters are essential to analyze the control flow of the SPEs. Across all SPEs, Streambox, and Flink introduce the highest number of branches and branch mispredictions. For Flink, data serialization and object allocation cause many dynamic branches and branch mispredictions, which was already shown by Zeuch et al. [76]. Saber achieves the fewest branch mispredictions but executes up-to 26x more branches compared to Grizzly^{++}. The main reason for the high number of branches is Saber's micro-batch processing model, which performs many prediction-friendly branches by looping over data in batches. Overall, both versions of Grizzly introduce very few branches and branch mispredictions. Finally, we show that the adaptive optimizations in Grizzly^{++} reduce branches and branch mispredictions by a factor of two.

Data Locality In the second block, Table 1 presents performance counters to analyze data locality of the SPEs. Stream processing workloads usually access each input record only once, which causes a relatively high number of data-related cache misses. As shown in Table 1, Streambox and Flink induce the highest number of data cache misses across all cache levels. Additionally, Streambox causes 29x more TLB-D misses than any other SPE. These results indicate that Streambox and Flink cause more memory accesses for the same input data and that the utilized data layout and access patterns are sub-optimal. In contrast, Saber directly processes raw data, which causes fewer cache misses across all cache levels. However, Saber still causes at least 3.4x more L1 cache misses, 2x more L2 misses, and 6x more LLC cache misses compared to Grizzly. Both versions of Grizzly cause significantly fewer cache misses compared to all other SPEs and achieve a higher data locality. As a result, the access latencies for records decrease, which leads to a significant speedup. Furthermore, Grizzly causes at least 10x fewer TLB-D misses compared to Saber. The high data locality of Grizzly highlights the benefit of direct data accesses on raw data without serialization, data copying, or object allocation overhead. The most efficient data locality is achieved by Grizzly^{++} that stores window state in a dense array, which results in 2.5x fewer L1 and 3.1x fewer L2 cache misses.

Code Locality In the last block, Table 1 shows performance counters related to code efficiency and locality of the SPEs. Overall, Streambox, Saber, and Flink execute at least 8x (Grizzly) and 27x (Grizzly^{++}) more instructions per input record. This highlights that Grizzly's code generation results in a very compact and CPU-friendly code. Furthermore, adaptive optimizations of Grizzly^{++} reduce the number of executed instructions by up to 3x. The results for instruction cache misses reveal, that Grizzly overall archives a much higher code locality. Flink and Streambox cause the most instruction cache misses, and many TLB-I misses. In contrast, Saber causes 10x fewer instruction cache misses as a result of its micro-batch processing model. However, both versions of Grizzly cause basically no instruction cache misses and TLB-I misses per record. This indicates that the generated code fits entirely into the L1 instruction cache, and the generated instruction sequence is CPU-friendly.

Discussion Our analysis of resource utilization reveals that both Grizzly versions result in better control flow as well as higher data and instruction locality. Furthermore, exploiting data characteristics in Grizzly^{++} is vital to achieve peak performance. In contrast, for Flink, Streambox, and Saber, we observe inefficient memory utilization, which is caused by data serialization, object allocation, and the execution of inefficient and complex code. In sum, Grizzly's code generation for stream processing is essential to utilize resources of modern CPUs efficiently.

8 RELATED WORK

We structure the related work into three areas: SPEs, query compilation, and adaptive optimizations.

Stream Processing Engines. The first generation of SPEs laid the foundation to handle continuous queries over unbounded data streams [1, 2, 19, 20]. Due to growing data sizes and higher velocities, the second-generation of SPSs follow scale-out architectures while focusing on higher throughput, lower latency, and fault tolerance with exactly-once semantics [5, 10, 15, 17, 54, 66, 71, 72]. System S introduced optimizations for stream processing [34, 37]. In contrast to System S, Grizzly is a scale-up SPE that efficiently utilizes modern hardware. To this end, it fuses operators deeply together and eliminates any function calls between them.

Further examples for scale-up SPSs are SABER [45], Streambox [53], BriskStream [79], and Trill [18]. SABER focuses on hybrid stream processing on CPUs and GPUs. Streambox groups records in epochs and processes them for each operator in parallel. In contrast, Grizzly compiles queries into efficient code, which is executed using a task-based approach on a shared global state. Trill applies code generation techniques to rewrite user-defined functions to a block-oriented processing model over a columnar data layout. In contrast to Trill, Grizzly focuses on the fusion of multiple operators into one code block. BriskStream optimizes execution for NUMA hardware by distributing operations across NUMA-regions. In contrast, Grizzly follows a data-centric approach and executes operators on the NUMA nodes where the data is located. Furthermore, Grizzly fuses all operators into code without introducing unnecessary boundaries. Previous work showed that current SPEs, do not fully utilize the resources of modern hardware [76, 78]. Our work recognizes these limitations and proposes Grizzly, which generates highly efficient code. As a result, Grizzly outperforms state-of-the-art SPEs by at least an order of magnitude and reaches the performance of hand-optimized code.

Query Compilation. Query compilation for batch processing was extensively studied by Rao et al. [61], Krikellas et al. [46], and Neumann [55]. It was applied in many data processing systems [25, 33, 44, 46, 55, 56, 63, 69]. Further work studied the support of user-defined functions [24], query compilation for heterogeneous hardware [12, 59], efficient incremental view maintenance [4], the architecture of query compilers [3, 29, 43], and the combination of compilation and vectorization [52]. In this work, we complement the state-of-the-art by introducing query compilation for stream processing. Our technique enables the fusion of queries involving complex operations such as the window assignment, triggering, and aggregation. Furthermore, we enable adaptive optimizations. Orthogonal to our work, Kroll et al. [47]

proposed ARC an intermediate representation to unify batch and stream queries, which could act as an input for Grizzly.

Adaptive Optimizations. In database research, adaptive optimizations have been extensively studied [8, 9, 63]. Răducanu et al. [60] proposed micro adaptivity by comparing the run-time of different operator implementations. Zeuch et al. [77], extended this approach by exploiting hardware counters to detect data properties like sortedness or operator order. In contrast, Dutt et al. [27] introduce explicit counters between operators to gather workload properties. In Grizzly, we combine these approaches to enable adaptive optimizations for stream processing. To this end, Grizzly monitors performance counters to detect changing data characteristics and generates instrumented code to collect detailed data statistics for optimization by using JIT-compilation. Additional work studied adaptive optimizations for stream processing [20, 34, 81]. These works mainly focused on the migration between query plans in a distributed setting. In contrast, Grizzly focuses on adaptive optimizations based on modern profiling techniques and query compilation to fully exploit modern hardware. Query compilers for data-at-rest apply adaptive compilation techniques to reduce compilation time [28, 44]. This is orthogonal to our work, as we apply adaptive code optimizations to react to changing data characteristics in stream processing queries.

9 CONCLUSION

In this paper, we transferred the concept of query compilation for data-at-rest queries to the operators and semantics of stream processing. We present Grizzly, the first adaptive, compilation-based SPE that is able to generate highly efficient code for streaming queries. Grizzly supports streaming queries with different window types, window measures, and window functions. It utilized adaptive optimizations to react to changing data-characteristics at runtime. To this end, we combine profiling techniques and apply task-based parallelization to fully utilize modern multi-core CPUs while fulfilling the ordering requirements of stream processing. Our extensive experiments demonstrate that Grizzly outperforms the state-of-the-art SPEs by up to an order of magnitude due to better utilization of modern hardware. With Grizzly, we lay the foundation for the efficient use of modern hardware in stream processing. Furthermore, we will incorporate Grizzly in our new data processing platform NebulaStream [73].

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the borealis stream processing engine.. In *CIDR*, Vol. 5. 277–289.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB Journal* 12, 2 (2003), 120–139.

[3] Sameer Agarwal, Davies Liu, and Reynold Xin. 2016. Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop. https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html. [Online; accessed 31.5.2019].

[4] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL Compiler for High-performance Delta Processing in Main-memory Databases. In *PVLDB*. VLDB Endowment, 1566–1569.

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. In *PVLDB*, Vol. 6. VLDB Endowment, 1033–1044.

[6] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *SIGPLAN*. ACM, 177–189.

[7] David I August, Wen-mei W Hwu, and Scott A Mahlke. 1997. A framework for balancing control flow and predication. In *MICRO*. IEEE, 92–103.

[8] Shivnath Babu and Pedro Bizarro. 2005. Adaptive query processing in the looking glass. In *CIDR*.

[9] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *SIGMOD*. ACM, 107–118.

[10] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. 2010. IBM infosphere streams for scalable, real-time, intelligent transportation services. In *SIGMOD*. ACM, 1093–1104.

[11] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In *PVLDB*, Vol. 3. VLDB Endowment, 232–243.

[12] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *The VLDB Journal* 27, 6 (2018), 797–822.

[13] David Broneske, Sebastian Breß, and Gunter Saake. 2013. Database scan variants on modern CPUs: A performance study. In *IMDM*. Springer, 97–111.

[14] David Broneske, Andreas Meister, and Gunter Saake. 2017. Hardware-sensitive scan operator variants for compiled selection pipelines. In *BTW*. Gesellschaft für Informatik, Bonn.

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 36, 4 (2015).

[16] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate sharing for user-defined windows. In *CIKM*. 1201–1210.

[17] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*. ACM, 725–736.

[18] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert De-Line, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. In *PVLDB*, Vol. 8. VLDB Endowment, 401–412.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, et al. 2003. Telegraphcq: Continuous dataflow processing for an Uncertain world.. In *CIDR*, Vol. 2. 4.

[20] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, Vol. 29. ACM, 379–390.

[21] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *IPDPS*. IEEE, 1789–1792.

[22] John Cieslewicz and Kenneth A Ross. 2007. Adaptive aggregation on chip multiprocessors. In *VLDB*. VLDB Endowment, 339–350.

[23] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

[24] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*.

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*. ACM, 1243–1254.

[26] M. Dreseler, J. Kossmann, J. Frohnhofen, M. Uflacker, and H. Plattner. 2018. Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans. In *ICDEW*. 102–109.

[27] Anshuman Dutt and Jayant R. Haritsa. 2014. Plan Bouquets: Query Processing Without Selectivity Estimation. In *SIGMOD*. ACM, New York, NY, USA, 1039–1050.

[28] Grégory Essertel, Ruby Tahboub, and Tiark Rompf. 2018. On-Stack Replacement for Program Generators and Source-to-Source Compilers. *Preprint* (2018).

[29] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *OSDI*. 799–815.

[30] Martin Faust, David Schwalb, and Jens Krueger. 2013. Fast column scans: Paged indices for in-memory column stores. In *IMDM*. Springer, 15–27.

[31] Stephen J Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO*. IEEE, 241–252.

[32] Agner Fog. 2009. How good is hyperthreading? https://www.agner.org/optimize/blog/read.php?i=6. [Online; accessed 31.5.2019].

[33] Craig Freedman, Erik Ismert, and Per-Åke Larson. 2014. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin* 37 (2014), 22–30.

[34] Bugra Gedik, Henrique Andrade, and Kun-Lung Wu. 2009. A code generation approach to optimizing high-performance distributed data stream processing. In *CIKM*. ACM, 847–856.

[35] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*. ACM, 102–111.

[36] Jamie Grier. 2016. Extending the Yahoo! Streaming Benchmark. https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark. [Online; accessed 31.5.2019].

[37] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *Comput. Surveys* 46, 4, Article 46 (March 2014), 34 pages.

[38] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *ACM Sigplan Notices*, Vol. 27. ACM, 32–43.

[39] Intel. 2019. Intel(R) Threading Building Blocks: Concurrent Hash Map. https://software.intel.com/en-us/node/506191. [Online; accessed 31.5.2019].

[40] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2014. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys & Tutorials* 17, 1 (2014), 381–404.

[41] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *ICDE*. IEEE, 1507–1518.

[42] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. 2013. Experimental evaluation of NUMA effects on database management systems. In *BTW*. Gesellschaft für Informatik eV.

[43] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. In *PVLDB*, Vol. 7. VLDB Endowment, 853–864.

[44] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *ICDE*. IEEE, 197–208.

[45] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*. ACM, 555–569.

[46] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. 613–624.

[47] Lars Kroll, Klas Segeljakt, Paris Carbone, Christian Schulte, and Seif Haridi. 2019. Arc: An IR for Batch and Stream Programming. In *DBPL*. ACM, New York, NY, USA, 53–58.

[48] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. ACM, 743–754.

[49] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *The VLDB Journal* 27, 5 (Oct. 2018), 643–668.

[50] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M Lohman. 2013. NUMA-aware algorithms: the case of data shuffling.. In *CIDR*.

[51] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General (?)! *TOPC* 5, 4 (2019), 16.

[52] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. In *PVLDB*, Vol. 11. VLDB Endowment, 1–13.

[53] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *USENIX*. 617–629.

[54] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.

[55] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *PVLDB*, Vol. 4. VLDB Endowment, 539–550.

[56] Paroski Paroski. 2016. Code generation: The inner sanctum of database performance. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-ofdatabase-performance.html. [Online; accessed 31.5.2019].

[57] Peter Pietzuch, Panagiotis Garefalakis, Alexandros Koliousis, Holger Pirk, and George Theodorakis. 2018. Do We Need Distributed Stream Processing? https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing. [Online; accessed 31.5.2019].

[58] Peter Pietzuch, Panagiotis Garefalakis, Alexandros Koliousis, Holger Pirk, and George Theodorakis. 2018. StreamBench. https://github.com/lsds/StreamBench. [Online; accessed 31.5.2019].

[59] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. In *PVLDB*, Vol. 9. VLDB Endowment, 1707–1718.

[60] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro adaptivity in vectorwise. In *SIGMOD*. ACM, 1231–1242.

[61] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. 2006. Compiled query execution engine using JVM. In *ICDE*. IEEE.

[62] Kenneth A Ross. 2004. Selection conditions in main memory. *TODS* 29, 1 (2004), 132–161.

[63] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *PVLDB*, Vol. 1. 19–28.

[64] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. In *PVLDB*, Vol. 8. VLDB Endowment, 702–713.

[65] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[66] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *SIGMOD*. ACM, 147–156.

[67] Jonas Traub, Philipp M. Grulich, Alejandro Rodrıguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2018. Scotty: Efficient Window Aggregation for out-of-order Stream Processing. In *ICDE*. IEEE.

[68] Jonas Traub, Philipp M. Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient window aggregation with general stream slicing. In *EDBT*.

[69] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin* (2014).

[70] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. In *PVLDB*, Vol. 2. VLDB Endowment, 385–394.

[71] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *ICDE*. IEEE, 723–734.

[72] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*. ACM, 423–438.

[73] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *CIDR*.

[74] Steffen Zeuch and Johann-Christoph Freytag. 2014. QTM: modelling query execution with tasks. In *PVLDB*, Vol. 7. VLDB Endowment.

[75] Steffen Zeuch and Johann-Christoph Freytag. 2015. Selection on modern CPUs. In *ADMS*. ACM, 5.

[76] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. In *PVLDB*, Vol. 12. VLDB Endowment, 516–530.

[77] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. 2016. Non-Invasive Progressive Optimization for In-Memory Databases. In *PVLDB*, Vol. 9. 1659–1670.

[78] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream

Processing Systems on Multi-Core Processors. In *ICDE*. 659–670.

[79] Shuhao Zhang, Jiong He, Chi Amelie Zhou, and Bingsheng He. 2019. BriskStream: Scaling Stream Processing on Multicore Architectures. In *SIGMOD*.

[80] Jingren Zhou, John Cieslewicz, Kenneth A Ross, and Mihir Shah. 2005. Improving database performance on simultaneous multithreading processors. In *PVLDB*. VLDB Endowment, 49–60.

[81] Yali Zhu, Elke A Rundensteiner, and George T Heineman. 2004. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*. ACM, 431–442.