# Darwin: Scale-In Stream Processing

Lawrence Benson
Hasso Plattner Institute
University of Potsdam, Germany
lawrence.benson@hpi.de

Tilmann Rabl
Hasso Plattner Institute
University of Potsdam, Germany
tilmann.rabl@hpi.de

## ABSTRACT

Companies increasingly rely on stream processing engines (SPEs) to quickly analyze data and monitor infrastructure. These systems enable continuous querying of data at high rates. Current production-level systems, such as Apache Flink and Spark, rely on clusters of servers to scale out processing capacity. Yet, these scale-out systems are resource inefficient and cannot fully utilize the hardware. As a solution, hardware-optimized, single-server, scale-up SPEs were developed. To get the best performance, they neglect essential features for industry adoption, such as larger-than-memory state and recovery. This requires users to choose between high performance or system availability. While some streaming workloads can afford to lose or reprocess large amounts of data, others cannot, forcing them to accept lower performance. Users also face a large performance drop once their workloads slightly exceed a single server and force them to use scale-out SPEs.

To acknowledge that real-world stream processing setups have drastically varying performance and availability requirements, we propose scale-in processing. Scale-in processing is a new paradigm that adapts to various application demands by achieving high hardware utilization on a wide range of single- and multi-node hardware setups, reducing overall infrastructure requirements. In contrast to scaling-up or -out, it focuses on fully utilizing the given hardware instead of demanding more or ever-larger servers. We present Darwin, our scale-in SPE prototype that tailors its execution towards arbitrary target environments through compiling stream processing queries while recoverable larger-than-memory state management. Early results show that Darwin achieves an order of magnitude speed-up over current scale-out systems and matches processing rates of scale-up systems.

## 1 INTRODUCTION

Today's large-scale Internet companies use stream processing engines (SPEs) to process up to terabytes of incoming data per second [1]. However, recent studies show that widely used SPEs such as Apache Flink and Spark Streaming do not fully utilize the underlying hardware and are resource inefficient [26, 27]. Thus, companies must *scale-out* their analytics jobs to millions of cores and tens of thousands of commodity servers. At this scale, large infrastructure teams are necessary to optimize analytics pipelines to maintain such a high level of processing capacity.

We briefly illustrate the required scale with an example from Alibaba's 2020 Singles' Day. At its peak, they processed 4 billion events per second in a Flink cluster with 1.5 million CPUs [1]. Running a general purpose VM (e.g., ecs.g6.4xlarge) with 16 cores on Alibaba's cloud currently costs $1/hour [2]. Scaling this to 1.5 million cores with 93,750 VMs totals at $93,750/hour or $2.25 million for the entire day, just in nominal infrastructure cost.

To overcome resource inefficiency, new *scale-up* SPEs were proposed that, e.g., use query compilation [11], optimize for NUMA-awareness [28], or utilize GPU-CPU co-processing [16], promising up to hundreds of millions of events per second on a single node. To showcase the stark contrast of scale-up to scale-out system performance, we assume a scale-up system with 100 million events per second in our Alibaba example. With this system, the workload could run on 40 VMs with 52 cores each (e.g., ecs.g6.26xlarge at $6/h) for a total of $5760/day [2], resulting in a nearly 400× reduction in price. While this calculation is simplified, it clearly shows the huge gap between what is achieved with current scale-out SPEs and what is possible with proposed scale-up systems.

However, the price to pay for this high performance and reduced infrastructure cost lies in a reduced feature set. While scale-up systems utilize the hardware more efficiently, they lack support for larger-than-memory state and crash recovery, which limits their use in production setups. When the server or application crashes, all state is lost and must be reprocessed. For workloads with small windows, reprocessing includes only a few hours of old data. For unbounded or large-window streaming jobs with global state, reprocessing may span days or weeks of old data, which quickly becomes infeasible.

In our example, additional problems arise that limit the use of scale-up systems. Even with modern high speed networks, it is currently not possible to ingest TB of data per second into a single server. To overcome this limitation, large-scale pipeline must build on scale-out systems, accepting the performance penalty they entail. Thus, we see a huge performance gap between workloads that fit onto a single server and can run in scale-up systems and ones that do not fit onto a single server and must retreat to scale-out SPEs.

In addition to the limitations discussed above, most companies simply do not operate at Internet-scale and process significantly less data. These smaller companies often cannot afford large infrastructure teams to optimize and tune their multi-node pipelines. Due to the resource inefficiency of current systems, this puts them at the disadvantage of still needing clusters or large machines to analyze smaller data volumes. And even for large companies, reducing these costs is highly beneficial as infrastructure is the dominant cost factor in cloud environments [10].

Also, regardless of scale, various business workloads require high availability and cannot afford a full reprocessing after all in-memory data is lost due to a crash in scale-up systems. This again forces users to chose inefficient scale-out systems over highly tuned scale-up SPEs, as production-grade scale-out systems support persistent, larger-than-memory state and crash recovery. However,

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

**Table 1: Feature set of existing SPEs with regard to resource efficiency and state management.**

| System | Scale | Language | State | Recovery | Hardware Resource Efficiency |
|---|---|---|---|---|---|
| Flink [4] | out | JVM | in-memory + persistent | ✓ | – |
| Spark Streaming [25] | out | JVM | in-memory + persistent | ✓ | – |
| Drizzle [23] | out | JVM | in-memory + persistent | ✓ | – |
| Hazelcast Jet [9] | out | JVM | distributed in-memory | ✓ | cooperative multi-threading |
| Briskstream [28] | up | JVM | in-memory | – | NUMA-aware scheduling |
| Saber [16] | up | JVM | in-memory | – | CPU/GPU co-processing |
| Trill [5] | up | C# | in-memory | – | query compilation, row/column layout |
| StreamBox [20] | up | C++ | in-memory | – | NUMA-aware, lock-free |
| Grizzly [11] | up | C++ | in-memory | – | adaptive query compilation, NUMA-aware |

scale-out SPEs rely on slow secondary storage for this, further decreasing overall system performance. Recent developments in storage technology significantly improve the performance of persistent storage devices, allowing us to reduce the gap between high-performance and persistent state in SPEs. Persistent Memory (PMem) offers byte-addressability at close-to-DRAM speed with SSD-like capacity [7, 24] and modern NVMe SSDs achieve up to 7 GB/s and two million random IOPS [13]. Efficiently incorporating these technologies into streaming applications has the potential to radically shift the way SPEs interact with persistent state.

Following both efficient hardware utilization and durable state management, we observe that systems support either one or the other, but not both. In this space, we identify three key challenges current SPEs face, *resource inefficiency*, *state management*, and *overall system optimization*. Overcoming these challenges heavily impacts SPE performance and constitutes an important step towards industry adoption and system maturity.

Based on these challenges, we propose *scale-in* stream processing, a new paradigm that adapts to varying application demands by achieving high hardware utilization on a wide range of hardware setups, reducing overall infrastructure requirements. In contrast to scaling-up or -out, it focuses on fully utilizing the given hardware instead of demanding more or ever-larger servers. Scale-in processing combines scale-out and scale-up concepts to efficiently process streaming data without sacrificing larger-than-memory state, crash recovery. To scale-in, we adapt common scale-up approaches that optimize for the underlying hardware and common scale-out approaches that enable large state management. On the one hand, this allows scale-in SPEs to support large-scale processing when raw processing power is needed. On the other hand, when data volumes are low and infrastructure cost is more important than performance, scaling-in reduces the hardware requirements, resulting in lower infrastructure cost and operational complexity, while still offering key functionality such as crash recovery. Compared to the current performance drop when switching from scale-up to scale-out SPEs, scale-in allows for graceful scaling when workloads exceed single server by optimizing for both single- and multi-node setups.

To this end, we introduce *Darwin*, our scale-in SPE prototype. Darwin leverages query compilation and modern storage to fully utilize the underlying hardware and handle large recoverable state. In summary, we make the following contributions.

1) We propose *scale-in* stream processing, a new paradigm that adapts to varying application demands by achieving high hardware utilization on a wide range of hardware setups, reducing overall infrastructure requirements.
2) We present Darwin, a scale-in SPE prototype that optimizes for high overall hardware utilization while supporting recoverable larger-than-memory state.

The remainder of this paper is structured as follows. In Section 2, we present current challenges in SPEs. In Section 3, we present scale-in processing and its opportunities. We introduce our scale-in SPE prototype Darwin in Section 4 before concluding in Section 5.

## 2 CURRENT SPE CHALLENGES

Recent work in stream processing focuses either on scale-up or scale-out concepts. Scale-up systems optimize for high system utilization while scale-out systems focus on application stability and efficient large state management. Both areas show promising advancements, but combining them has received little attention. In Section 2.1, we compare nine SPEs to see how they offer resource efficiency and state management. From this comparison, we observe that numerous challenges remain in the intersection of scale-up and scale-out systems. In Sections 2.2 to 2.4, we present three major challenges that current SPEs face, *resource inefficiency*, *state management*, and *overall system optimization*.

### 2.1 Focus of Existing Systems

In this section, we briefly discuss the feature sets of common SPEs with regard to resource efficiency and state management. We show the comparison in Table 1. For most features, we observe a clear distinction between scale-up and scale-out systems. While scale-out systems support recoverable, larger-than-memory state with persistent or distributed state management, all scale-up systems support only in-memory state without recovery. Instead, scale-up systems offer optimizations for higher hardware resource utilization on a single server. These include query compilation, NUMA-awareness, lock-free data sharing, and GPU co-processing. Except for NUMA-awareness, none of these are exclusive to large servers and are applicable optimizations also in commodity machines. They represent a large area of improvement for scale-out systems, which currently offer very little hardware optimization.

Finally, all selected scale-out systems target the JVM with managed languages. The group of scale-up systems is not as homogeneous, spanning managed languages and C++. The recent scale-up SPE Grizzly demonstrates large performance gains by using a system language such as C++, outperforming other scale-up and JVM-based systems by orders of magnitude [11]. Thus, using a system-level language is highly advantageous to achieve high utilization when targeting the underlying hardware.

Overall, we observe distinct characteristics for scale-up and scale-out systems. While none of the scale-up systems offer recoverable state management, the scale-out systems neglect hardware optimizations. To achieve high performance and resource efficiency, future SPEs must focus on combining these features. Scaling-up should not come at the price of data loss and scaling-out should not come at the price of poor hardware utilization.

## 2.2 Resource Inefficiency

Recent studies show that widely used scale-out SPEs do not fully utilize the underlying hardware [26, 27]. When designing future SPEs, overcoming this resource inefficiency has great potential to reduce cost and improve performance. Higher resource utilization leads to higher system performance, i.e., higher throughput or lower latency. However, when processing smaller data volumes, scalability is not the primary concern for many users. Efficient server use allows users to reduce the number of required servers while still satisfying their performance needs. This not only reduces infrastructure cost but also overall system complexity.

## 2.3 State Management

Recent scale-up SPEs focus on maximizing hardware utilization through, e.g., query compilation [11], CPU-GPU co-processing [16], or NUMA-awareness [28]. Yet, none of these systems support larger-than-memory state or crash recovery, both important features for industry adoption. We identify efficient state management as a largely uninvestigated topic in scale-up SPEs compared to numerous computational improvements.

Common scale-out SPEs such as Apache Flink use persistent state backends (e.g., RocksDB) to handle larger-than-memory state. However, general-purpose key-value stores do not always fit stream-specific state access patterns [14]. They treat state as a black box, while many streaming-specific patterns are known in advance. Also, currently used general-purpose stores are not optimized for emerging storage technology. Research on modern storage-aware systems shows significant performance gains compared to traditional approaches [3, 17]. Storage-aware and streaming-specific state management presents a wide range of research challenges to improve the overall performance of modern SPEs.

## 2.4 Overall System Optimization

Database systems show that optimizing the overall system brings large performance benefits. Databases are commonly implemented in system languages such as C and C++, which compile to machine code. They are highly tuned towards the underlying system for maximum performance and offer, e.g., hardware-conscious joins and indexes, CPU-optimized scans, or NUMA-aware scheduling.

On the other hand, many widely used SPEs are written in high-level languages such as Java and Scala, targeting the JVM. Especially memory-management has a high performance impact due to, e.g., garbage collection overhead. Also, common SPEs often do not optimize internal operators at the level known from databases. Overall, we observe a major gap between optimization levels in SPEs and database systems. With the increasing maturity of SPEs, reducing this gap is essential to improve the performance of future applications. Fortunately, many operations are similar in databases and SPEs, allowing us to benefit from database optimization research.

# 3 SCALE-IN STREAM PROCESSING

To overcome the current challenges in stream processing and acknowledge the fact that real-world setups have drastically varying performance and availability requirements, we propose *scale-in* processing. Scale-in processing is a new paradigm that adapts to varying application demands by achieving high hardware utilization on a wide range of hardware setups, reducing overall infrastructure requirements. In contrast to scaling-up or -out, it focuses on fully utilizing the given hardware instead of requiring more or ever-larger servers. We identify six opportunities for scale-in SPEs, based on the challenges discussed in Section 2. In Section 3.1, we discuss how query compilation and specialized network communication aid in overcoming general resource inefficiency. In Section 3.2, we present three opportunities to improve state management in scale-in systems: emerging persistent storage media, crash recovery, and streaming-specific access patterns. To increase the overall system performance, we discuss CPU-aware optimizations in Section 3.3.

To achieve high performance, scale-up systems commonly optimize for large high-end servers and scale-out SPEs commonly add more commodity servers. Solving performance limitations by adding more machines results in neglected individual server performance and poor resource utilization in scale-out systems. On the other hand, current scale-up systems are confined to a single server and require ever-larger machines to overcome performance issues. By combining both scale-up and scale-out concepts, scale-in SPEs treat every machine as a server that requires optimization, even if it contains only off-the-shelf components. To this end, scale-in systems adapt their execution to the underlying hardware and the specified queries. For large workloads, scale-in systems achieve the raw performance known from scale-up systems and for medium or small pipelines, they reduce hardware requirements while still offering key functionality such as crash recovery.

## 3.1 Opportunities for Resource Inefficiency

In this section, we discuss how query compilation and network communication aid in overcoming general resource inefficiency.

**Query Compilation.** At the core of scale-in processing, query compilation allows for hardware-conscious optimization on each server. This has many advantages, which are clearly demonstrated in previous work on SPEs [11] and database systems [21]. Compiling queries allows the compiler to optimize the execution for the given CPU without pre-compiling the runtime engine for all possible systems. This enables compiler features such as auto-vectorization and architecture-aware tuning without any development overhead.

To highlight this advantage, we run a short experiment with CPU-specific optimization enabled and disabled. We execute a query in-memory based on Nexmark Q3, containing an auction-like setup with a join and aggregation. We first compile the query generically, i.e., no CPU-aware compiler flags. This represents the general case in which we do not target the underlying hardware and use a generic implementation for all servers. We then enable CPU-optimizations via `-march=native`. On an Intel Xeon Gold 5220S CPU, adding the CPU optimizations achieves a 12% higher throughput without any changes to the code. This shows that even without explicitly writing CPU-optimized code, automatically targeting the underlying hardware is very beneficial.

Additionally, query compilation allows us to integrate query information as compile-time information, enabling additional optimizations. A push-based execution model improves data locality and reduces the number of virtual method calls compared to interpreted execution by compiling the entire execution into a tight loop [21]. Overall, query compilation allows us to tailor the execution exactly to the given query, system, and user requirements. It is the foundation for numerous other optimizations that we describe below, such as CPU optimization and state management.

**Network Communication.** An important component of scale-out SPEs is network communication between the servers. Recent work shows that the network constitutes a performance bottleneck and causes resource inefficiency on the servers [15]. In their study, the authors show that scale-out systems like Flink reach network limits long before they reach the actual saturating data rate. When performing a distributed aggregation across multiple nodes with 1 GBit LAN, Flink sustains only 1.2 million events/s, which amount to only 40 MB or 1/3 of the network bandwidth. While this constitutes the main bottleneck in 1 GBit LAN, today even medium-sized cloud instances have 10 or more GBit/s network connections, matching or surpassing SSD storage bandwidth. Managing this bandwidth with techniques like late merging [26] to reduce data shuffling or user-space networking to reduce TCP/IP overhead [18], enables higher effective bandwidth utilization even for smaller workloads.

For large-scale workloads, advances in network technology drastically improve cross-server communication performance via high bandwidth Infiniband and RDMA connections of up to 200 GBit/s per network card [19]. Current research demonstrates that SPEs benefit from RDMA for data ingestion [26] and that RDMA-based message passing achieves very high throughput with low latency [22]. These two findings show that there is a large potential for optimizing SPEs through fast RDMA connections. Especially in combination with modern byte-addressable storage, such as PMem, this opens new opportunities for more efficient checkpointing, state migration, and recovery approaches.

## 3.2 Opportunities for State Management

In this section, we present three opportunities to improve state management in scale-in systems: emerging persistent storage media, crash recovery, and streaming-specific access patterns.

**Persistent Storage.** Scale-out systems use persistent storage to handle larger-than-memory state. This entails a large performance decrease as storage access is significantly slower than DRAM access. However, new persistent storage technology is closing the gap between slow secondary storage and fast volatile memory. Recent work on PMem storage systems shows that persistency can be achieved with less than 2× performance decrease [3]. Also, fast NVMe SSDs are used in modern database systems to extend storage capacity while still offering close-to-DRAM performance [17].

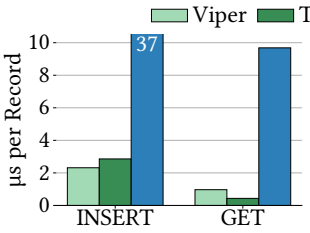


**Figure 1: Insert and get performance.**

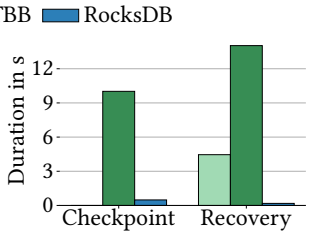


**Figure 2: Checkpoint and recovery duration.**

We show the performance of modern storage systems in Figure 1. We choose Intel's TBB concurrent hash map as a representative of in-memory state management, as it is used in recent scale-up SPEs [11, 20]. We choose RocksDB as a representative of a classical generic byte-based key-value store, as it is used in Apache Flink. Finally, we choose Viper as a representative of modern storage-aware key-value stores, which is based on a hybrid DRAM-PMem index and log structure [3]. Viper stores records directly in a PMem log without intermediate buffering in DRAM. To leverage the higher random access performance of DRAM compared to PMem for index updates, its index is located in DRAM. This hybrid design achieves 4× higher insert rates than PMem-only stores.

We prefill 100 million 50 Byte records before measuring another 100 million inserts/gets with 32 threads. Viper slightly outperforms TBB for inserts and is only ~2× slower for gets. We note that recent work shows TBB to not be the fastest concurrent in-memory storage system [6], but it is used in common scale-up SPEs [11, 20] and serves as an in-memory reference. The clear gap between Viper/TBB and RocksDB shows the major shift in persistent storage performance that systems can leverage. Unlike existing scale-out systems, storage-aware scale-in systems do not need to trade performance for persistence. Fast storage enables both efficient recoverability and high overall throughput.

**Recovery.** Considering server-local state is necessary when restarting an application after a crash and highly beneficial for regular application restarts. Scale-up systems run on high-end servers that contain hundreds of GBs of state. For specialized hardware, replacing the server is not always possible and transferring its state to another server quickly becomes a recovery bottleneck. In this case, state recovery must occur on the same server. Additionally, current scale-out SPEs use server-local state when restarting an application on the same node without a crash, e.g., when re-scaling or deploying a newer version. For both recovery and restarting, it is essential to have persistent state that outlives the application.

We show the advantage of using modern storage technology to achieve efficient checkpointing and recovery in Figure 2. In this microbenchmark, we store 200 million 50 Byte records, i.e., 10 GB raw data, in three different storage instances. As representatives of their respective system classes, we again use TBB, Viper, and RocksDB. To persist TBB's data, we store all entries in a tightly

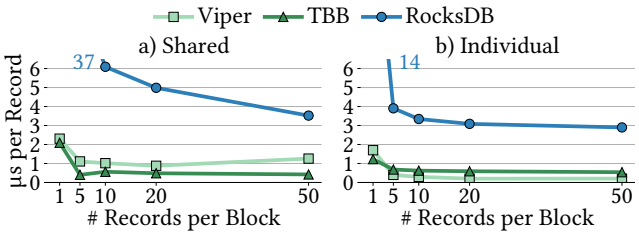


**Figure 3: Grouped state access performance for a) *shared* and b) *individual* storage instances.**

packed byte array in a file stored on SSD. We see that the persistent systems RocksDB and Viper perform a checkpoint very efficiently. RocksDB must only flush its volatile write buffer and Viper must persist only metadata. TBB takes significantly longer, as all in-memory data is converted and copied to secondary storage, which is I/O-bound. For recovery, we see that RocksDB performs best, as it reads only metadata and immediately accepts requests. Viper must recover its volatile index, which depends on the number of entries. TBB reads all data from storage and re-creates its in-memory state, which takes significantly longer than the other two systems.

This experiment shows a large difference in the recovery performance of volatile and persistent state. Expanding on this is a key element of scale-in stream processing, as it impacts both runtime performance while checkpointing and start-up time after a crash. With state in the order of TBs, re-creating in-memory state from secondary storage becomes infeasible.

**Streaming-specific State Access.** In addition to storage-aware state management, streaming-specific access patterns improve performance even on slow storage media. We demonstrate this based on current behavior in Apache Flink. In Flink, windowed operations store incoming events in a list of records belonging to a given window. When using the RocksDB backend, the operator gets the current value, deserializes it, appends the new value, and serializes the updated list back to its byte representation. This incurs an unnecessarily high overhead for each record. As the records are not needed immediately and are accessed only as a list, they can be buffered in small in-memory lists before writing to RocksDB.

We show the effects of buffering records in Figure 3. In this experiment, we prefill 100 million 50 Byte records before performing another 100 million inserts with 32 threads. We distinguish between a shared instance, in which all threads operate on the same store, and individual instances, in which each thread has its own store. We store the records in small blocks, consisting of up to 50 records. We observe that Viper outperforms TBB for individual instances but performs worse for the shared one. This difference is important when designing streaming state, as it can either be shared across operator instances, e.g., in Grizzly [11], or partitioned by key, e.g., in Flink. Our results show that depending on the underlying storage and chosen system, one or the other is more beneficial. More importantly, RocksDB performs between 14–20× worse than TBB when storing individual records, showing the high overhead of persistent storage. However, compared to individual records in TBB, 50 grouped records are only 2–3× slower in RocksDB. This shows that streaming-specific access significantly improves state performance, even for low-end storage media.

## 3.3 Opportunities for System Optimization

In this section, we discuss CPU-aware optimizations to increase the overall system performance.

**CPU-aware Optimization.** Poor CPU utilization is a major contributor to resource inefficiency in current scale-out SPEs [27]. To overcome this, scale-in SPEs target the system's CPU to achieve higher overall utilization. Recent work shows the potential of adapting OLAP queries towards a given workload and system setup [12]. Exploring different computation modes, such as compiled or vectorized execution, is heavily researched in databases. However, they have received little attention in SPEs so far. Transferring these concepts to SPEs has the potential to further increase the overall system performance. For example, storing network-buffered records in a row or column format depending on the data and query allows for a performance trade-off between processing time and ingestion rate, while also enabling scalar or vectorized execution modes.

Another optimization is based on simultaneous multithreading (SMT). Depending on the workload, using SMT hides memory access latency while not using it improves cache locality. When multiple operators have low CPU consumption, they are placed on the same core to achieve better utilization. When CPU utilization is high, features such as explicit SIMD instructions achieve higher throughput with the same utilization. While query compilation generally targets the underlying CPU, explicit optimizations and domain knowledge additionally improve performance.

# 4 INTRODUCING DARWIN

In this section, we present Darwin, our scale-in SPE prototype. Darwin treats each server it runs on like a scale-up system by fully utilizing the given hardware configuration. To this end, it uses query compilation to generate efficient execution plans for each query targeting the server's hardware. This targeting currently includes storage-aware state management and CPU-specific optimizations. As Darwin is still in early stages, we plan to add support for more opportunities discussed in Section 3, e.g., network-aware data transfer to achieve efficient multi-server processing or efficient checkpointing and recovery mechanisms.

## 4.1 Darwin Architecture

In this section, we present Darwin's high-level architecture, components, and execution flow, as shown in Figure 4.

**Data Pipeline.** Users create queries via a data pipeline object, which currently offers an SQL-like API inspired by Apache Flink's Table API [8]. Additionally, the user configures runtime options such as the compiler to use, which storage medium to use for state, and which architecture to optimize for. To run on heterogeneous hardware without manually adapting for each server, this config also auto-detects system characteristics.

**Query Plan.** From the query, a query plan is created. The query plan represents the logical version of the query, similar to relational algebra for classical database queries. Compared to relational algebra, it requires a few additions, e.g., for windowing logic or external I/O. The query plan is the first step in the execution in which optimizations are performed, e.g., predicate push-down.

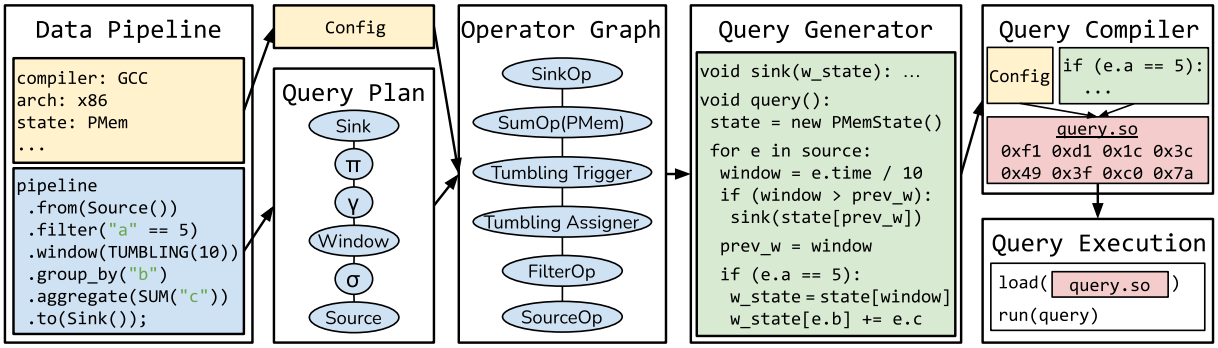


**Figure 4: Darwin's architecture and execution flow.**

**Operator Graph.** Together with the config, the query plan is translated to an operator graph. We briefly describe the translation based on the query plan shown in Figure 4. Starting from the sink, each node recursively translates its input node. The resulting operator graph represents the physical operators, i.e., the specific implementation chosen for the given query, system, and config. As start and end nodes, source and sink operators require special treatment. They contain buffering logic, e.g., for external network-based I/O, and either have no input or output operators. After the source is translated, the selection node is translated to an equality-filter operator. The translation of window nodes requires reordering, as windowing logic impacts the operator order. The tumbling window assignment and trigger run before the aggregation, but count-based triggers run afterward. Thus, window translation requires splitting and distributing certain nodes across the operator graph. For the aggregation, the translator chooses a hash-based sum aggregation operator with state in PMem, as specified by the user.

**Query Generator.** The query generator takes the operator graph and generates a C++ string representation of the actual query. In Figure 4, we show a pseudo-code version of the produced code. When the FilterOperator is called, it receives a record with schema information. From this, it generates a conditional statement with the correct predicate (e.g., equality) based on the filtered attribute (e.g, `e.a`). Depending on the predicate and execution model, the filter operator can also produce vectorized or SIMD-based filters, allowing for more fine-grained hardware optimization. Afterward, it calls the downstream *tumbling assigner*. The assigner generates code to assign the record to a tumbling window by mapping the record's timestamp to a window key. This process is continued until all operators are called and the query is fully generated.

As the windowed aggregation buffers data, it represents a pipeline breaker [11, 21]. The sink operator is executed after the aggregation is complete, i.e., when the window is triggered. The query generator creates a new function for the sink, which represents a new pipeline that can be executed independently. Splitting pipelines allows us to independently scale sources, sinks, and other operators.

As the query generation contains runtime information such as data types or filter conditions, the generated query is optimized accordingly. The SumOperator knows the key and value types, so it instantiates a state object with them. This is an advantage over key-value store interfaces such as RocksDB in Flink, which operate on generic byte representations. Storing records with explicit type information removes serialization and deserialization overhead and allows the compiler to optimize data move instructions, e.g., by issuing SIMD loads/stores instead of regular 8 Byte `mov`s [3].

**Query Compiler.** Once the query code is generated, the query compiler compiles it. It uses information in the config to target the underlying hardware, e.g., by enabling vectorization features of the CPU. As the compiler runs independently of Darwin, users can specify a different compiler than was used to compile Darwin. Darwin can be compiled once and distributed while still providing flexibility towards the system it executes queries on. The code is compiled into a shared library that is dynamically loaded by Darwin during execution. This allows Darwin to interact with the query, e.g., when passing allocated memory, data, or other resources.

**Query Execution.** In the last step, the query is loaded and executed. Depending on the generated pipeline and specified parallelism, multiple source, sink, and operator instances are started. During execution, Darwin monitors the performance of the query to allow for changes in parallelism and thread placement. If pipelines have low utilization, they are merged to free resources. If pipelines are creating backpressure, Darwin splits them to keep up with the data rate. This approach allows for some flexibility during runtime when data loads vary or are skewed. It also supports adaptive changes to the query if gathered performance metrics and data characteristics allow for more aggressive optimization [11].

## 4.2 Performance

We compare the performance of Darwin with the state-of-the-art scale-up SPE Grizzly [11] and the widely used scale-out SPE Apache Flink [4]. In this experiment, we run a 60-second tumbling window sum aggregation on 32 Byte records with 15000 unique keys. Our server contains an Intel Xeon Gold 6240L CPU with 18 cores, 96 GB DRAM, and 1.5 TB (6× 256 GB) Intel Optane DC Persistent Memory 100 Series. The experiments are run with 32 threads. For the in-memory version, Grizzly and Darwin use the TBB concurrent map. For the persistent version, Darwin uses the hybrid DRAM-PMem key-value store Viper [3] and Flink uses RocksDB.

We show the results in Figure 5. On the left, we see that Darwin performs equally to Grizzly for in-memory processing, as both systems are limited by TBB to store the aggregations. We note that this is not the highest performance that Grizzly can achieve, but the additional optimizations proposed by the authors are orthogonal to the basic concept and could also be applied to Darwin.

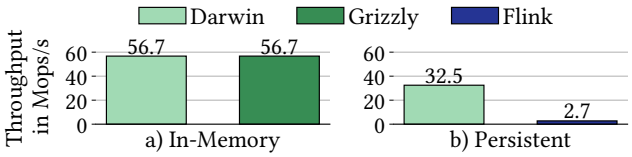


**Figure 5: Throughput of Darwin, Grizzly, and Flink.**

On the right, we see that Darwin outperforms Flink by over an order of magnitude (12×). Additionally, in-memory Darwin is only 1.7× better than the persistent version, which is in line with the PMem–DRAM gap presented in the Viper paper [3]. This shows that modern storage significantly closes the gap between volatile and existing durable state management, enabling us to efficiently support larger-than-memory state. Overall, our results show that Darwin achieves both state-of-the-art scale-up performance and an order of magnitude improvement over existing larger-than-memory scale-out systems.

## 5 CONCLUSION

To bridge the gap between performance and core features, we propose *scale-in* stream processing and present our prototype system Darwin. By achieving high hardware utilization on a wide range of hardware setups, scale-in systems adapt to application-specific demands. Combining scale-out concepts with advancements in persistent storage and scale-up concepts that focus on the underlying hardware, scale-in processing achieves high system utilization without sacrificing key features for industry adoption, such as recoverable, larger-than-memory state. Our scale-in SPE prototype Darwin uses query compilation and storage-aware state management to match state-of-the-art scale-up performance while outperforming existing scale-out systems by an order magnitude. Scale-in processing enables application-proportional scaling of server requirements, making it economical for all levels of performance needs.

## ACKNOWLEDGMENTS

This work was partially funded by the German Ministry for Education and Research (ref. 01IS18025A and ref. 01IS18037A), the German Research Foundation (ref. 414984028), and the European Union's Horizon 2020 research and innovation programme (ref. 957407).

## REFERENCES

[1] Alibaba. 2020. Four Billion Records per Second! www.alibabacloud.com/blog/four-billion-records-per-second-stream-batch-integration-implementation-of-alibaba-cloud-realtime-compute-for-apache-flink-during-double-11_596962
[2] Alibaba. 2021. Elastic Compute Service. www.alibabacloud.com/product/ecs
[3] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proceedings of the VLDB Endowment* 14, 9, 1544 – 1556. https://doi.org/10.14778/3461535.3461543
[4] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. 2015. Apache Flink(TM): Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4, 28–38.
[5] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: a high-performance incremental query processor for diverse analytics. *PVLDB* 8, 4, 401–412. https://doi.org/10.14778/2735496.2735503
[6] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD '18*. ACM, 275–290. https://doi.org/10.1145/3183713.3196898
[7] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *SIGMOD '21*. ACM. https://doi.org/10.1145/3448016.3457292
[8] Apache Flink. 2021. Table API & SQL. ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/table/overview
[9] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile. *arXiv:2103.10169 [cs]*.
[10] Albert Greenberg, James Hamilton, Dave Maltz, and Parveen Patel. 2009. The Cost of a Cloud: Research Problems in Data Center Networks. *Computer Communications Review* 39, 1.
[11] 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 *SIGMOD '20*. ACM, 2487–2503. https://doi.org/10.1145/3318464.3389739
[12] Tim Gubner and Peter Boncz. 2021. Charting the design space of query execution using VOILA. *PVLDB* 14, 6, 1067–1079. https://doi.org/10.14778/3447689.3447709
[13] Intel. 2021. Optane SSD P5800X. www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html
[14] Vasiliki Kalavri and John Liagouris. 2020. In support of workload-aware streaming state management. https://www.usenix.org/conference/hotstorage20/presentation/kalavri
[15] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Processing Engines. In *ICDE*. IEEE, 1507–1518.
[16] 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 '16*. ACM, 555–569. https://doi.org/10.1145/2882903.2882906
[17] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE '18*. IEEE, 185–196. https://doi.org/10.1109/ICDE.2018.00026
[18] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment* 13, 7, 1091–1104. https://doi.org/10.14778/3384345.3384356
[19] Mellanox. 2021. 200Gb/s ConnectX-6 Ethernet Single/Dual-Port Adapter IC. www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic
[20] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn Mckinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *ATC*. USENIX Association, 617–629.
[21] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9, 539–550. https://doi.org/10.14778/2002938.2002940
[22] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: The Data Flow Interface for High-Speed Networks. In *SIGMOD '21*. ACM, 1825–1837. https://doi.org/10.1145/3448016.3452816
[23] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *SOSP '17*. ACM, 374–389. https://doi.org/10.1145/3132747.3132750
[24] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST '20*. USENIX Association, 169–182.
[25] Matei Zaharia, Scott Shenker, Haoyuan Li, Tathagata Das, Timothy Hunter, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP '13*. ACM, 423–438. https://doi.org/10.1145/2517349.2522737
[26] 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. *Proceedings of the VLDB Endowment* 12, 5, 516–530. https://doi.org/10.14778/3303753.3303758
[27] 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 '17*. 659–670. https://doi.org/10.1109/ICDE.2017.119
[28] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *SIGMOD '19*. ACM, 705–722. https://doi.org/10.1145/3299869.3300067