Scabbard: Single-Node Fault-Tolerant Stream Processing

Georgios Theodorakis
Imperial College London
grt17@imperial.ac.uk

Fotios Kounelis
Imperial College London
f.kounelis20@imperial.ac.uk

Peter Pietzuch
Holger Pirk
Imperial College London
{prp,pirk}@imperial.ac.uk

ABSTRACT

Single-node multi-core stream processing engines (SPEs) can process hundreds of millions of tuples per second. Yet making them fault-tolerant with exactly-once semantics while retaining this performance is an open challenge: due to the limited I/O bandwidth of a single-node, it becomes infeasible to persist all stream data and operator state during execution. Instead, single-node SPEs rely on upstream distributed systems, such as Apache Kafka, to recover stream data after failure, necessitating complex cluster-based deployments. This lack of built-in fault-tolerance features has hindered the adoption of single-node SPEs.

We describe Scabbard, the first single-node SPE that supports exactly-once fault-tolerance semantics despite limited local I/O bandwidth. Scabbard achieves this by integrating persistence operations with the query workload. Within the operator graph, Scabbard determines when to persist streams based on the selectivity of operators: by persisting streams after operators that discard data, it can substantially reduce the required I/O bandwidth. As part of the operator graph, Scabbard supports parallel persistence operations and uses markers to decide when to discard persisted data. The persisted data volume is further reduced using workload-specific compression: Scabbard monitors stream statistics and dynamically generates computationally efficient compression operators. Our experiments show that Scabbard can execute stream queries that process over 200 million tuples per second while recovering from failures with sub-second latencies.

PVLDB Reference Format:
Georgios Theodorakis, Fotios Kounelis, Peter Pietzuch, and Holger Pirk. Scabbard: Single-Node Fault-Tolerant Stream Processing. PVLDB, 15(2): 361 - 374, 2022.
doi:10.14778/3489496.3489515

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lsds/LightSaber.

1 INTRODUCTION

By 2025, 30% of all data is likely to be analyzed in real-time [91]. Therefore, it is not surprising that stream processing is quickly becoming the fourth important data-intensive application workload (next to transaction processing, reporting, and online analytics). Stream processing enables applications ranging from real-time credit card fraud detection [36] to click-stream analytics [2, 20, 44], and live mining of sensor data [28, 29]. Given future data volumes and velocities, high throughput and low latency performance are key requirements for stream processing.

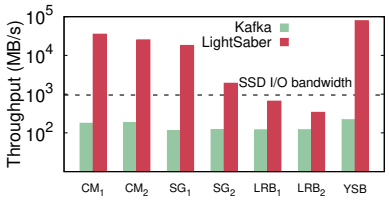


Figure 1: Data ingestion rates for stream queries in a single-node SPE (LightSaber) vs. a persistent message queue (Apache Kafka)

To accommodate growing data amounts, distributed stream processing engines (SPEs) such as Flink [19] and Spark Streaming [108] scale out processing to a cluster of nodes through appropriate data partitioning [19, 108] – at substantial operational cost. With the rise of parallel hardware, such as multi-core CPUs and GPUs, we witness scale-up designs for single-node SPEs [65, 76, 77, 96, 110] that rival the performance of cluster-based deployments. While high-speed networking such as RDMA [16, 59] provides 200 Gbps per-port bandwidth with microsecond latencies [11], which allows for fast stream ingestion and remote storage [63], existing cluster-based SPEs cannot saturate these fast interconnects [109]. In contrast, single-node SPEs yield up to an order of magnitude higher performance with fewer resources and lower maintenance costs [86]. Such high execution efficiency is achieved by avoiding abstractions for distributed processing and incorporating techniques such as just-in-time (JIT) code generation [47, 96].

Despite these advantages, single-node SPEs have seen limited adoption in practice due to a lack of fault-tolerance mechanisms that guarantee correct results after system failure [6, 53, 94]. Existing cluster-based SPEs achieve at-least-once or exactly-once delivery semantics by persisting input tuples along with the computational state [19, 37] or logic [107]. Systems typically offload persistence to external distributed messaging systems such as Kafka [7], Kinesis [4] or Pulsar [87], or stores such as RocksDB [35] or Faster [21]. The use of external systems for persistence introduces overheads [33, 89, 90] that increase the size of scaled out deployments.

While the same persistence approaches could be used for single-node SPEs, relying on an external cluster-optimized system for persistence, such as Kafka, counteracts the benefits of a single-node deployment. A single Kafka node cannot support the performance requirements of modern single-node SPEs. To illustrate the magnitude of the problem, Fig. 1 shows the difference in ingestion throughput for a set of real-world stream queries [96] between LightSaber [96], a high-performance single-node SPE with query

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489515

compilation, and Kafka, a popular persistent message queue system. As the results show, a single Kafka node can only ingest data streams at rates that are several orders of magnitude lower than LightSaber's query performance and does not even saturate the SSD bandwidth (indicated by a dashed line). While it is possible to scale out the Kafka deployment to increase its throughput linearly through stream partitioning, this requires a large cluster (with associated maintenance costs) just to support a single SPE node.

A strawman solution is to design a "self-contained" fault-tolerance mechanism for a single-node SPE in which the SPE persists all input data streams (and temporary processing state) to stable storage to recover processing after failure. We observe that, for such an approach, disk I/O bandwidth becomes the limiting factor for a majority of queries in Fig. 1, capping performance to 950 MB/s. While I/O bandwidth can be increased through hardware solutions (e.g., NVMe SSDs [106] or RAID [82]), this also increases costs.

Our goal is, thus, to design and implement a single-node fault-tolerant SPE whose fault-tolerance mechanism (i) accounts for the limited available I/O bandwidth (especially when using remote storage [3]); (ii) has a low impact on processing performance without failures; and (iii) allows fast recovery after failures. Our key idea is to reduce the required disk I/O bandwidth by tightly integrating *stream* and *state persistence* with the *operator dataflow graph* of the query. This way, the SPE can apply workload-specific optimizations to (a) reduce I/O bandwidth by only persisting stream and operator state after high selectivity operators have executed and (b) compress data before persistence with query-specific compression.

We describe Scabbard, a new single-node fault-tolerant SPE that provides exactly-once semantics without compromising processing throughput. Scabbard's query execution engine is based on LightSaber [96], a state-of-the-art SPE that uses JIT query compilation and balances parallelism and incremental processing for windowed stream queries. Scabbard's fault-tolerance approach is to persist input streams and transient operator state to an SSD. Here Scabbard makes the following three novel contributions:

(i) Persistent operator graph model. Scabbard introduces a new *persistent operator graph* model that allows for *workload-aware* decisions about data persistence: operators can be reordered to reduce the needed I/O bandwidth for persistence. Based on query characteristics, Scabbard "pushes persistence up",[1] i.e., pruning data with high-selectivity operators. To enable parallelism when persisting streams and operator state, the persistent operator graph uses two abstractions: persistent streams, or *p-streams*, and fault-tolerant operators, or *ft-operators*. P-streams are reliable FIFO channels that support the parallel logging of streams; ft-operators enable the parallel checkpointing and recovery of stateful operators' state. To coordinate persistence decisions, the persistent operator graph uses control tuples (*markers*) that flow between operators and trigger storage and garbage collection operations.

(ii) Query-specific adaptive compression. To further reduce the persisted data, Scabbard compresses p-streams by generating custom compression operators, taking stream statistics (e.g., data ranges, sequences of equal values, etc.) into account. This exposes the trade-off between the computational cost of a compression algorithm and the compression benefit in terms of saved I/O bandwidth. Scabbard then selects a suitable compression algorithm (e.g., run-length encoding, null suppression, delta-encoding, etc.) and inserts compression operators dynamically into the persistent operator graph. The choice of compression algorithm is adaptive: when the statistics of the p-stream change, Scabbard switches to a new compression algorithm while processing.

(iii) Efficient failure recovery mechanism. Scabbard achieves sub-second recovery latencies by reducing the data loaded from storage. It persists the ft-operator state frequently with low overhead through asynchronous checkpointing. To avoid the overhead of query compilation during recovery, Scabbard also stores the optimized code of compiled queries in a native binary format. To recover only the minimum data, persisted data is garbage collected when the dependent results have been emitted or persisted.

Our evaluation shows that Scabbard introduces less than 30% overhead in processing throughput compared to no fault-tolerance. On a 16-core server, it processes over 200 million tuples per second with 8 ms latency (95^{th} percentile) and recovers below a second. It outperforms Apache Flink, a state-of-the-art fault-tolerant SPE, by at least an order of magnitude for all our benchmarks. Scabbard achieves stream persistence similar to a 20-node Kafka cluster with 3× lower 95^{th} percentile latency. It achieves a throughput of up to 10.5 GB/s using 100 Gb/s InfiniBand with RDMA for stream ingress.

2 FAULT-TOLERANCE IN STREAM PROCESSING

We begin with a discussion of fault-tolerance approaches for stream processing. First, we describe our failure model (Sec. 2.1) and how fault tolerance is realized in SPEs (Sec. 2.2). We then formalize the stream processing model that the paper assumes (Sec. 2.3).

2.1 Failure model

SPEs [19, 37, 53, 65, 108] execute continuous queries that translate into *operator graphs*, $q = (O, S, B)$, where O is a set of *operators*, S is a set of *streams* and B is a set of *feedback channels* for sending acknowledgments to operators. The graph's nodes represent the operators; both the streams and the feedback channels are directed edges (i.e., FIFO communication channels). Every graph has special operators that act as *sources* and *sinks* by subscribing to input streams or committing results externally.

The operators of such a graph can be stateless (e.g., `SELECTION`) or stateful (e.g., `AGGREGATION`) and maintain arbitrary state, usually defined with finite windows [8] of tuples. However, given typical failure rates in large data centers [42], stateful operators pose a challenge for providing correct results under failure.

We consider failures that cause an SPE node to *fail-stop* [34]. We assume that an SPE node is connected to external sources/sinks via a reliable network and has access to storage that survives failures (e.g., flash storage [3, 43, 48, 66]).

Upon failure, operators must resume processing from the point at which they failed. For stateful operators, recovery requires redundant storage [97]: of the computational logic to replay past tuples; and of the computational state [53] to avoid replay if the state can depend on the entire stream history.

SPEs can achieve high availability [52] using *passive standby*, *active standby*, or *upstream backup*: with passive standby, streams

[1] We use a relational view in which "up" means closer to the output.

(and state) are maintained in stable storage or the memory of another node; with active standby, redundant nodes are deployed that receive and process the same streams as the primary ones; with upstream backup, each node retains its output and, in case of failure, restores the downstream node's state by replaying it.

To mask the effects of failure fully, an SPE must remove duplicate tuples when restoring state. Fragkoulis et al. [40] distinguish between *exactly-once state* and *exactly-once output*, with only the latter avoiding duplicates. Providing exactly-once output is also referred to as the output commit problem [34], precise recovery [52] or strong productions [2].

2.2 Failure recovery in SPEs

We now examine how SPEs achieve fault-tolerance with exactly-once output and discuss the challenges for single-node designs. The four most common fault-tolerance approaches are:

(i) Transaction-based: Trident [98, 99] and MillWheel [2] remove duplicates. They assign unique identifiers to tuples and commit state updates or produced tuples to an external transactional store [23].

(ii) Lineage-based: Spark Streaming [10] tracks and persists the input/output dependencies of operators (i.e., lineage [107]) before execution. Using the failed operator's lineage, Spark restores the previously computed state by re-executing tasks.

(iii) Checkpointing: Flink [18] uses a distributed protocol for global checkpointing that asynchronously persists operator state with epochs (*aligned checkpoints*). Other approaches [37, 38, 79] log tuples from streams for better runtime performance at the expense of higher recovery times [40], called *unaligned checkpoints*. With an embedded key/value store, such as RocksDB [35], Flink also supports incremental checkpoints [97].

(iv) Changelog-based: To enable state recomputation without persisting state dependencies, Kafka Streams [53] persists state metadata in a changelog, which is stored in Kafka [67]. Although its design combines computation with storage, the use of Kafka as the messaging system between operators increases latency.

While these approaches offer strong guarantees under failures in a distributed deployment, they face limitations for single-node SPEs. First, they rely on external messaging systems [67, 87] to create fault-tolerant sources. These messaging systems require non-trivial tuning [17, 32] and do not maintain compact representations of stream data, which can lead to higher recovery times [74]. Second, they use key/value stores for state management that are often not designed for stream applications [58]. This limits performance [46, 60, 73] and misses optimization opportunities.

2.3 Stream processing model

Following the semantics of the *continuous query language* (CQL) [8], we adopt a relational stream model with *windows*.

Data model. A *stream* s is an infinite sequence of *tuples*, $t \in s$. Each tuple $t = (\varepsilon, \tau, p)$ has: an event timestamp $\varepsilon(t) \in \mathcal{E}$ that denotes when the event occurred, where $\mathcal{E}$ is an ordered time domain of discrete non-negative integer values; a logical timestamp $\tau(t) \in N^+$ assigned by a monotonically increasing logical clock at each operator upon receipt [37]; and p, a sequence of values of primitive data types. We assume that tuples in a stream arrive *in-order* based on their event timestamps.

Operator model. Each operator o receives tuples from n upstream operators to its input queues, $I = \{s_1, ..., s_n\}$. It then applies an operator function f, and produces tuples for its downstream operators stored in a result buffer, denoted by R. An operator keeps track exchanged tuples with two progress vectors, PV^{in} and PV^{out} [105]; stateful operators have processing state Θ. For ease of presentation, we denote an operator snapshot as $C = (I, R, \Theta, PV^{in}, PV^{out})$ and use the notation C_{τ_e} to indicate that it has all values up to τ_e.

An operator function f is composed of: a state transition function ρ that accepts the current state Θ_i and an input tuple t_i and yields the new state Θ_{i+1}; and an output function ω that accepts a state and an input tuple and outputs one or more[2] tuples $\langle t_j, ..., t_{j+x} \rangle$.

We consider queries that use *window* functions over streams to transform them into finite sequences, called *window fragments* [65]. For efficient operator parallelization [65, 96] without depending on distinct keys, every state transition function is decomposed into: (i) a *fragment* function ρ^f that processes a sequence of fragments and produces immutable partial results; and (ii) an *assembly* function ρ^α that constructs and reorders complete window results. Each operator generates computational tasks by bundling fixed-sized data *batches* from its inputs with ρ^f and ρ^α.

3 SCALE-UP PERSISTENCE

A single-node SPE has limited disk bandwidth, disk space, and CPU capacity. Therefore, its fault-tolerance mechanism must persist only the required parts of streams and operator state to enable recovery. Selecting what to store or how to manage persistence and recovery operations are non-trivial tasks, which are highly query- and input-specific. In particular, which data to persist and discard cannot be determined statically and instead requires runtime knowledge about e.g., the data lifetime and its distribution.

Our idea is to exploit the information of the operator graph to enable optimizations related to persistence by encoding it in a structure that we call a *persistent operator graph* (*POG*). The POG contains aspects of both query compile-time and runtime. Fig. 3 shows the POG for the third LR Benchmark query [9] (defined in Sec. 6.1, listed in Fig. 2). The POG extends the operator graph with two new compile-time abstractions (shown in red) and two coordination abstractions (shown in green). Through its compile-time abstractions, *persistent streams* (*p-streams*) and *fault-tolerant operators* (*ft-operators*), a POG supports stream and state persistence. As these persistence operations require runtime coordination to achieve consistency when recovering, the *POG* also introduces *persistence units* (*p-units*) and a set of coordination *markers*. A p-unit is a batch of data (i.e., a finite subsequence of a stream or operator state) with associated metadata (i.e., lineage information), which are persisted, recovered and discarded atomically. Markers are special control tuples in a stream [18, 22] that coordinate the flow between operators and trigger persistence and removal operations.

[2] e.g., join operators produce multiple output tuples per input tuple

Table 1: SCABBARD fault-tolerance abstractions

Entity	Definition in C++ notation	Description
PStream	`void subscribe(Operator, ReaderId, Offset)`	Subscribes an operator as reader to the channel
	`promise<void> write(PUnit<Tuple>, BatchId, Offset, isPersistent)`	Adds data to the channel
	`PUnit<Tuple> read(ReaderId)`	Returns the next available tuples
	`promise<void> checkpoint(BatchId, CheckpointId)`	Writes data to stable storage
	`void trimFrom(ReaderId, Offset, isPersistent)`	Removes data from the channel
	`promise<void> loadStream(PUnit<Tuple>, BatchId)`	Loads data to the channel from stable storage
FTOperator	`void prepareCheckpoint(CheckpointId)`	Mark the streams/state that are going to be checkpointed
	`promise<void> checkpoint(PUnit<Tuple>, BatchId, CheckpointId)`	Writes state to stable storage
	`promise<void> loadState(PUnit<Tuple>, BatchId, CheckpointId)`	Loads state from the last valid checkpoint
	`void sendAck(FTOperator)`	Sends a retain marker to an upstream operator
	`void setDependencies(PUnit<Tuple>)`	Calculates the data dependencies of a PUnit
	`list<Offset> getLatestOffsets()`	Returns a list of the latest offsets it has received
PUnit	`void compress((…)->{…}, StorageBuffer, Index)`	Compresses tuples with a given compression function
	`void decompress((…)->{…}, StorageBuffer, Index)`	Decompresses tuples with a given decompression function
	`list<Offset> getDependencies()`	Returns a list of data dependencies
POG	`void insertRetainMarker(PStream)`	Sends a retain marker to a channel
	`void insertCheckpointMarker(FTOperator)`	Sends a checkpoint marker to an operator

```
(select timestamp, vehicle, highway, direction, segment, count(*)
from SegSpeedStr [range 30 slide 1]
group by highway, direction, segment, vehicle) as R  --
select timestamp, highway, direction, segment, count(vehicle)
from R group by highway, direction, segment
```

Figure 2: LRB_3 query in CQL

3.1 Persistence abstractions

The compile-time abstractions of POGs expose operations for the persistence management of streams and state in an operator graph, which can be executed in parallel at runtime. These abstractions are accessible through intuitive interfaces (summarized in Table 1):

A **persistent stream (p-stream)** provides a reliable FIFO communication channel between two operators in the POG, supporting asynchronous stream persistence at the granularity of p-units. For example, p-streams can be used as ingress streams to persist the incoming data in the absence of a fault-tolerant stream source. Every p-unit in a p-stream is assigned a monotonically increasing logical timestamp $\tau(t)$, which maps to the logical position of the first tuple in the stream. If a p-stream is marked for persistence with coordination markers (see Sec. 3.2), its data becomes available only after it is stored to disk.

While a p-stream is related conceptually to the well-known notion of *upstream backup* [52], upstream backup persists only ingress streams (an external system usually creates backups). In contrast, a POG allows persistence anywhere in the operator graph, which enables new optimizations (see Sec. 3.4).

The p-stream interface allows to `subscribe` to, `write` to and `read` from its channel (see Table 1). As multiple operators can subscribe to a single p-stream, a p-stream tracks their progress using a stream `Offset`.

A **fault-tolerant operator (ft-operator)** is an operator with support for consistent checkpointing and recovery through progress tracking. In Table 1, we describe its interface to create `checkpoints` and `loadState` from the last snapshot. In general, ft-operators can be stateless ($\Theta = \emptyset$), e.g., `PROJECTION` (π), `SELECTION` (σ), or stateful, e.g., `AGGREGATION` (α), `GROUP-BY` (γ), `JOIN` ($\bowtie$). For stateful operators, the ft-operator partitions its state Θ into immutable p-units, which can be persisted to and recovered from storage.

Tracking data dependencies, however, poses a challenge as a p-unit may contribute to multiple results. An ft-operator solves this problem by computing the dependencies between p-units. It

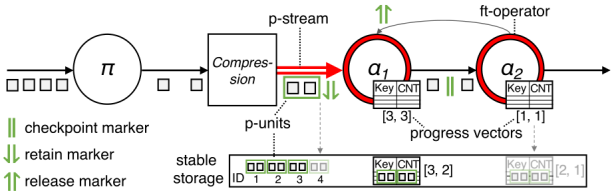


Figure 3: Persistence operator graph (POG) with p-streams, ft-operators, and markers for LRB_3 query

attaches a lightweight graph structure using `setDependencies`, and the dependencies are calculated based on the logical timestamps, the input ordering and the window semantics (similar to Timestream [88] or D-Streams [108]). Thus dependencies capture the relationship: (i) between p-units from different streams (i.e., stream-to-stream dependencies); and (ii) p-units from state and streams (i.e., state dependencies). The logical timestamps of the graphs can be serialized to vector clocks *VC* [75], which determine the event ordering upon recovery.

Although every deterministic operator in the POG can become an ft-operator, checkpointing overhead can be traded-off against recovery time by replacing only the most downstream operators with ft-operators. To prevent inconsistent operator state after a failure [34], if an operator is marked as fault-tolerant, the POG replaces all its downstream operators with ft-operators.

3.2 Persistence and recovery coordination

After a failure, the SPE must recover and recompute the data required to recreate the POG's operator state, which is challenging for exactly-once semantics. To manage the operations required to achieve this on a single-node SPE, we introduce a *persistence protocol* with *markers*. *POGs* support three types of markers: (i) *checkpoint* markers trigger operator checkpoints; (ii) *retain* markers mark a p-unit in a p-stream for persistence; and (iii) *release* markers signal that a specific p-unit is no longer required for recovery. For state recovery, the protocol uses consistent snapshots. Following the state recovery, all data that is not part of the last checkpoint must be replayed, while tuples already produced are dropped. The persistence protocol has five asynchronous primitives, shown in Alg. 1 and described next.

Consistent checkpoint coordination is achieved by *checkpoint* markers similar to the Chandy-Lamport algorithm [22]. They are

Algorithm 1: POG's persistence protocol executed by operator o

```
1  init                                                          ▷ Initialize local variables
2      C = (I, R, Θ, PV^in, PV^out) ← ({∅}, ∅, ∅, {0}, {0})
3      U ← {o_i, ..., o_{i+x}},  D ← {o_j, ..., o_{j+y}}        ▷ Upstream and downstream
         operators
4      snapshot ← ∅,  marked ← ∅,  taskQueue ← ∅,  persist ← {false}
5  upon receive < marker > from in ∈ I
6      (type, VC) ← marker
7      if type = checkpoint then
8          marked ← marked ∪ in
9          if |marked| = 1 then                                  ▷ The first marker overtakes all t ∈ I or R
10             broadcast(D, marker),  snapshot ← snapshot ∪ I ∪ R ∪ Θ
11         if marked = I then                                    ▷ Store to disk when all markers are received
12             taskQueue ← taskQueue ∪ checkpointTasks(snapshot)
13             snapshot ← ∅,  marked ← ∅
14     else if type = release then                               ▷ Remove obsolete data from C
15         C = C \ C_{VC[D]}, broadcast(U, marker)
16     else persist[in] ← true
17 upon receive < punit > from in ∈ I
18     (tuples, offset, VC) ← punit
19     if offset > PV^in[in] then                                ▷ Persist channels that have not sent a marker
20         if |marked| ≠ 0 ∧ in ∉ marked ∧ persist[in] = false then
21             snapshot ← snapshot ∪ punit
22         (Q, id, offset) ← in
23         taskQueue ← taskQueue ∪ persistTask(Q, tuples, id, offset, persist[in])
24         in ← (Q, id + 1, offset + |tuples|)
25         PV^in[in] ← PV^in[in] + |tuples|,  persist[in] ← false
26 upon receive < notification > from in ∈ I
27     taskQueue ← taskQueue ∪ queryTask(ρ^f, ρ^α, I, R, Θ)
28 upon receive < notification > from R
29     for out ∈ D do                                            ▷ Simplified version of sending data downstream
30         (punit, offset, VC) ← read(R, out)
31         if offset > PV^out[out] then
32             ack ← send(out, punit)
33             if ack then
34                 PV^out[out] ← PV^out[out] + |tuples|
35                 if o = mostDownstream then
36                     marker ← (release, VC),  broadcast(U, marker)
37 upon recovery
38     VC = loadMetadata() ⊕ requestMetadata(D), broadcast(U, VC)
39     PV^in ← VC[U],  PV^out ← VC[D]
40     taskQueue ← taskQueue ∪ recoveryTasks(C)
```

injected with `insertCheckpointMarker` at regular intervals or using a custom dynamic trigger. When an ft-operator receives a checkpoint marker (line 7, Alg. 1): (i) on the first invocation, the `prepareCheckpoint` function triggers the synchronous *prepare phase* for all the p-units of an ft-operator's transient state Θ and its queues I and R; and (ii) once all checkpoint markers are received from its upstream operators, the operator creates and dispatches asynchronous tasks to write the p-units to storage. We decide to accelerate persistence with asynchronous I/O operations because they can overlap with CPU operations (i.e., query execution). The checkpoint completes when all marked p-units have been persisted.

Given a global checkpoint GC_{τ_e} of graph q at τ_e and a snapshot $C^{o_i}_{\tau_{e_i}}$ of operator o_i at τ_{e_i}, the Chandy-Lamport algorithm guarantees that every tuple from GC_{τ_e} is captured either in the upstream operator's queue or the downstream operator's queue or state: $\forall C^{o_1}_{\tau_{e_1}} \in GC_{\tau_e} \forall C^{o_2}_{\tau_{e_2}} \in GC_{\tau_e}, (o_1, o_2) \in S\ \forall \tau_n : (\tau_n \leq \tau_{e_1} \Rightarrow t^{o_1}_n \in R^{o_1}) \wedge (\tau_n > \tau_{e_1} \Rightarrow t^{o_1}_n \in I^{o_2} \vee \tau_{e_2} \geq \tau_n)$. We refer to that as the *at-least-once property*.

Efficient data replay. The persistence protocol replays tuples that are not captured in the last checkpoint by determining what data to persist and remove with *retain* and *release* markers. The *retain* markers are injected into the ingress p-streams at periodic intervals using the POG's `insertRetainMarker` function. They flow through the POG (e.g., Fig. 3 shows a retain marker between operators *Compression* and a_1). Upon receipt of a retain marker (line 16, Alg. 1), the p-stream creates a sequence of p-units with the tuples that follow. To remove parts of the p-stream that are no longer required for the output result, the POG provides *release* markers (line 14), which are sent on feedback channels to discard p-units (Fig. 3 shows a release marker being sent from a_2 to a_1).

Data deduplication is achieved by exploiting the dependencies between p-units to track the query progress and remove duplicates (line 31). Persisting the dependency graphs for all operators before execution, however, introduces a substantial overhead.

Therefore, to achieve exactly-once output, the persistence protocol persists only the most downstream operators' progress with two different methods: the first requires a transactional sink [67, 87], and the most downstream operator performs a two-phase commit to persist progress; the second requires the sink to store a serialized vector clock VC with every output result, return the most recent one on request and filter duplicate tuples similar to line 19 of Alg. 1.

The **recovery protocol** retrieves the query progress from storage before deciding which stream and state parts to restore. Recovery is divided into four phases (lines 37–40, Alg. 1): *progress recovery*, *upstream requesting*, *data recovery* and *upstream replay*.

All operators in progress recovery (line 38) load the timestamp intervals captured by their latest checkpoints and most recent committed dependencies. In upstream requesting (line 38), operators send requests downstream for the latest persisted vector clock. At the end of this phase, every operator has sufficient information to reload the data from the last checkpoint/streams and drop results already processed with its progress vectors PV^{in} and PV^{out}. Next, the protocol moves on to the data recovery phase (line 40) using the `loadState` and `loadStream` functions, while parallelizing the process for effective hardware utilization. In the final upstream replay phase (line 26), operators send data downstream as they would during normal operation. This last phase transitions into regular execution as ingress streams receive new data.

3.3 Garbage collection

In contrast to relational processing, stream queries perform computation over infinite streams, which raises two challenges when persisting data: (i) the finite disk capacity, especially when considering faster non-volatile memory or the storage cost in a cloud infrastructure [64]; and (ii) the high recovery latency when replaying large amounts of data to ensure exactly-once semantics. Thus, it is necessary to remove persisted tuples, state and recovery metadata that is no longer required.

An SPE can use garbage collection (GC) to discard obsolete data with either retention policies [67] or classic mark & sweep. Neither approach, however, applies to a single-node SPE: retention policies require the user to define a threshold for data removal for each query, which cannot be automatically derived from the query semantics; mark & sweep has a high runtime overhead [79]. Therefore, we propose a GC approach that is *semantically partitioned* and *optimistic* under failure.

With semantic partitioning, a single p-stream or ft-operator retains ownership of each p-unit. Given that each operator tracks the dependencies of p-units and manages their ownership when

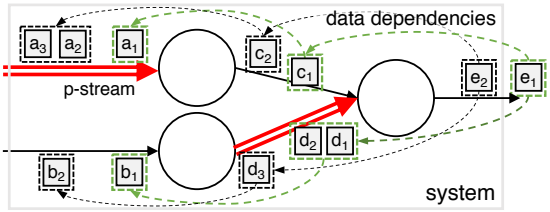


Figure 4: P-unit dependency tracking

passing them downstream, it is easier to reason about correctness when discarding data, and GC is simplified under concurrency.

As p-units in streams are ordered, and the checkpoints capture the progress of ordered data and deterministic operations, GC can be performed at a coarse granularity. This minimizes overhead because previous p-units with an `Offset` less or equal to a given value can be discarded in bulk. Without data loss, a p-unit can be removed if all its dependent results [69] have been either (i) persisted to disk or (ii) committed to the outside world. When these conditions are met, a reverse topological ordered traversal of the dependency graph is performed to send *release* markers upstream (lines 36 and 15). We refer to this GC approach as *optimistic* because it guarantees that all p-units are eventually removed: when p-units with a larger `Offset` are discarded, they invalidate all previous ones.

Example. Fig. 4 shows the emission of tuple e_1 at which point its dependencies (shown in green) can be garbage collected. Note that all transitive dependencies of e_1 appear earlier than the dependencies of e_2, because p-units are ordered by increasing `Offsets` in their operator's partition. By using the `trimFrom` function, each operator removes obsolete stream and state data.

3.4 Persistence push-up

We now describe the optimizations enabled by the POG to reduce disk I/O bandwidth and shorten the recovery process. Stream queries often consist of highly-reductive, inexpensive operators early on in their operator graphs, e.g., `SELECTION`, which eliminates tuples. By considering persistence as an operation within the POG, it can be "pushed up", i.e., executed after the data reducing operators. This provides a compact representation of the stream required for recovery and accelerating persistence.

To perform *persistence push-up*, the POG is traversed in topological order, and a set of transformation rules are applied to rewrite it. These transformation rules identify unused attributes and insert appropriate `PROJECTION` operators to prune them or push down selective operators (e.g., `SELECTION`). Persistence push-up is restricted to the following operator types: operators with selectivity below one (e.g., `SELECTION` or a `HAVING` clause), i.e., ones that output fewer tuples than they consume, or ones that reduce the number of bytes required to represent a tuple (e.g., `PROJECTION` or `COMPRESSION`).

Fig. 3 shows a POG instance after persistence push-up, where the `PROJECTION` and `COMPRESSION` operators are placed to the left of the p-stream, thus decreasing I/O bandwidth for persistence.

The rationale behind the choice of the previous operator types is straightforward: stateful operators (e.g., `AGGREGATION` or `JOIN`) would amplify the output stream size based on the window semantics (e.g., for small window slides), increasing the amount of stored data. In general, stateful operators expose complex data dependencies that are expensive to capture and, thus, are more suitable for checkpointing. Therefore, persistence push-up avoids pushing down stateful operators, as this would increase recovery latency and burden the external sources with buffering data for longer periods before the protocol acknowledges their persistence.

3.5 Correctness

Next, we formally show the correctness of the persistence protocol. The protocol considers the operator graph as a single fail-stop recovery unit [95], i.e., if one or more operators fail, the whole graph must recover. The SPE has access to persistent storage that survives failures, allowing recovery to a different node. It communicates over reliable FIFO network channels with external sources/sinks to guarantee data delivery; the ingress channels[3] allow replay even under failure. To ensure exactly-once output, the protocol requires deterministic operators without side effects, a consistent checkpoint mechanism and that the vector clock VC of the last externally committed tuple is persisted.

First, let us prove that the persistence protocol guarantees exactly-once output for a single operator and then generalize this to arbitrary operator graphs.

Definitions. As discussed in Sec. 2.3, each operator function is modeled as a pair of a state transition function ρ and an output function ω. F denotes the infinite (deterministic and correct) sequence of all tuples produced by an operator without failures. To restrict a tuple sequence to an interval, we use the notation $F[m, n]$ to denote $\langle \omega(\Theta_i, t_i) | i \in [m, n] \rangle$. We denote the deduplication function that uses logical timestamps to filter tuples as ϕ .

Theorem 1. Given a single operator graph, failure at timestamp τ_f and recovery from timestamp τ_r, the persistence protocol produces a recovery sequence $F_r, F_r = \phi\big(F\left[0, \tau_f\right] + \langle \omega\left(\Theta_i, t_i\right) | i \in N, i \geq \tau_r \rangle\big)$ that is equal to the correct sequence, i.e., $F_r = F$.

Proof. Let $C_{\tau_e} = (I, R, \Theta_{\tau_e}, PV^{in}, PV^{out})$ be the checkpoint at timestamp τ_e; let τ_p be the timestamp of the last persisted input tuple in the operator's p-streams such that $\tau_e \leq \tau_p$; let $X[\tau_{p+1}, \infty] = \left\langle t_i | i \in N, i \geq \tau_{p+1} \right]\big\rangle$ be the sequence of tuples held by the external sources (i.e., all tuples after τ_p); and let $VC = \langle \tau_{v_I}, \tau_{v_R} \rangle$ be the last committed vector clock at recovery time (τ_{v_I} and τ_{v_R} being the timestamps of input and output streams, respectively).

The first part of the recovery sequence $F[0, \tau_f]$ denotes the tuples emitted before failure, while $\langle \omega\left(\Theta_i, t_i\right) | i \in N, i \geq \tau_r \rangle$ denotes the sequence produced after failure. For the latter sequence, the operator retrieves its input sequences $I[\tau_{v_I}, \infty]$ and state Θ_{τ_f} in one of three ways: (i) if $\tau_p = 0$ (i.e., no data has been persisted and $\Theta_{\tau_f} = \emptyset$), all data is received from $X[\tau_{p+1}, \infty]$ and the recovery sequence becomes $F[\tau p + 1, \infty] = \left\langle \omega\left(\Theta_i, t_i\right) | i \in N, i \geq \tau_{p+1} \right]\big\rangle$; (ii) if $\tau_{v_R} \leq \tau_e$ (i.e., all output dependent to the checkpoint has been emitted), the operator reconstructs its state Θ_{τ_f} from an empty set by using the state transition function and replays all data persisted in its p-streams until τ_p. The remaining data is received from the sequence $X[\tau_{p+1}, \infty]$ and the operator uses the reconstructed Θ_{τ_f} to produce the sequence $\left\langle \omega\left(\Theta_i, t_i\right) | i \in \left[\tau_{v_I}, \tau_p\right] \right\rangle + F[\tau p + 1, \infty]$; (iii) if $\tau_{v_R} > \tau_e$ (i.e., there is output that depends on the checkpoint), Θ_{τ_f} is restored from C_{τ_e} and data replay from the p-streams and

[3]For non fault-tolerant external sources, the channels are replaced with p-streams.

used to produce the recovery sequence as in case (ii). Thus, in all three cases, F_r can be reconstituted from the last checkpoint and a finite external source buffer (i.e., only data between τ_p and the timestamp at the beginning of recovery).

Given that there may be overlap between the output before and after failure, the duplicate elimination function ϕ ensures at-most-once output. As the concatenated output stream before and after failure guarantees at-least-once, F_r equals F and has the exactly-once property. □

Let us now generalize the property to arbitrary graphs.

Theorem 2. Given an arbitrary execution graph with a single most-downstream operator o_d that is fault-tolerant,[4] a global coherent checkpoint GC, a failure at timestamp τ_f and recovery from timestamp τ_r, the exactly-once guarantee of the final operator extends to the entire operator graph.

Proof. Let us prove the theorem by induction, using Theorem 1 as the base case. The exactly-once fault-tolerance property of the graph is equivalent to the fault-tolerance of the most downstream operator o_d. Analogous to the single operator case, the fault-tolerance of o_d is proven for three cases: just as for Theorem 1, in cases (i) and (ii), the operator replays data from its inputs. By induction, the sequence produced by each input has the exactly-once property, even under failure; in case (iii), the operator loads its state from the last snapshot C_{τ_e} before triggering a downstream replay.

While the at-most-once guarantee stems from the ϕ-function, we must prove at-least-once processing of every input tuple, i.e., that every input tuple is either in its producer's output queue, the operator's input queue, or already reflected in the operator state. Formally, $\forall o_i\ \forall t_j \in F^{o_i}\ \exists\ \tau_x \mid \tau_x < \tau_f : \tau_j > \tau_x \Rightarrow t_j \in R^{o_i} \vee t_j \in I^{o_d} \vee \tau_e \geq \tau_x$. This follows trivially from the at-least-once property of a snapshot, as defined in Sec. 3.2. As o_d guarantees at-most and at-least once results, the operator graph guarantees exactly-once. □

Discussion. The persistence protocol can be extended for out-of-order data processing and non-deterministic operations. With out-of-order data, we can add punctuation tuples [15] to markers for sorting tuples deterministically in a stream.

For the support of non-deterministic operations, the protocol must log all non-deterministic decisions and replay them for recovery [2, 102]: input and output channels must be replaced with p-streams for logging all tuples [34], which may incur a high overhead. New operators can be specified as user-defined functions (UDFs) by implementing the interface from Table 1, while a similar approach to [93] can be used to capture all the sources of non-determinism.

4 SCABBARD ARCHITECTURE

While POGs provide a high-level interface for fault-tolerance operations, an SPE must coordinate these operations efficiently, considering the limited single-node resources and workload characteristics. We describe Scabbard, an SPE for multi-core CPUs that realizes the POG model. Its goal is to provide exactly-once fault-tolerance with minimal performance impact by making workload-aware decisions during execution. After an overview of the Scabbard architecture (Sec. 4.1), we explain how Scabbard manages persistent data, and reduces the recovery time (Sec. 4.2).

[4] Decomposing a query with multiple outputs into multiple queries with a single output is straightforward.

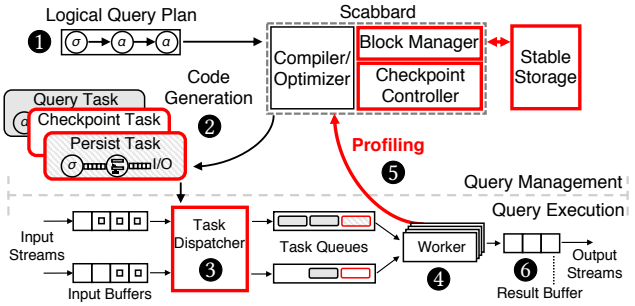


Figure 5: Scabbard architecture (For simplicity, the figure omits the interaction of the query execution layer with the Block Manager and the Checkpoint Controller.)

4.1 Overview

Scabbard is based on the query execution engine and compiler from LightSaber [96]. To support persistence, Scabbard introduces: (i) a *Block Manager* that stores streams and state; and (ii) a *Checkpoint Controller* that orchestrates consistent checkpoints and recovery. For efficient persistence, Scabbard uses task-based parallelization for multi-core execution and adaptive data pruning for I/O bandwidth reduction. Scabbard schedules tasks to a set of worker threads, with each worker bound to a physical CPU core. Depending on the number of pipeline breakers [109] (e.g., `AGGREGATION`), it instantiates one task dispatcher for each pipeline fragment when creating computational tasks.

Fig. 5 shows Scabbard's architecture with a single operator pipeline, highlighting in red the features for workload-aware persistence. Next, we describe the different query execution stages, from the logical plan input to the generation of in-order results:

In stage ❶, a user provides a stream query that is transformed into a logical plan. This plan is optimized in ❷ with rule-based optimizations including (i) operator reordering (i.e., persistence push-up) and (ii) operator fusion. Scabbard uses the optimized plan to generate code for *persistence*, *checkpoint*, and *query* tasks.

The task creation stage ❸ follows after code generation. As data and markers arrive in the input queues of a query pipeline [80] through network sockets or RDMA, different tasks with their data dependencies are created and placed in system-wide queues. When the task queues contain tasks, the workers execute them in ❹.

To provide an up-to-date view of data characteristics (e.g., value distributions), the workers profile a subset of tuples before persistence in ❺. The profiling information may trigger another code generation process for workload-aware data reduction in step ❷ (see Sec. 5.2). Finally, the execution of a query task produces results in immutable batches, which are reordered and assembled in ❻ using the assembly function ρ^{α}.

4.2 Managing fault-tolerance operations

We now explain the role of Scabbard's components, its data storage format, and how it accelerates persistence and recovery.

The **Block Manager** manages the persistent data of a query. When a p-stream or the Checkpoint Controller issue read/write requests to stable storage, they invoke the Block Manager, which returns a valid file pointer for these operations. The Block Manager maintains a pool of files, uniquely identified by a *FileId*, to reduce the overhead

Table 2: Compression algorithms

Name	Description
Base-delta [84]	Represents values as differences (deltas) from a base value
Delta-of-delta [85]	Delta-encoding over the delta-encoded data
Null suppression (NS) [1, 5, 71, 92]	Omits leading zeros from the bit representation
Simple-8b [5]	Stores integers in fixed-size blocks, first bits denote minimum values' bit-length
Variable byte (Var-Byte) [30]	Represents integers as variable number of bytes, using 1 status and 7 data bits per byte
Run-length encoding (RLE) [92]	Represents repeated sequences as pairs of values & counts
XOR compression [85]	Uses XOR'ed floating-point values
Dictionary [1, 92]	Data-agnostic compression scheme that replaces each value with a unique key from a dictionary
Snappy [112]	Dictionary encoding according to LZ77 [112]

of OS file allocation. For p-streams, the Block Manager maintains a circular list of files that maps directly to stream offsets because data is stored in offset order. The Block Manager also tracks which files must be garbage collected and returned to the pool.

The **Checkpoint Controller** coordinates persistence and recovery operations (see Sec. 3.2). During normal execution, it injects markers to trigger the persistence of p-units and creates asynchronous tasks in stage ❸ for parallel execution. Task completion is monitored using a lock-free queue with atomics per pipeline to minimize overhead. When the Controller triggers a checkpoint, the operators withhold their outputs until checkpoint completion to ensure consistency. Regular processing is not disrupted, as the immutable p-units support persistence without an application-level copy-on-write operation.

Storage format. Our goal is to perform parallel non-sequential disk operations without conflicts. We partition each file into smaller logical segments (aligned 256 KB blocks), accelerating reads/writes at the expense of storage space [12].

Serialization costs are reduced by using state management primitives (e.g., vectors or hashtables) that contain tuples with primitive data types (e.g., integers) based on a fixed predefined schema. These primitive types do not require deserialization from storage without compression. For the retrieval of compressed data, however, metadata must be stored at the start of each segment: (i) the offsets of data; (ii) its representation (e.g., data types, row/column format); and (iii) the used compression algorithms, which may change dynamically. For example, for dictionary encoding [1, 92], the hashtable's file offset (implemented with open addressing) and the metadata (e.g., schema) are stored to deserialize it. For windowed operators, the number and sizes of window fragments [65, 96] (e.g., open windows) must be stored for state reconstruction.

I/O optimizations. Scabbard supports NUMA-aware persistence: the task placement respects the affinity of *p-units* to reduce cross-socket communication. It also uses software prefetching of data from remote NUMA nodes, which leads up to 35% better performance for memory-bound queries. To saturate the I/O bandwidth of SSDs and minimize latency, it uses Linux' non-blocking API with asynchronous notifications [62]. All files are opened using the `O_DIRECT` flag to bypass the kernel's page cache and reduce the CPU overhead when performing I/O operations. Workers bulk up writes into chunks to decrease fragmentation and the number of entries in the disk's device queue.

Reducing recovery time. Fast recovery necessitates frequent checkpoints, short initialization times and fast data loading from storage. Scabbard reduces the checkpointing impact by performing them asynchronously. It also partitions streams and state into p-units to enable parallel persistence and recovery (see Sec. 6.4 and Sec. 6.5). During recovery, Scabbard avoids costly code generation by recovering previously-persisted compiled operators: it compiles queries using the LLVM compiler [70] and stores the binaries on disk. Upon restart, it loads the compiled operators, which reduces the restart time by an order of magnitude. Finally, dependency tracking allows Scabbard to load only required p-units.

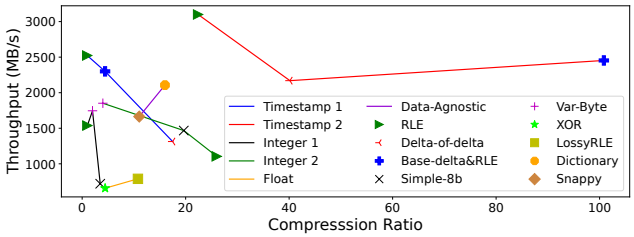


Figure 6: Compression for various data types and distributions

5 WORKLOAD-AWARE STREAM COMPRESSION

Since stream queries are long running, it is beneficial to react to changing workload characteristics at runtime [14]. Scabbard, therefore, reduces the required I/O bandwidth for p-stream persistence using *adaptive compression*. It considers dynamic workload characteristics by monitoring p-streams and generating suitable compression operators. However, the best choice of a compression algorithm exposes a trade-off between compression ratio and throughput and depends on stream and query characteristics [31].

Prior work in stream processing [83] shows that heavyweight schemes [51, 104] with high compression ratios are prohibitively expensive. Therefore, we consider lightweight techniques [1] that combine high-performance with resource efficiency, which we summarize in Table 2. Next, we describe how the supported compression techniques apply to different data types (Sec. 5.1), and how Scabbard chooses between them using an adaptive mechanism (Sec. 5.2).

5.1 Exploiting workload characteristics

We base the decision which compression algorithm to use at runtime on three factors: (i) stream data distribution; (ii) compression ratio; and (iii) compression throughput. Fig. 6 shows the associated performance trade-offs by plotting the compression ratio and throughput for different compression algorithms. Each line represents an input data type, and the marker location indicates the performance (in terms of throughput) for different algorithms. For example, the red line refers to a stream of timestamps; the markers show the compression ratio and throughput for each algorithm applicable to timestamp data.

Timestamps. While timestamps are not a workload-specific data type, we consider them separately because they have discrete non-negative integer values with a relative order. In Fig. 6, we explore RLE, Delta-of-delta, and Base-delta algorithms for two different distributions. If timestamps occur in fixed intervals (Timestamp 1), Delta-of-delta exhibits the best compression ratio and, thus, is used as the default. If multiple events occur within the same interval (Timestamp 2), Base-delta & RLE offers better performance.

Integers. We apply three compression schemes to integer types: Var-Byte, RLE with word-aligned NS (NS & RLE), and Simple-8b. With random not repeated values (Integer 1), Simple-8b achieves the best compression ratio and is thus used by default. If there are multiple runs of values though (Integer 2), NS & RLE yields better results. Var-Byte has the highest throughput and is suitable for lower compression ratios when there is sufficient disk I/O bandwidth.

Floating-points. The nature of floating-point values makes them more challenging to compress efficiently with low overhead. XOR compression offers a good trade-off here. If full precision is unnecessary, the user can set the decimal point precision to a fixed error bound to improve compression. This converts floating-point values into integers, allowing for integer compression schemes. In Fig. 6, we use a floating-point stream with a predefined error bound, thus showing the performance difference with lossy compression.

Data-agnostic. For other data types in streams (e.g., fix-length strings), we observe, based on our evaluation, that dictionary compression works well, especially for a limited set of repeating values. When no statistics are available, Scabbard uses Snappy as the default compression scheme. When it is possible to infer that the data can be mapped to a limited range of distinct values, Scabbard uses a static hashtable, shown as Dictionary in Fig. 6.

As shown in Fig. 6, while some algorithms achieve the highest compression ratio, they have low throughput. The decision of the appropriate algorithm becomes even more complicated when considering that algorithms, such as lossy compression for floating-points or dictionary encoding, produce new data types that can be further compressed with other approaches. Scabbard, therefore, chooses the compression algorithms adaptively.

5.2 Adaptive stream compression

Scabbard adds lightweight instrumentation code to each pipeline fragment to carry out fine-grained profiling. For a pipeline fragment and input column in a p-stream, information is collected about the value distribution (e.g., the min/max value) and characteristics specific to the compression schemes (e.g., the average run-length of consecutive equal values).

Scabbard analyzes the statistics periodically at a configurable interval[5] and combines them with static information, e.g., the p-stream schema, and heuristics about the algorithms [1]. This allows Scabbard to reason about the data characteristics (e.g., data range) and insert newly generated compression operators into the POG.

At the beginning of query execution, Scabbard starts with a predefined compression scheme per column for each pipeline fragment. Upon detecting workload changes for a pipeline fragment, Scabbard JIT-compiles new compression/decompression operators and fuses them with the query-specific pruning operators from Sec. 3. The generated operators are memoized and maintained as function pointers, inserted dynamically into the operator graph.

With the above approach, however, different compression operators may execute simultaneously. Thus, workers store the metadata for each approach (see Table 4.2) and use the generated decompression functions on the compressed data. If the p-stream characteristics change, e.g., a column's bit precision changes, a worker may decide for deoptimization [39, 47, 50] by falling back to the default compression scheme to ensure correct results.

This adaptive approach supports a wide range of optimizations, such as selecting the most resource-efficient algorithm or specializing the underlying data structures (e.g., the hashtable for dictionary encoding). An example of such optimizations is using bit precision information for integers to replace the more expensive Simple-8b algorithm with word-aligned NS; another example is the use of average run-length statistics to decide whether to use RLE.

[5]It is statically defined, but it could change dynamically based on the collected statistics.

Table 3: Evaluation datasets and workloads

Datasets		Queries		
Name	# Attr. / Size (B)	Name	Windows (s)	Operators
Cluster Monitoring (CM) [25, 61]	12 / 64	CM_1	$\omega_{60,1}$	$\pi, \gamma, \alpha_{\text{sum}}$
		CM_2	$\omega_{60,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$
Smart Grid (SG) [57]	7 / 32	SG_1	$\omega_{3600,1}$	π, α_{avg}
		SG_2	$\omega_{128,1}$	$\pi, \gamma, \alpha_{\text{avg}}$
		SG_3	$\omega_{1,1}, \omega_{1,1}$	$\pi, \sigma, \bowtie$
Linear Road Benchmark (LRB) [9]	7 / 32	LRB_1	$\omega_{300,1}$	$\pi, \sigma, \gamma, \alpha_{\text{avg}}$
		LRB_2	$\omega_{30,1}$	$\pi, \gamma, \alpha_{\text{count}}$
		LRB_3	$\omega_{30,1}, \omega_{1,1}$	$\pi, \gamma, \alpha_{\text{count}}$
Yahoo Streaming (YSB) [26]	7 / 128	YSB	$\omega_{10,10}$	$\sigma, \pi, \bowtie_{\text{relation}}, \gamma, \alpha_{\text{count}}$
NEXMark (NQ) [100]	9 / 128	NQ	$\omega_{60,1}$	$\pi, \gamma, \alpha_{\text{count}}, \alpha_{\text{max}}, \bowtie$
Sensor Monitoring (SM) [56]	14 / 64	SM	$\omega_{60,1}$	π, α_{avg}

6 EVALUATION

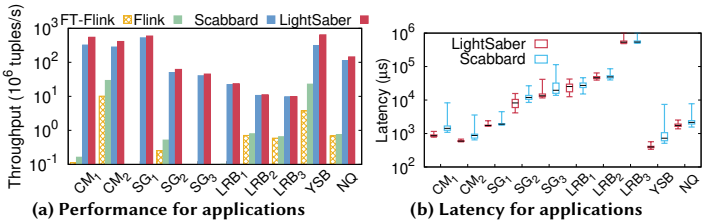
We evaluate Scabbard to explore the benefits of its design in a top-down fashion: we start by comparing Scabbard with state-of-the-art SPEs in terms of throughput and latency under a range of real-world query benchmarks (Sec. 6.2). We then investigate the efficiency of stream persistence (Sec. 6.3), checkpointing (Sec. 6.4), recovery (Sec. 6.5), persistence push-up, compression (Sec. 6.6), and execution with remote sources, sinks and storage (Sec. 6.7).

6.1 Experimental setup

We run experiments on three servers: Server A with two Intel Xeon E5-2640 v3 2.60 GHz CPUs (16 physical cores), a 20 MB LLC cache, 64 GB of memory, and a local 256 GB SSD (950 MB/s write bandwidth; 72k IOPS); a c5.4xlarge AWS EC2 instance (Server B) with EBS [3] for remote storage (700 MB/s write bandwidth; 16k IOPS); Server C with four Intel Xeon E5-4660 v4 2.20 GHz (64 physical cores), a 40 MB LLC cache, 528 GB of memory, and a local 1.6 TB SSD (1.5 GB/s write bandwidth; 90k IOPS). We use Ubuntu 18.04 and Clang 9.0.0 with `-O3 -march=native`. Unless stated otherwise, all experiments are executed on Server A using all cores.

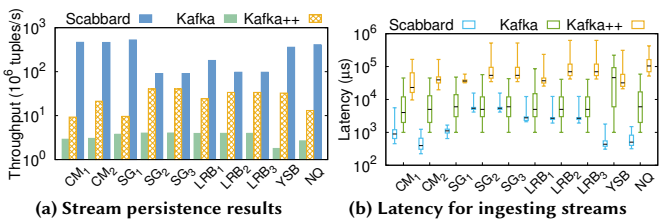
Stream persistence systems. We compare to (i) Apache Kafka v2.3.0 [7], a persistent messaging system; and (ii) a C++ prototype (Kafka++) that flushes data to disk before acknowledging it.

For a fair comparison, we tune Kafka for high throughput by batching input messages; using multiple partitions and producers per topic; and maintaining a median latency of less than 90 ms, which we deem acceptable given the latency results measured below. We use `acks= "all"` mode to persist tuples to disk before acknowledging them, and `replication.factor="1"`. We find that, in most cases, the compression algorithms supported by Kafka lead to performance degradation or increased latency; thus, we disable this feature. For the prototype, we manage memory and execution



(a) Performance for applications (b) Latency for applications

Figure 7: Application benchmark queries



(a) Stream persistence results (b) Latency for ingesting streams

Figure 8: Ingestion of streams

as in Scabbard, pre-partition the input to avoid the additional cost and use Snappy [112] compression.

Stream processing engines. We compare to (i) Apache Flink v1.12.0 [6], a Java-based scale-out SPE; (ii) a hardcoded C++ implementation of Flink's execution strategy (Flink++); and (iii) LightSaber [96], a single-node SPE without fault-tolerance.

Following best practices [27] for data ingestion, we configure Kafka to use as many partitions as Flink workers. We enable object reuse and preload the input data into Kafka partitions before starting experiments to avoid bottlenecks. For Flink++, we pre-partition the input, perform operator fusion, manage memory as in Scabbard, and use Kafka++ as its persistent source (Flink-Kafka++).

We examine Scabbard with and without stream persistence (Scabbard-Chk). If not stated otherwise, we checkpoint all operators every second and generate in-memory ingress streams for the remaining systems. We pre-populate large buffers and replay tuples by updating their timestamps to avoid network bottlenecks.

Workloads. We use the macro-benchmark stream queries from previous work [96] as well as four additional queries: (1) the first workload emulates two cluster monitoring applications (CM) [103] that apply a grouped aggregation over a sliding window; (2) the smart grid queries (SG) [57] perform anomaly detection: SG_1 calculates a sliding global average of a meter load, SG_2 reports the sliding load average per plug in a household, and SG_3 joins their results with a tumbling window; (3) the Linear Road Benchmark (LRB) [9] computes three queries on a network of toll roads with multiple key groupings: LRB_1 performs a grouped window aggregation with a selection to find congested road segments; LRB_2 and LRB_3 (a tumbling window count over LRB_2) count the number of vehicles in road segments; (4) the Yahoo Streaming Benchmark (YSB) [26] emulates an advertisement application with a table join and a windowed count using numerical values (128 bits) [86]; (5) the fifth query (NQ) from NEXMark benchmark [100] that monitors auction items with the most bids over a sliding window; (6) finally, the sensor monitoring (SM) [56] query computes the running average of three energy readings. Table 3 summarizes the workloads, with the window sizes and slides measured in seconds.[6]

Metrics. Following prior work [101], we define end-to-end processing latency as the difference between the time when a tuple enters the system and when a window result is produced. Candlesticks in plots show the 5th, 25th, 50th, 75th and 95th percentiles, respectively.

[6] All window sizes and slides are defined using event time to be independent of processing latency.

6.2 System comparison

Fig. 7a compares the performance of Flink with 1-sec checkpoints (denoted as Flink-FT) with that of Flink without fault-tolerance, LightSaber (no fault-tolerance), and Scabbard. The results show that for compute-intensive queries (SG_3, $LRB_{1\text{-}3}$), Scabbard exhibits less than 11% performance drop over LightSaber, and effectively hides the cost of persistence. For the remaining memory-intensive queries, we observe that the overhead of persistence leads to a greater degradation: 69% for CM_1, 45% for CM_2, 12% for SG_1, 23% for SG_2, up to 2× for YSB, and 28% for NQ, respectively.

Compared to Flink-FT, Scabbard performs at least an order of magnitude better for all queries, even though it performs additional work for stream persistence. To investigate the fault-tolerance overhead, we use the bpftrace tools [45] and measure the average block I/O device latency for disk operations. While Flink has an average latency of 16 ms with frequent spikes (up to 64 ms), Scabbard exhibits low and predictable (around 64 μs) average latency with 1 ms spikes by bypassing the kernel's page cache. The average disk latency explains the increased number of memory stalls for Flink that lead to a 4–6× performance overhead for $LRB_{2\text{-}3}$ and YSB.

Next, we compare the end-to-end latency of Scabbard against LightSaber (we omit the results for Flink, as they are an order of magnitude worse [46, 101]). Fig. 7b shows that, similar to LightSaber, Scabbard exhibits median latency lower than 50 ms for all queries, except for LRB_3, in which both systems have sub-second latency. For the compute-intensive queries (SG_3 and $LRB_{1\text{-}3}$), the increase in latency is shown mostly in the 95th percentile, while for the rest, we observe that the median latency is more than 2× higher.

The experiments show that Scabbard achieves at least an order of magnitude higher throughput compared to state-of-the-art fault-tolerant SPEs, with only an up to 10× increase in the 95th percentile latency. Having established Scabbard's high-level performance profile, we study the factors contributing to its performance.

6.3 Stream persistence cost

Next, we compare Scabbard's stream persistence to that of Kafka to reveal the overhead of existing approaches. Fig. 8a shows that Kafka achieves comparable performance for all applications (up to 4m tuples/s). However, Scabbard has at least two orders of magnitude greater throughput for all benchmarks. When analyzing resource utilization, we observe that Kafka introduces more instruction cache misses (the JVM leads to a large code footprint) and memory cache misses (caused by serialization, copying, and object allocation), which prevent scaling even with compression.

Next, we remove the aforementioned bottlenecks with Kafka++, which achieves up to an order of magnitude higher throughput and performs almost the same as Scabbard for queries $SG_{2\text{-}3}$ and

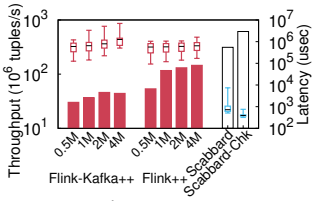


Figure 9: YSB

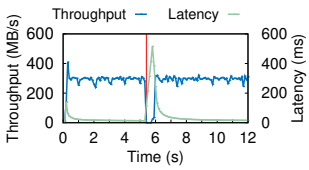


Figure 10: Performance with failure

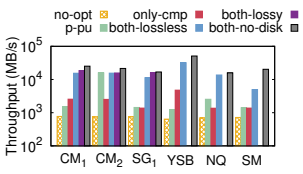


Figure 11: Data reduction

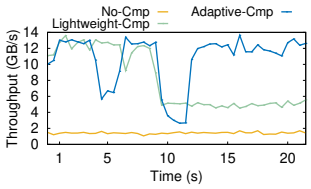


Figure 12: Adaptive compression

Table 4: Checkpointing based on application characteristics

App	State (MB)		Avg checkpoint time (ms)		Overhead	
	Scabbard	*Flink*	*Scabbard*	*Flink*	*Scabbard*	*Flink*
CM_1	18	0.08	25.7	292	4%	5%
CM_2	10	0.08	44.3	275	9%	3%
SG_1	2	0.03	89.6	291	1%	1%
SG_2	41	0.08	68.6	290	7%	11%
SG_3	115	3.3	103.6	> 60000	2%	17%
LRB_1	114	4.2	122.7	9000	5%	15%
LRB_2	105	5.9	168.6	1000	14%	20%
LRB_3	143	6	361	2000	1%	10%
YSB	23	0.13	23.1	311	6%	1%
NQ	27	2.68	49.1	932	4%	10%

$LRB_{2\text{-}3}$, at the expense of a 7× latency increase, as shown in Fig. 8b. This increase is caused by the large batch size required to achieve high throughput with synchronous disk writes.

In addition, we observe a significant percentage of stalls and high I/O device latency for both implementations that perform synchronous flushes to the page cache. Scabbard, in contrast, has more efficient resource utilization (e.g., NUMA locality), writes fewer bytes to disk per tuple, reduces the transmission overhead with compression and block-aligned writes, and submits more asynchronous I/O requests per second to disk.

6.4 Checkpointing overhead

In Table 4, we measure the performance overhead of checkpointing with a 1-sec interval for Scabbard without p-streams (Scabbard-Chk) and Flink: in terms of the average checkpoint size, the checkpointing time, and the performance overhead.

With $LRB_{1\text{-}3}$, the checkpoint time is affected by the persisted state size. Compared to Flink, Scabbard has higher throughput for all queries, which leads to larger state sizes. When the state grows to several MBs, checkpointing affects performance adversely over time. Thus, for queries SG_3 and $LRB_{1\text{-}3}$, we had to increase Flink's checkpoint interval. Overall, Scabbard combines efficient parallelization of persistence with data reduction and predictable I/O latency, allowing frequent snapshots and low recovery time.

We also consider the efficiency of unaligned checkpoints (i.e., persisting streams along with state) in a single-node SPE using Flink++ and YSB: we choose a workload with tumbling windows that allows a comparison without aggregation optimizations [96].

Fig. 9 compares Flink++ for different batch sizes with and without stream persistence (using Kafka++ from Sec. 6.3) to Scabbard and Scabbard-Chk. With only checkpointing, the prototype exhibits 5× worse performance, two orders of magnitude higher latency, and 6× greater checkpoint time with a batch size >1 MB. This latency increase is due to message passing [109] and the alignment phase required during Flink's shuffle stage. While Flink++ waits for the checkpoint completion before committing results, Scabbard uses its dependency tracking mechanism to output the results immediately. However, Flink++ stores to disk 100× less data with aligned checkpoints, demonstrating how the additional I/O pressure can become a bottleneck for Scabbard without the persistence optimizations. With stream persistence enabled, Scabbard interleaves persistence with normal execution and yields 7× higher throughput.

6.5 Recovery with remote storage

In this experiment, we evaluate Scabbard's behaviour during recovery with remote storage. We use an AWS EC2 instance with Elastic Block Storage (EBS), excluding the time for failure detection and machine restart. We use LRB_1 (the other queries exhibit similar behaviour) and persist the ingress stream and checkpoint every second and configure the generator to generate the stream at 300 MB/s (i.e., half the maximum sustainable throughput) to allow Scabbard to catch up with the input when recovering.

Fig. 10 shows the throughput before and after manually triggering a failure by terminating the Scabbard process (indicated by the red vertical line). Upon failure, it initializes (memory pre-allocation and precompiled code loading), which takes approximately 360 ms, followed by 100 ms of recovery time. In terms of average latency, there is an initial increase while Scabbard is down and restarts, but it then recovers to the pre-failure latency within 2 s.

To emulate a failure in Flink, we stop and restart the worker process (TaskManager) and collect the logged events. The restart time of the TaskManager is 38 s, and recovering the state from disk takes 2 s. This is the effective recovery time expected from a hot-standby system and, thus, the key metric of this experiment. Using this metric, Scabbard performs roughly 20× better than Flink.

6.6 Optimization breakdown

We study Scabbard's data reduction techniques. In the first experiment, we execute the queries bound by the disk bandwidth ($CM_{1\text{-}2}$, SG_1, YSB, NQ, SM) and evaluate Scabbard using six configurations: (i) no compression and no persistence push-up (no-opt); (ii) only persistence push-up (p-pu); (iii) only compression (only-cmp); (iv) both p-pu and lossless floating-point compression (both-lossless); (v) both p-pu and two-decimal digit precision floating-point compression if applicable (both-lossy); and (vi) both optimizations without persistence to emulate a fast storage medium (both-no-disk). The last configuration assumes that disk bandwidth is no longer a bottleneck in future hardware architectures.

Fig. 11 shows that Scabbard without data reduction reaches up to 780 MB/s, which matches the disk bandwidth. For all queries apart from CM_2 (low filter selectivity), using only one of the techniques does not yield the optimal throughput. With both optimizations, Scabbard outperforms the baseline (no-opt) from 7× to 50×, depending on the input data characteristics. For CM_1 and SG_1, the lossy compression yields 20–40% performance improvement.

We conclude that Scabbard benefits from disks with higher bandwidth and lower latency operations, as we observe a 1.2–4×

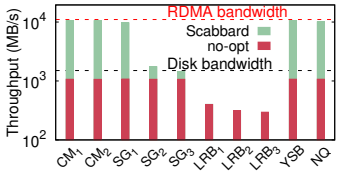


Figure 13: Remote ingestion

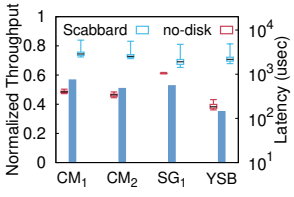


Figure 14: Remote storage

speedup. With a faster disk, it may become necessary to sacrifice the compression ratio for throughput. As future work, we want to develop a cost-based model to resolve this.

In the second experiment, we explore the performance benefit of adaptive compression when the data characteristics change over time. We execute the SM query, and after 10 secs, we change the value distribution of the integer columns.

Fig. 12 compares Scabbard with adaptive compression against the default compression approach and without compression. While Scabbard profiles execution and decides to switch the compression function every 4 secs, this does not affect performance. Based on the collected statistics, Scabbard generates a more efficient compression scheme, resulting in a 5–10% performance improvement. After 10 secs, the data characteristics change, invalidating the assumptions of the generated code, and Scabbard falls back to the generic compression algorithms. Finally, after 12 secs, it uses the most recent statistics to generate a new compression function, yielding a 2× improvement. The reoptimization interval can be reduced to adapt more quickly at the cost of higher overhead. We conclude that Scabbard adapts effectively to changing workload characteristics at runtime, resulting in up to a 2× performance gain.

6.7 Remote I/O bottlenecks

We consider the impact of the network that interconnects Scabbard with remote sources/sinks and storage. We observe its behaviour when ingesting data over the network with and without data reduction (no-opt). To have sufficient bandwidth, we connect Server C (see Sec. 6.1) using RDMA over 100 Gb/s with two separate machines (similar to Server A) to generate streams and commit results.

In Fig. 13, for the memory-intensive queries, Scabbard manages to saturate the RDMA bandwidth with less than 6 physical cores, which shows the importance of Scabbard's data reduction techniques when the ingestion rate is higher than the disk bandwidth. The performance improvement for $SG_{2\text{-}3}$ is up to 65%, and for the remaining queries, data reduction does not improve performance and increases latency. This experiment reveals that data reduction plays a crucial role with fast networks.

In Fig. 14, we compare Scabbard's performance with and without (no-disk) remote block storage in terms of throughput and latency using the EC2 instance (Server B). For queries $CM_{1\text{-}2}$, SG_1, and YSB, Scabbard exhibits greater throughput degradation compared to the local disk experiments (Sec. 6.2) because it saturates the IOPS of the EBS volume. Thus, we increase the batch size to reduce IOPS, which results in up to 12× higher 75^{th} percentile latency. The remaining queries exhibit similar performance to local storage with less than a 2× latency increase. We conclude that high-speed networking allows for remote storage with low overhead.

7 RELATED WORK

Fault-tolerance in SPEs. Many industrial [68, 81, 98] and academic [65, 77, 96, 110] SPEs only achieve high throughput and low latency with limited fault-tolerance. Compared to systems with partial fault-tolerance [52, 54] that sacrifice the precision of recovered results, Scabbard offers stronger processing guarantees.

More recent scale-out systems [2, 18, 18, 37, 55, 99] use checkpointing for fault-tolerance. SEEP uses continuous state checkpointing and input replay for recovery, which shares similarities with our work, but it does not efficiently manage the shared-memory state persistence. IBM Streams, Apache Flink, and Naiad employ a variation of the Chandy-Lamport algorithm [22] but are not designed for persisting streams efficiently. Instead, these systems rely on messaging systems [4, 7, 87] and general-purpose stores [21, 35]. In contrast, Scabbard integrates persistence with the operator graph to enable workload-aware optimizations.

Another common approach is the use of a lineage-based mechanism [10, 72, 88] that persists all data dependencies, which would compromise performance for scale-up designs. Data migration (e.g., Rhino [78] or Megaphone [49]) is an orthogonal technique that uses fast remote storage to speed up recovery to a new machine, and it also enables query reconfiguration at runtime.

Adaptive optimizations in SPEs have been used extensively in SPEs [13, 24, 41, 47, 65, 111]. Early research focused on plan migration [24, 41, 111] in distributed deployments or operator reordering [13]. SABER [65] uses an online algorithm to choose between CPU and GPU execution of operators. Grizzly [47] employs adaptive optimizations with query compilation to accelerate execution. These approaches are orthogonal to our work, which uses adaptative compression to reduce I/O bandwidth.

Compression in SPEs. Gorilla [85] is a time series database that introduces compression schemes for stream timestamps and floats. Scabbard utilizes these techniques and provides a more general solution for stream data through adaptive optimization [14]. Tersecades [83] uses hardware accelerators for compression and performs some computation directly over compressed data; Scabbard uses compression to accelerate persistence.

8 CONCLUSION

To enable fault-tolerance with exactly-once semantics in a single-node SPE without compromising performance, we developed Scabbard. It tightly couples the persistence operations with the operator graph through a novel *persistent operator graph* model and dynamically reduces the required disk bandwidth at runtime. Scabbard achieves sub-second recovery latencies by performing frequent checkpointing and optimistic garbage collection. Consequently, it outperforms the state-of-the-art fault-tolerant SPEs by at least an order of magnitude on all our benchmarks, processing hundreds of millions of tuples/sec with millisecond latencies.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS; EP/L016796/1).

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *ACM SIGMOD*. 671–682.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Is, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. In *Proc. VLDB Endow.*, Vol. 6. 1033–1044.

[3] Amazon. 2021. Amazon Elastic Block Store. https://aws.amazon.com/ebs/. Last access: 28/10/21.

[4] Amazon. 2021. Amazon Kinesis. https://aws.amazon.com/kinesis/data-streams/. Last access: 28/10/21.

[5] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Software: Practice and Experience* 40, 2 (2010), 131–147.

[6] Apache Flink. 2021. https://flink.apache.org. Last access: 28/10/21.

[7] Apache Kafka. 2021. https://kafka.apache.org/. Last access: 28/10/21.

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006), 121–142.

[9] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proc. VLDB Endow.*, Vol. 30. 480–491.

[10] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *ACM SIGMOD*. 601–613.

[11] InfiniBand Trade Association. 2021. InfiniBand Roadmap - Advancing InfiniBand. https://www.infinibandta.org/infiniband-roadmap/. Last access: 28/10/21.

[12] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture.. In *EDBT*. 461–466.

[13] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *ACM SIGMOD*. 261–272.

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

[15] Magdalena Balazinska. 2005. *Fault-tolerance and load management in a distributed stream processing system.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[16] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *ACM SIGMOD*. 1463–1475.

[17] Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest? 2020. https://www.confluent.io/blog/kafka-fastest-messaging-system/. Last access: 28/10/21.

[18] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proc. VLDB Endow.* 10, 12, 1718–1729.

[19] 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 TCDE* 36, 4 (2015).

[20] 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. In *Proc. VLDB Endow.*, Vol. 8. 401–412.

[21] 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 *ACM SIGMOD*. 275–290.

[22] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM TOCS* (1985), 63–75.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM TOCS* (2008), 1–26.

[24] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *ACM SIGMOD Rec.* (2000), 379–390.

[25] Xin Chen, Charng-Da Lu, and K. Pattabiraman. 2014. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *ISSRE*. 167–177.

[26] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *IEEE IPDPSW*. 1789–1792.

[27] Confluent. 2020. Optimizing Your Apache Kafka Deployment. https://www.confluent.io/thank-you/white-paper/optimizing-your-apache-kafka-deployment/. Last access: 28/10/21.

[28] OpenFog Consortium. 2017. Smart cities scenario. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. Last access: 28/10/21.

[29] OpenFog Consortium. 2017. Transportation scenario: Smart cars and traffic control. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. Last access: 28/10/21.

[30] Doug Cutting and Jan Pedersen. 1989. Optimization for Dynamic Inverted Index Maintenance. In *ACM SIGIR*. 405–411.

[31] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM TODS* (2019), 1–46.

[32] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In *ACM DEBS*. 227–238.

[33] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*.

[34] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM CSUR* (2002), 375–408.

[35] Facebook. 2012. RocksDB. http://rocksdb.org/. Last access: 28/10/21.

[36] feedzai.com. 2013. Modern Payment Fraud Prevention at Big Data Scale. http://tinyurl.com/nwnzdxs.

[37] 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 *ACM SIGMOD*. 725–736.

[38] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*. 49–60.

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

[40] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. A Survey on the Evolution of Stream Processing Systems. *arXiv preprint arXiv:2008.00842* (2020).

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

[42] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *ACM SIGCOMM*. 350–361.

[43] Google. 2021. Google compute engine persistent disk. https://cloud.google.com/persistent-disk. Last access: 28/10/21.

[44] Thore Graepel, Joaquin Quiñonero Candela, Thomas Borchert, and Ralf Herbrich. 2010. Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *ICML*. 13–20.

[45] Brendan Gregg. 2019. *BPF Performance Tools.* Addison-Wesley Professional.

[46] Jamie Grier. 2016. Extending the Yahoo! Streaming Benchmark. https://www.ververica.com/blog/extending-the-yahoo-streaming-benchmark. Last access: 28/10/21.

[47] Philipp M Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient stream processing through adaptive query compilation. In *ACM SIGMOD*. 2487–2503.

[48] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network support for resource disaggregation in next-generation datacenters. In *ACM HotNets*. 1–7.

[49] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. In *Proc. VLDB Endow.*, Vol. 12. 1002–1015.

[50] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *ACM SIGPLAN*. 32–43.

[51] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *IRE* (1952), 1098–1101.

[52] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. 2005. High-availability algorithms for distributed stream processing. In *ICDE*. 779–790.

[53] Introduction to Kafka Streams. 2017. http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple. Last access: 28/10/21.

[54] Gabriela Jacques-Silva, Bugra Gedik, Henrique Andrade, and Kun-Lung Wu. 2009. Language level checkpointing support for stream processing applications. In *DSN*. 145–154.

[55] Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. 2016. Consistent regions: Guaranteed tuple processing in ibm streams. In *Proc. VLDB Endow.*, Vol. 9. 1341–1352.

[56] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 Grand Challenge. In *ACM DEBS*. 393–398.

[57] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 Grand Challenge. In *ACM DEBS*. 266–269.

[58] Vasiliki Kalavri and John Liagouris. 2020. In support of workload-aware streaming state management. In *USENIX HotStorage*.

[59] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *USENIX ATC*. 437–450.

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

[61] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. An Analysis of Traces from a Production MapReduce Cluster. In *CCGrid*. 94–103.

[62] Kernel Asynchronous I/O for Linux. 2021. http://lse.sourceforge.net/io/aio.html. Last access: 28/10/21.

[63] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *EuroSys*. 1–15.

[64] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *USENIX ATC*. 759–773.

[65] 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 *ACM SIGMOD*. 555–569.

[66] G. Kovacs. 2017. EBS, EFS, or Amazon S3: which is the best cloud storage system for you? https://cloud.netapp.com/blog/ebs-efs-amazons3-best-cloud-storage-system. Last access: 28/10/21.

[67] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *NetDB*. 1–7.

[68] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *ACM SIGMOD*. 239–250.

[69] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[70] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. 75.

[71] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* (2015), 1–29.

[72] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *USENIX NSDI*. 439–453.

[73] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez-Hernández, Bogdan Nicolae, Radu Tudoran, and Stefano Bortoli. 2018. Kera: Scalable data ingestion for stream processing. In *IEEE ICDCS*. 1480–1485.

[74] title = Roadmap to Building a Streaming Database on Timely Dataflow Materialize. 2020. https://materialize.io/blog-roadmap/. Last access: 28/10/21.

[75] Friedemann Mattern et al. 1988. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science.

[76] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. 2019. Streambox-hbm: Stream analytics on high bandwidth hybrid memory. In *ASPLOS*. 167–181.

[77] 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 ATC*. 617–629.

[78] Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *ACM SIGMOD*. 2471–2486.

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

[80] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. In *Proc. VLDB Endow.*, Vol. 4. 539–550.

[81] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed stream computing platform. In *IEEE ICDM*. 170–177.

[82] David A Patterson, Garth Gibson, and Randy H Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD*. 109–116.

[83] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. 2018. TerseCades: Efficient Data Compression in Stream Processing. In *USENIX ATC*. 307–320.

[84] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-Delta-Immediate Compression: Practical Data Compression for on-Chip Caches. In *ACM PACT*. 377–388.

[85] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. In *Proc. VLDB Endow.*, Vol. 8. 1816–1827.

[86] Peter Pietzuch, Panagiotis Garefalakis, Alexandros Koliousis, Holger Pirk, and Georgios Theodorakis. 2018. Do We Need Distributed Stream Processing? https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing. Last access: 28/10/21.

[87] Apache Pulsar. 2016. Open-sourcing Pulsar, Pub-sub Messaging at Scale. https://yahooeng.tumblr.com/post/150078336821/open-sourcing-pulsar-pub-sub-messaging-at-scale. Last access: 28/10/21.

[88] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *ACM EuroSys*. 1–14.

[89] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *ACM SOSP*. 497–514.

[90] Redpanda Raison D'etre. 2019. https://vectorized.io/blog/redpanda-raison-detre/. Last access: 28/10/21.

[91] David Reinsel, John Gantz, and John Rydning. 2018. Data age 2025: The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. Last access: 28/10/21.

[92] Mark A Roth and Scott J Van Horn. 1993. Database compression. *ACM Sigmod Record* (1993), 31–39.

[93] Pedro F Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *ACM SIGMOD*. 1637–1650.

[94] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *ACM Sigmod Record* (2005), 42–47.

[95] Rob Strom and Shaula Yemini. 1985. Optimistic recovery in distributed systems. *ACM TOCS* (1985), 204–226.

[96] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-core Processors. In *ACM SIGMOD*. 2505–2521.

[97] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* (2018), 847–872.

[98] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *ACM SIGMOD*. 147–156.

[99] Trident. 2021. http://storm.apache.org/Trident-tutorial.html. Last access: 28/10/21.

[100] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark—A Benchmark for Queries over Data Streams DRAFT*. Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.

[101] 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 *ACM SOSP*. 374–389.

[102] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage stash: fault tolerance off the critical path. In *ACM SOSP*. 338–352.

[103] John Wilkes. 2011. More Google Cluster Data. Google Research Blog. http://bit.ly/1A38mfR. Last access: 28/10/21.

[104] Ross N Williams. 1991. An extremely fast Ziv-Lempel data compression algorithm. In *IEEE Data Compression Conference*. 362–363.

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

[106] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *ACM International Systems and Storage Conference*. 1–11.

[107] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*. 15–28.

[108] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *ACM SOSP*. 423–438.

[109] 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 *Proc. VLDB Endow.*, Vol. 12. 516–530.

[110] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *ACM SIGMOD*. 705–722.

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

[112] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* (1977), 337–343.