S-QUERY: Opening the Black Box of Internal Stream Processor State

Jim Verheijde$^{\delta}$, Vassilios Karakoidas*, Marios Fragkoulis$^{\delta}$, Asterios Katsifodimos$^{\delta}$
$^{\delta}$Delft University of Technology, *Delivery Hero SE
{j.l.verheijde,m.fragkoulis,a.katsifodimos}@tudelft.nl, bkarak@deliveryhero.com

***Abstract*—Distributed streaming dataflow systems have evolved into scalable and fault-tolerant production-grade systems. Their applicability has departed from the mere analysis of streaming windows and complex-event processing, and now includes cloud applications and machine learning inference. Although the advancements in the state management of streaming systems have contributed significantly to their maturity, the internal state of streaming operators has been so far hidden from external applications. However, that internal state can be seen as a materialized view that can be used for analytics, monitoring, and debugging.**

In this paper we argue that exposing the internal state of streaming systems to outside applications by making it queryable, opens the road for novel use cases. To this end, we introduce S-QUERY: an approach and reference architecture where the state of stream processors can be queried - either live or through snapshots, achieving different isolation levels. We show how this new capability can be implemented in an existing open-source stream processor, and how queryable state can affect the performance of such a system. Our experimental evaluation suggests that the snapshot configuration adds only up to 8ms latency in the 99.99th percentile and negligible increase in 0-90th percentiles.

I. INTRODUCTION

Over the last decade stream processing systems have evolved from experimental engines producing approximate results, to production-ready sophisticated platforms providing consistent execution of long-running jobs on hundreds of nodes, even in face of failures [1]. State management in particular, enabled important advancements in fault tolerance and scalability by partitioning state and enforcing global coordinated checkpoints [2], [3]. Now that streaming systems can reason about their state and keep it consistent, exposing their internal state to applications can pave the way for new capabilities, such as auditing and debugging.

At the same time streaming systems are no longer used just to serve analytics use cases, but they are increasingly preferred for executing new types of workloads such as serving machine learning models [4] and running cloud applications based on microservices and stateful functions [5]–[7]. Especially for operational use cases such as the execution of cloud applications, the ability to query the distributed state of a streaming system in one shot offers a database view of its processing state, similar to what query interfaces offer in traditional database systems. For instance, an e-commerce application running on top of a streaming system would be able to join user accounts with purchases to determine sales grouped by user characteristics, such as age and gender.

The problem of querying the state of distributed streaming systems externally presents important challenges in terms of overhead to the operation of the streaming system, correctness of query results, and scalable management of state sizes. First, streaming systems are susceptible to operational overhead given that they are designed and used for low-latency high-throughput processing. In fact, the processing they perform is continuous and it triggers continuous state updates. Thus, it is important that state access for answering an external query induces minimal overhead. Second, the correctness of query results under continuous processing also poses a challenge especially for queries that combine different parts of a streaming system's distributed state, which typically reside in remote hosts. Capturing a consistent view of the involved parts of the state requires aligning them in some way. The third challenge regards the size of the accumulated state over time, which can become considerably difficult to manage. The state size affects both query performance and the amount of space required for holding the state.

Surprisingly, external queries to the internal state of streaming systems have received almost no attention in the literature. Systems such as TSpoon [8] and Apache Flink [9] are able to query the system's state, but they are limited to key-value lookups over the live state. Thus, they do not expose the state in a way that can support more sophisticated operations such as joins, filters and aggregates of aggregating state values, which would be valuable to external applications.

In this paper we propose S-QUERY, an approach for querying the internal distributed state of a stream processing system. S-QUERY exposes the live state of a streaming system to external systems in a safe manner, thereby providing a fresh view of its live state. In addition, S-QUERY supports queries over the system's snapshot state produced by periodic checkpoints, without obstructing the normal processing of the system. These two querying capabilities are complementary and can be very beneficial to external applications: live state queries offer a realtime view with no correctness guarantees, while snapshot queries produce consistent results over past states but may be slightly outdated.

We implement S-QUERY in Hazelcast Jet [10], a distributed streaming system that optimizes low-latency performance. Our experimental evaluation suggests that the snapshot configuration adds only up to 8ms latency in the 99.99th percentile and negligible increase in 0-90th percentiles. In addition, S-QUERY scales horizontally, allowing for sustainable throughput to

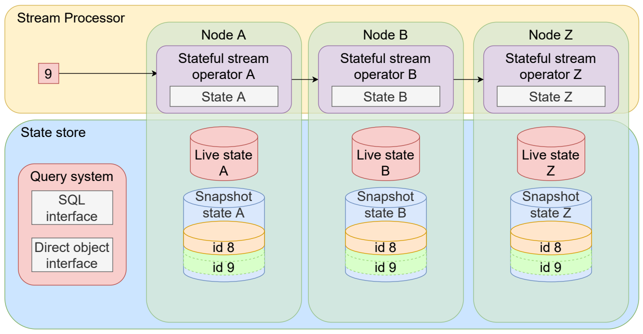


Fig. 1: S-QUERY's architecture. Stream processors store state in the state store which is in turn queried by the query system. The query system only queries snapshot 8, as snapshot 9 is still in progress.

scale linearly with nodes in the cluster. S-QUERY is available as open source software.[1]

With this paper we make the following contributions:

- We present the design and implementation of S-QUERY, the first approach that enables high-level interfaces, such as SQL, to query the state of distributed streaming systems.
- We provide different isolation levels on state queries (read committed, repeatable read, and serializable) alongside use-cases where those can be used.
- We make a thorough performance evaluation with queries of the NEXMark benchmark highlighting the tradeoffs between state size, checkpoint frequency, isolation levels, performance, and scalability.
- We describe a real-world application of S-QUERY for real-time order-delivery monitoring in Delivery Hero SE, a global company offering Q-commerce services.

II. Approach Overview

S-QUERY is an approach for querying the state of a distributed stream processing system where the state is dispersed over the system's operators. S-QUERY distinguishes between two modes of state: live state and snapshot state. Live state is the actual running state of an operator at any given moment of execution. Snapshot state is a past version of state captured by a checkpoint. Each checkpoint records a version of the state back at the time when the checkpoint was taken. S-QUERY can query both live state and snapshot state, as required depending on the use-case. Finally, since a streaming system stores past snapshots of its state, S-QUERY can query that state as it evolves with time.

Architecture. Figure 1 depicts the high level architecture of S-QUERY. The architecture consists of two separate but tightly coupled systems: the *stream processor* which comprises a Directed Acyclic Graph (DAG) of stateful operators and the *state store* which is a partitioned database system, such as an in-memory key-value store. Any change in the stateful operator state is directly reflected in the *state store* by S-QUERY and updates the live state stored there. At the same time, the state store holds the snapshots that are triggered by the checkpointing mechanism [3], [11] of the stream processor.

The query subsystem of the state store, computes queries on the snapshot state or the live state. The live state can be accessed directly, whereas queries to the snapshot state require a specific snapshot id. By default, the latest snapshot id is implied when querying the snapshot state, but users are free to pinpoint any valid snapshot id. In the case of Figure 1, snapshot with id 9 is still being processed, so 8 is the latest snapshot id which corresponds to the latest completed checkpoint.

Requirements. S-QUERY imposes two requirements to a host streaming system: $i)$ a checkpoint-based approach that produces consistent state snapshots and $ii)$ queryable state storage. S-QUERY enables queries to the streaming system's live and snapshot state in a number of ways. First, by providing, protecting, and optimizing access to the live state. Second, by managing multiple versions of snapshot state consistently across the distributed system, and finally by optimizing the memory or disk space utilized by state snapshots through the support of incremental state snapshots.

Colocating State & Compute. Note that in Figure 1 we can see that both the live state of each operator as well as the respective snapshots are colocated with the compute layer. This is one of the main design decisions enabling an important optimization introduced by S-QUERY: in order to guarantee local updates to the state without going through the network for each state update, the state store and the stream processor share the same partitioning function and, thus, the system's scheduler enforces that the state and compute of the same partition are colocated.

III. The Case for Queryable State

The capability of exposing and querying the distributed state of a streaming system, which in short we refer to as queryable state throughout the paper, introduces novel and important advancements and use cases that we highlight in this section.

Substituting Caching and Static Views. For half a century people have been using databases for application state management. With the advent of the web and mobile devices, user experience requirements have increased sharply, putting pressure to the database layer. In order to provide a satisfying experience to end users, a level of indirection, caching, has been added to make the desired data available faster.

Caches provide the expected performance, but they introduce a difficult problem. Being a level of indirection, caches have to be in sync with the database in order to provide consistent results, not just fast. Keeping a cache consistent in a distributed setting, that is the norm for many real-world web applications, is challenging and adds significant overhead. Moreover, an application has to retrieve the cached data and compute views over the data at the application side. The views are most probably computed on stale data, since the

[1]https://github.com/delftdata/s-query

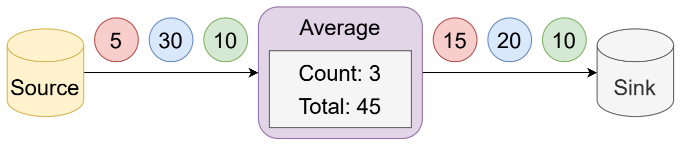


Fig. 2: Stream processing pipeline with stateful 'average' operator. Input items are 10, 30, and 5 respectively. Output items corresponding to the input items have the same colors (and same order as in the input stage). Internal state of the operator is in the rectangle inside the operator.

data may have changed in the database in the meantime. Finally, this extra work that is irrelevant to the application logic, burdens application developers who suddenly have to deal with memory space overheads and efficient computation of views.

Instead of constantly transferring data from the database to the cache in order to maintain the consistency between the two different levels of state management, continuous queries on data flowing in a streaming system and ad-hoc queries on the system operators' state obviate the need for different levels of state management and relieve applications from the programming and processing overhead of view computation in a scalable and fault tolerant fashion.

Reducing Complexity. It is quite common for streaming systems to update an external database with intermediate results. That database is then used to answer application queries. However, using queryable state, this extra database layer can be removed and the application can directly query the state of the streaming system. Additionally, since traditional relational database management systems (RDBMS) are hard to scale out, they can become bottlenecks in high throughput data workloads, which are well met by streaming systems. Thus, queryable state reduces database dependencies and potential bottlenecks from a streaming topology.

Auditing and Compliance. Queryable state makes streaming systems auditable. The processing of personal data is one important case. According to article 4 of EU's GDPR, 'processing' means any operation that operates on personal data [12], therefore streaming systems used to process personal data need to comply with GDPR. In addition, individuals also have the right to request their personal data as defined in article 15 of the GDPR [12]. Thus, organizations using streaming systems need to provide even their internal state on request. Therefore, the ability to query the internal state of a streaming system strongly facilitates regulation compliance.

Debugging. Currently, debugging stateful streaming topologies is a daunting process. With state being internal, only the streaming system's input and output are observable. With queryable state, however, it is possible to have a complete view of this state, or to isolate part of it. This makes debugging stateful operators easy, as one can access their internal state just like usual data stored in a database. Furthermore, if there is also the option of switching between specific versions of the state, one would also be able to see how the state mutates over time. This is an invaluable capability for debugging complex streaming systems.

Simplifying Streaming Topologies. Often developers need to include new computations on a streaming job for ad-hoc analytics, auditing, and other use cases. Currently, these ad-hoc computations are adapted to existing jobs making them more complex, which also leads to reduced maintainability and higher resource consumption.

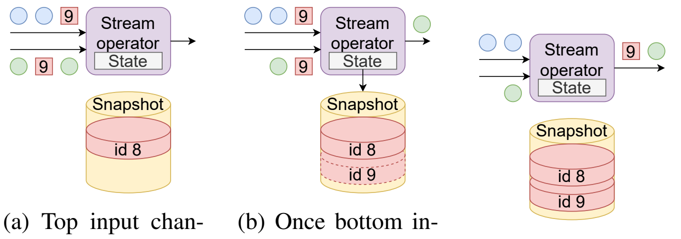
Let us consider the example of a simple streaming job depicted in Figure 2. The internal state of the job is a counter and the total sum of all items, which are used to calculate the average. Now imagine if, besides the average, there is a need to know the amount of items that have come in so far. A new job can be created that also takes in the same items, and outputs the amount of numbers that it received. With queryable state though it is possible to query the amount of numbers directly from the state of the existing averaging operator. By querying the operator's state, the need for an extra (or more complex) streaming job is eliminated.

IV. Preliminaries: State Management in Streaming

This section provides background knowledge on the streaming model assumed in this paper, state management and fault tolerance fundamentals, and the pertinent role of key-value stores in stream processing.

Streaming Model. We consider a typical stream processing model in which stream processing jobs are modeled as a directed acyclic graph (DAG) of operators [11], [13], [14]. The edges on the DAG represent the data streams and the vertices represent the operations on the (incoming) data edges. An output edge points to downstream operator(s). In order to distribute a streaming job over a cluster, stream processing systems typically perform data partitioning and deploy one or more instances of a partitioned operator on each cluster node (or CPU core) and connect upstream with downstream operators across nodes when downstream operators are partitioned by key range.

State Management & Fault Tolerance. In short, the most common state management approach in stream processing involves periodic coordinated checkpoints, which are performed when special markers that flow through the topology of a dataflow graph, instruct the operators to snapshot their state. Checkpoint markers are a specific instance of punctuations [15], metadata annotations embedded in data streams for communicating information orderly across the dataflow graph. The state of operators is typically stored in stable storage in order to survive node failures. In order to achieve exactly-once processing guarantees, a marker alignment phase ensures that operators with multiple input channels will not process inputs following a marker until all markers are received by all input channels and the checkpoint is performed. During recovery, all operators of the system roll back to the latest checkpoint and start processing input from that point onwards ensuring

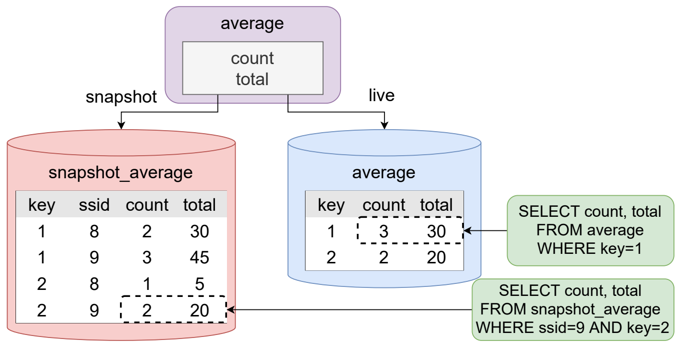


(a) Top input channel at marker 9, bottom input channel still processing. Top input channel needs to wait.

(b) Once bottom input channel has finished processing and is at marker 9, snapshot with id 9 is created.

(c) Snapshot done, broadcast marker to output channels and continue processing input records.

Fig. 3: Marker alignment phase on checkpoint for operator with two input channels. Red squares are checkpoint markers, circles are records belonging to different checkpoints.



key	ssid	count	total
1	8	2	30
1	9	3	45
2	8	1	5
2	9	2	20

key	count	total
1	3	30
2	2	20

```
SELECT count, total
FROM average
WHERE key=1
```

```
SELECT count, total
FROM snapshot_average
WHERE ssid=9 AND key=2
```

Fig. 4: S-QUERY stream operator state representation for both live and snapshot state including queries.

that the processing of each input record will be recorded to an operator's state exactly-once.

Figure 3 depicts this process. First, Figure 3a depicts an operator with two input channels. The top channel is at marker 9, while the bottom channel still needs to process some input records. The operator will only start taking a snapshot of its state when all markers arrive, as shown in Figure 3b. After the snapshot is complete, the operator will forward the marker to the output channel(s) and continue processing the records from the input channels, as is shown in Figure 3c. S-QUERY takes advantage of such a checkpointing mechanism found in many streaming systems in order to make the stream processor's state available for querying.

Key-Value Stores & Relation to Stream Processors. Many streaming systems, such as Apache Spark [16], Apache Flink [9], Facebook Puma [17], IBM Streams [13], and Hazelcast Jet [10], opt for a key-value store as a state backend. This design decision aligns well with the data-parallel shared-nothing architecture of modern streaming systems where data streams are split into a number of partitions based on a key and are typically processed by an equal number of parallel instances of each operator. Thus, each operator instance in the course of its processing maintains the state of a particular data partition.

In similar spirit, modern key-value (KV) stores such as Cassandra [18] and DynamoDB [19] implement a distributed map data structure, comprising a key and value. Like stream processors partition streams, KV stores partition their key space on multiple machines with a partitioning function, e.g., based on hashing. Consequently, KV stores are a perfect match for streaming systems' state backend. Finally, KV stores typically support a dialect of SQL, which allows external applications to query the KV store.

V. Exposing Internal State to External Systems

In this section we detail how S-QUERY utilizes a KV store in order to make the state of a stream operator externally visible and queryable.

A. *Storing Operator State in a KV Store*

A distinguishing feature of S-QUERY is that it stores both its live state and its snapshots in a KV store, using the same partitioning key as the key of the operator holding that state as seen in Figure 1. This way, instead of performing remote calls for each change to the operator state that resides in the KV store, the change remains local, allowing for high throughput processing of events. The same holds for snapshots: each snapshot is first written locally and the KV store can replicate it according to its internal replication strategy, typically implementing Paxos [20] or Raft [21]. Again, at first the snapshots are only written locally speeding up both the checkpointing mechanism but also the recovery process. If a node fails, the respective operator can be scheduled on the node holding that snapshot's replica.

B. *Modeling & Storing State Externally*

S-QUERY uses two KV data structures per stateful operator in the dataflow graph: one for live state, and one for snapshot state as depicted in Figure 4. It is important to note here that we assume that the state of an operator is in the form of a map data structure, i.e., it holds a key and an associated value to it. Finally, note that the value can be any object (e.g., complex objects in Java, Python, etc.).

Storing Live State. The schema for storing live state in a KV data structure is shown in Table I. The key of the table simply corresponds to the key of the actual operator state and it is stored along with the corresponding state object, which becomes the value object in the KV store. Each KV data structure is named after the operator whose live state it holds. Finally, the name of a KV data structure is used to address

TABLE I: Live state structure

Key	Value
Key	State object

TABLE II: Snapshot state structure

Key		Value
Key	Snapshot ID	State object

SQL queries to the live state of the corresponding operator. As seen in Figure 4, the operator is called `average` and that matches the name of the KV data structure. Note that if the `average` operator has multiple instances i.e., it runs partitioned in multiple nodes, the distributed KV data structure called `average` will contain all the KV pairs of the state of all these operators across the cluster.

Storing Snapshots. The way that snapshot state is stored, can be seen in Table II. Storing snapshots differs slightly from storing live state in that the table is now formed by two elements: the key of the keyed state (determining the partition), and the snapshot ID. Again the value is the actual state object. This makes it possible to store different snapshots of the same keyed state independently. This is useful for the cases when different versions of the state need to be kept, either for auditing reasons or for historical queries. The name of the table holding the snapshot state is -by convention- `snapshot_<operator name>`, where `<operator name>` is the name of the stateful operator in the DAG for which the snapshot state is stored. For example, if the operator name is `statefulmap`, then the table containing the live state is also identified by `statefulmap`, and the table storing the snapshot state is called `snapshot_statefulmap`. Using this mechanism each stateful stream operator has its own live/snapshot state KV data structure, which stores the complete state of the operator that is distributed across its partitioned instances in different nodes. In addition, it is also possible in S-QUERY to enable/disable the live/snapshot state mechanism. In case only snapshot state is needed, the live state can be disabled for better performance.

C. Querying Live & Snapshot State

The KV data structures storing the state can now serve as the connection point between the streaming system and applications or other external systems that want to query the state of a stream processing job. As seen in Figure 4, an SQL query can be used to query the state of a running operator, or the state at different moments in time captured by snapshots during the lifetime of the streaming job.

VI. S-QUERY IN STREAMING SYSTEMS

As we briefly mentioned in Section II, the ideas behind S-QUERY can be implemented in different systems given that they satisfy two main requirements. The first is that the host streaming system offers a checkpointing mechanism that produces consistent snapshots and the second is a queryable state backend. In this section we detail how this applies to different streaming systems and state-store combinations and present our implementation in detail.

Many modern stream processing systems, including Apache Flink [9], Apache Spark [16], Jet [10], and IBM Streams [13] have converged to a variant of Chandy-Lamport's seminal distributed snapshots protocol [22], which has been adapted to stream processing [3], [11] leveraging also transactional queues, such as Apache Kafka [23]. At the same time stream processors use a state storage backend, typically a Log-Structured Merge Tree [24] implementation such as RocksDB [25] or FASTER [26]. In order to query the state, S-QUERY requires a query engine that can query both the snapshot state and the live state of operators. It is S-QUERY's job to provide safe, queryable access to the two types of state. In the following, we first detail how S-QUERY achieves this with Hazelcast Jet and IMDG, and how a similar implementation would work with Apache Flink and Cassandra.

A. S-QUERY *on Hazelcast Jet with IMDG*

We implement S-QUERY on top of Hazelcast Jet [10] since Jet fulfills the requirements of S-QUERY: it features a checkpoint-based state management approach and a KV state backend with an SQL interface,[2] Hazelcast's distributed in-memory data grid (IMDG). Jet leverages the IMDG as a low-latency backend for storing state snapshots.

S-QUERY enables external queries to the distributed state of Jet operators by exposing the live and snapshot state of each operator individually as first class key-value structures in the KV store. Table I and Table II depict the key-value structures storing an operator's live state and snapshot state respectively. Specifically, S-QUERY propagates updates on the live state of each operator to the designated data structure (Table I) in the KV store and performs access synchronization between live state updates and query processing.

In addition, S-QUERY accommodates snapshot state in the KV store in a queryable way by exposing the key-value pairs contained in the snapshot. Formerly, snapshot state in the KV store was a mere blob structure. Furthermore, S-QUERY ensures that that the latest snapshot is atomically acknowledged across the distributed system in order to guarantee that a query is answered from the most recent snapshot at the time the query is issued.

Snapshot Versions. By default, S-QUERY keeps the two most recent state snapshot versions, a design that results in constant memory usage and ensures that there is at least one version available for querying. A new snapshot overwrites the oldest of the two snapshots. If maintaining more versions of the snapshot state is important to an application, S-QUERY can be configured to preserve many versions. In this case S-QUERY accommodates snapshots by version so that a result set can integrate the state of multiple snapshot versions with explicit mention of each key-value pair's snapshot version.

Incremental Snapshots. As an optimization, S-QUERY performs incremental snapshots and supports queries to them in order to relieve the state backend from holding complete state snapshots. Instead, after each checkpoint only the latest changes are recorded while S-QUERY retains versions of the incremental snapshots and executes queries on them starting from the latest snapshot of interest, which contains the most recent state updates for a set of keys, and going backwards to supplement the query results with the latest state updates for other keys. A downside of incremental snapshots is that as they increase, the overhead of this differential query process

[2]https://docs.hazelcast.com/imdg/4.2/sql/expressions.html

increases as well. For this reason, S-QUERY prunes obsolete states, thereby providing for better query performance and less memory/disk space allocated to snapshots.

Finally, S-QUERY extends the SQL interface exposed by Hazelcast IMDG with join operations, thereby enriching significantly the expressiveness of queries that can be conveyed.

B. S-QUERY *on Apache Flink and Cassandra*

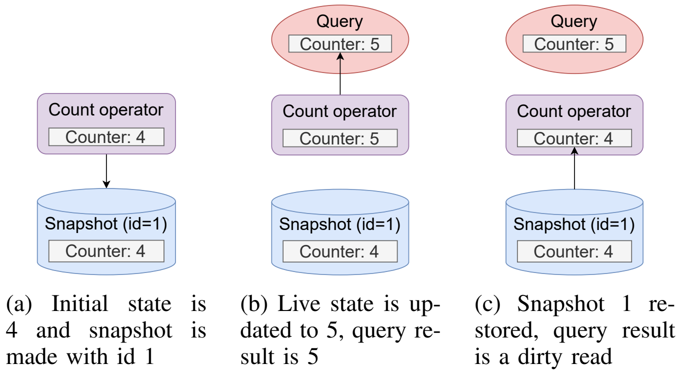
Although S-QUERY is implemented in Jet and IMDG, the very same principles can also be implemented with Apache Flink and an external distributed database, such as Cassandra. Normally, Apache Flink's operators use RocksDB as their state backend, and each checkpoint is stored on stable storage (e.g., HDFS or blob store such as AWS S3). If RocksDB, which is not distributed, were to be used, a distributed query engine that can perform joins on the local states of each of the partitioned streaming operators would also be required. On the other hand, RocksDB supports incremental snapshots out of the box and handles compaction of states transparently and optimally. For instance, level-based compaction bounds read amplification and would reduce the search time for historic changes per key, which now limits the performance of S-QUERY. With Cassandra S-QUERY can match Cassandra's key-range partitioning to the operator key-range partitioning in Flink like with Jet and IMDG. Contrary to IMDG, since Cassandra is disk-persistent, it is cheap to keep multiple snapshots at the same time. Finally, an alternative approach would be to import live and snapshot state updates in a database like MySQL.

VII. ISOLATION LEVELS

S-QUERY exposes the internal state of a distributed streaming system to external applications. The offered capabilities of querying either the live or the snapshot state of the system provide diverse performance characteristics and isolation and consistency guarantees. In this section, we discuss the different isolation levels [27]–[29] and elaborate on the isolation levels that S-QUERY offers as well as how S-QUERY could achieve higher isolation levels.

A. *Use-cases for different isolation levels*

The ability to cover different isolation levels in S-QUERY opens the road for different use-cases. For instance, for debugging purposes, one may want to read uncommitted state, as observed in the state of the system while it is running. As we detail in the next subsection, this can be covered by the "read uncommitted" isolation level. Now, let us consider the case where an operator stores the state of a shopping cart checkout function that needs to update the inventory/stock of items in an online shop, while another operator holds the state of the current stock. In that case, one would want to wait until the markers of the system's checkpointing mechanism have traversed the whole topology. In that case, the shopping cart would only be observed to be in the "checked out" state, only after the stock has been updated. This would be the "read committed" isolation level. In the following we detail how different isolation levels are implemented in S-QUERY and what that entails for the performance of the system.



(a) Initial state is 4 and snapshot is made with id 1 (b) Live state is updated to 5, query result is 5 (c) Snapshot 1 restored, query result is a dirty read

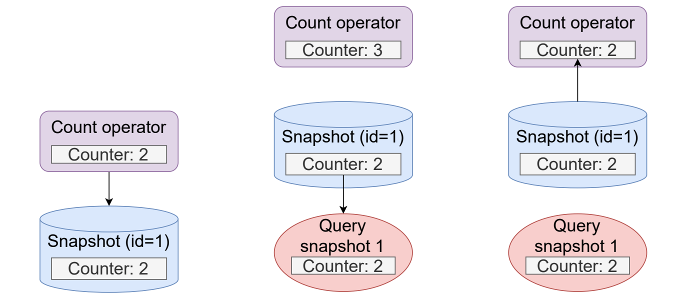
Fig. 5: Live state isolation level example.

B. *Isolation Levels in* S-QUERY

Read uncommitted. The most relaxed isolation level in the database literature is *read uncommitted*, which allows uncommitted state updates to be observed by concurrently executing queries. This is true for updates and queries performed on the live state by S-QUERY, since a failure will force the streaming system to roll back its state to the latest checkpoint and reprocess input from that point onward. Then queries on the live state issued while the system is recovering will observe past state compared to the observed state prior to failure. Notably, in case of processing nondeterministic computations, the state that evolves after fault recovery can also diverge from the state that existed prior to failure. To avoid this effect, nondeterministic computations can be captured and handled in a way that produces deterministic results [11], [30], [31]. In summary, live state updates are considered uncommitted until the next checkpoint takes effect and, consequently, live state queries read dirty data resulting in the read uncommitted isolation level.

Take, for instance, the following example where the state of a stream processor that counts incoming records is 4. At that point a checkpoint is taken that creates a snapshot with id 1 (Figure 5a). Following the checkpoint the state becomes 5. Then, a query on the live state returns 5 (Figure 5b). However, before the state could be committed to a new snapshot, the job fails. Now, according to the query, the state is still 5, but in reality the state is 4 after the new processor recovered its state from the latest snapshot (Figure 5c).

Read committed. At the *read committed* isolation level concurrently executing state updates and queries will only read committed (i.e., checkpointed) values. S-QUERY could raise the isolation level it offers for live state queries to read committed by providing a high availability setup using active replication [32] where hot standby operators maintain a replica of the state by processing synchronously the same updates as the primary operator. This setup ensures that if the primary operator fails, the standby can substitute it immediately and continue to process updates and queries moving forward from the point of failure. A roll back of the state will never be



(a) Initial state is 2 and snapshot is made with id 1

(b) State is updated to 3, query result for snapshot 1 is 2

(c) Snapshot 1 restored, query result is still correct

Fig. 6: Snapshot state isolation level example.

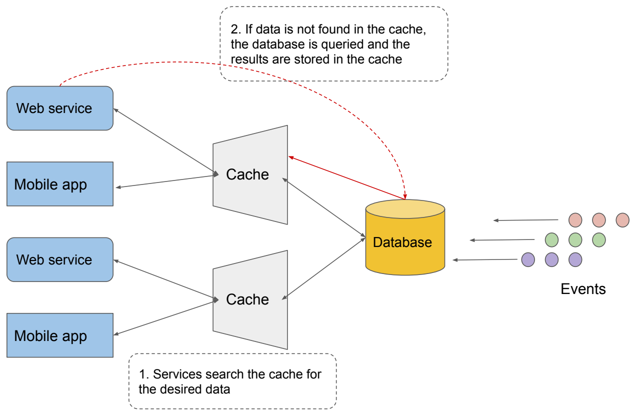

2. If data is not found in the cache, the database is queried and the results are stored in the cache
Web service
Mobile app
Cache
Web service
Mobile app
Cache
Database
Events
1. Services search the cache for the desired data

Fig. 7: Traditional caching architecture

required. In fact, standby operators can also process queries, thereby helping to balance the query load. Combining the high availability setup with S-QUERY's key-level locking in order to perform an update, reliably results in the read committed isolation level. Another approach would be to use a transactional state backend where every update to the state is persisted reliably like Google's Millwheel [33] does with Spanner [34]. Interestingly, the read committed isolation level is offered by S-QUERY for live state queries if we assume no failures. This is because if the system does not fail, there is no event that can cause state updates to become unstable and S-QUERY protects state updates from read actions via key-level locking for the duration of access to each key-value pair.

Repeatable read. The *repeatable read* isolation level entails that all of the state that is read or written in the course of an update or query is completely protected from read, update, and delete actions on the same state attempted by other transactions. To abide to the repeatable read isolation level for live state queries, S-QUERY could hold the locks for the whole duration of query processing instead of releasing the locks after each individual read or write action to a key-value pair. On the downside, this design decision would severely affect both the streaming system's performance, as well as the performance of the state querying.

Snapshot isolation guarantees that each transaction operates on a snapshot of the state that is completely isolated from operations carried out by other transactions. It permits concurrent operations on different snapshots of the state thereby offering better performance at the working stage of transactional processing. However, in case the updates performed by a transaction intersect with updates made by another transaction on the same state, conflicts arise at commit time, which cause the transaction to be aborted. S-QUERY's snapshot state query configuration guarantees snapshot isolation since queries are executed on the latest committed snapshot of the state and they are unaffected by uncommitted changes to the live state. When a new snapshot is created it becomes atomically available to the distributed system, and is immediately used in query processing thereby evading phantom reads, which would be possible if the snapshot update process was not atomic.

Serializable isolation. The handling of write conflicts is what separates snapshot isolation from the *serializable* isolation level where the workings of transactions produce a schedule that is always equivalent to a serial schedule of execution. Serializability is enforced by using locks, including range locks on selected data, or multi-version concurrency control. However, in S-QUERY there is no notion of concurrent updates that can create write conflicts neither in live state operations nor in the way that snapshot state is created. Live state updates are performed by parallel instances of single-threaded operators in disjoint state partitions. This is a fundamental architecture decision adopted by many streaming systems like Flink and Jet. Thus, state updates are serialized by design and concurrent updates on the same state are impossible. Finally, a state snapshot is produced periodically by crystallizing the distributed state of the systems' operators into a single snapshot. By deduction snapshot state queries in S-QUERY provide serializable isolation.

Let us consider an example that highlights the aforementioned situation. At some point the state of a stream operator is 2. Then a snapshot with id 1 is created (Figure 6a). After the checkpoint, the state is updated to 3. A query to the snapshot state is issued with the latest snapshot id, which is 1, so the result will be 2 (Figure 6b). Even if the stream operator were to fail and recover, the query result will always be 2 as the query specifically targets snapshot 1 (Figure 6c).

VIII. Use Case: Q Commerce in Delivery Hero

Delivery Hero SE is a global company enabling Q-commerce in more than 50 countries. Q-commerce is an advancement of e-commerce offering fast, on-demand delivery with innovations at the last mile of delivery. To serve its customers, the company relies on a sophisticated large-scale software infrastructure where fast access to consistent data is of outmost importance. Therefore, databases hosting daily data about points of sales, orders, purchases, and rider locations are supported by caches in order to respond fast to user requests sent via web browsers and mobile applications according to the architecture in Figure 7.

Caching as a pattern is used in application development to avoid the load of complex operations by storing results from expensive database queries to intermediate memory-based layers like Redis [35] or Memcache [36]. With this common pattern, application programmers can develop scalable and low-latency web services. At the same time, however, they are forced to adopt the following issues that arise with it.

- The engineering team has to implement the caching mechanism, dealing with sophisticated issues, such as throttling and invalidation, since caching is implemented at the application layer.
- A time-to-live is a common functional requirement for each data source that is being cached, resulting to stale data for a period of time.
- Out-of-the-box systems such as Redis, do not support queries to the data thereby pushing the manipulation to the application level.

The aforementioned issues increase the complexity of the services and rely on the expertise of the engineering team to implement them properly for use in production. In addition, the caching pattern promotes duplication of development effort across the organization, since each engineering team should develop its own caching solution.

Instead S-QUERY can substitute caching along with its inefficiencies leading to a more scalable and efficient system as depicted in Figure 1. In this paper we demonstrate S-QUERY's effectiveness and efficiency by applying it to a real workload composed of order delivery events ingested by a Jet job, which accumulates state for rider locations, order statuses, and order information in each of the job's operators respectively. We use four real queries to evaluate the expressiveness and performance of S-QUERY, Query 1, Query 2, Query 3 and Query 4. Each of the queries captures the need for a real-time ad-hoc view on the state of orders in the system that can guide on-the-spot business decisions and improve customer service. The data stream workload consists of the following events.

Rider location includes the coordinates of the delivery rider with latest update timestamp.

Order status contains the state of an order, that is from `ORDER_RECEIVED` to `PICKED UP` to `DELIVERED` (and several other states omitted for space savings). It also includes a deadline when it should have transitioned to the next state.

Order info is a one-time event per order containing general information about an order such as customer location, vendor location, and vendor category.

IX. EVALUATION

We evaluate S-QUERY on a) a real query workload capturing real-time views of actual, anonymized, online Q-commerce order and delivery data provided by Delivery Hero SE and b) NEXMark [37], the de facto benchmark in stream processing. The real anonymized data provided by Delivery Hero SE have been enriched with data generated based on the real data in order to achieve the throughput required for the duration and scale of the experiments. We focus on four different dimensions of S-QUERY's operation. First, we measure S-QUERY's overhead to Jet in terms of latency (Section IX-B. Second, we analyze the overhead of S-QUERY to Jet's snapshot mechanism with and without query execution(Section IX-C). Third, we present S-QUERY's performance using four real queries on a workload of online order and delivery events that is central to the everyday business of Delivery Hero SE (Section IX-D). We also compare S-QUERY's query performance against TSpoon [8]. We chose TSpoon because it was the only approach that could run part of our target workload in a distributed setting with isolation guarantees.

Finally, we study the scalability of S-QUERY by measuring the system's throughput with different cluster sizes (Section IX-E).

A. Experimental Setup

Two clusters of 7 nodes in Amazon AWS, one of them provided by Delivery Hero SE, were used in the experiments. The hardware specification of the cluster nodes used in the experiments is detailed in Table III. Per cluster node, 12 CPUs are used for processing data in Jet while the other 4 are used for garbage collection. The same configuration was chosen by Jet's development team for evaluating Jet [10]. In this work, the 4 CPUs used for garbage collection are also used to process S-QUERY queries.

```
        SELECT COUNT(*), deliveryZone FROM
↪  "snapshot_orderinfo" JOIN "snapshot_orderstate"
↪  USING(partitionKey) WHERE
↪  (orderState='VENDOR_ACCEPTED' AND
↪  lateTimestamp<LOCALTIMESTAMP) GROUP BY
↪  deliveryZone;
```

Query 1: How many orders are late (in preparation by the vendor for too long) per area?

```
        SELECT COUNT(*), vendorCategory FROM
↪  "snapshot_orderinfo" JOIN "snapshot_orderstate"
↪  USING(partitionKey) WHERE (orderState='NOTIFIED'
↪  OR orderState='ACCEPTED') GROUP BY
↪  vendorCategory;
```

Query 2: How many deliveries are ready for pickup per shop category?

```
        SELECT COUNT(*), deliveryZone FROM
↪  "snapshot_orderinfo" JOIN "snapshot_orderstate"
↪  USING(partitionKey) WHERE
↪  (orderState='VENDOR_ACCEPTED') GROUP BY
↪  deliveryZone;
```

Query 3: How many deliveries are being prepared per area?

```
        SELECT COUNT(*), deliveryZone FROM
↪  "snapshot_orderinfo" JOIN "snapshot_orderstate"
↪  USING(partitionKey) WHERE orderState='PICKED_UP'
↪  OR orderState='LEFT_PICKUP' OR
↪  orderState='NEAR_CUSTOMER' GROUP BY
↪  deliveryZone;
```

Query 4: How many deliveries are in transit per area?

TABLE III: c5.4xlarge node properties

CPU	16 vCPUs
Memory	32 GB
Network	10 Gbit/s
OS	Ubuntu 20.04.2 LTS
Java	AdoptOpenJDK (build 15.0.2+7)

Overhead experiments measure the latency from source to sink, that is, how long it takes for the effects of an input record to reach a sink. The experiments are executed on a three-node cluster with a warmup period of 20 seconds and a measurement period of 240 seconds. The streaming job executed by Jet in the overhead experiments is query 6 of Apache Beam's Nexmark benchmark implementation.[3][4] The job computes the average selling price for each seller in an auction from a bid and auction stream. It accumulates state for 10K auction sellers and checkpoints state snapshots every second. The average selling price is taken over the last 10 auctions per seller.

The snapshot performance experiments are performed on the Delivery Hero SE workload (Section VIII) demonstrating that S-QUERY is capable of supporting real world applications. For the snapshot experiments, 1K, 10K and 100K unique keys, representing the number of orders in Jet's state, provide a variable and significant workload for the snapshot management system, with a snapshot interval of 1 second. The snapshot latency is measured at three points in the node that controls the 2PC process, before phase 1 begins, after phase 1 completes, and after phase 2 completes. Two concurrent threads run queries on the state in parallel at full speed to create a significant workload on the system. Each configuration was run for at least 20 minutes, with the first minute used as a warmup period.

For the query experiments there are two setups, the SQL query experiment, which shares the same setup as the snapshot performance experiment and the direct object access experiment, which uses a 3 node cluster totalling 48 vCPUs to compare to related work, TSpoon [8].

Finally, the scalability experiment is performed on NEXMark query 6 while varying the cluster size between 3, 5, and 7 nodes. The number of unique keys that represent the number of auctions is 10K, same as in the overhead experiments. In parallel to the job execution of query 6, S-QUERY is used to input and process 10 SQL queries per second that select the list of the 10 latest auction prices.

B. Overhead Experiments

The experiment results for query 6 of NEXMark are shown in Figure 8. The experiment consists of four configurations, a) S-QUERY with both live and snapshot state enabled, b) S-QUERY with only live state enabled, c) S-QUERY with only snapshot state enabled, and d) Jet. Live state incurs significant overhead, which is to be expected, since it amounts to communicating every single state change that happens at each operator of the system to the respective live state representation in IMDG. The latency distribution of S-QUERY's snapshot state configuration is almost identical to Jet's configuration. For the rest of the experiments, we focus our evaluation on the snapshot state configuration.

Figure 9 compares S-QUERY's snapshot state configuration to Jet at three input throughput levels, 1M/5M/9M events/s. Naturally, higher throughput results in higher latency. At 1M events/s throughput, S-QUERY's overhead is unnoticeable. For throughput at 5M events/s, S-QUERY achieves virtually equal latency as Jet until around the 90th percentile. In higher percentiles S-QUERY becomes marginally slower by 4ms at most. At 9M events/s throughput, S-QUERY's overhead is minor reaching up to 8ms more latency at the 99.99th percentile. In conclusion, S-QUERY achieves similar low latency performance as Jet demonstrating that the state snapshot configuration of S-QUERY has very little impact on the system's latency and throughput.

[3]https://beam.apache.org/documentation/sdks/java/testing/nexmark/
[4]https://github.com/hazelcast/big-data-benchmark

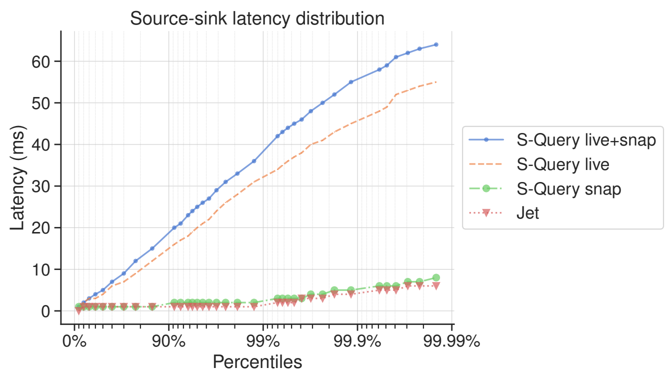


Fig. 8: Latency distribution of S-QUERY's live and snapshot state configurations vs Jet for NEXMark query 6 in 3-node cluster at 1M events/s. X-axis shows percentiles on an inverted log scale; y-axis shows the latency in milliseconds.

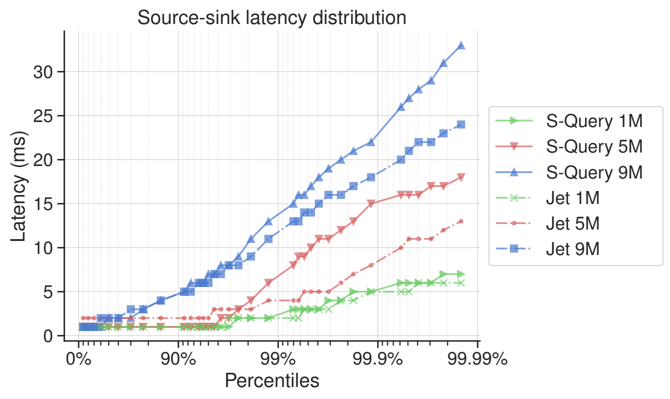


Fig. 9: Latency distribution of S-QUERY vs Jet on NEXMark query 6 in 3-node cluster at 1M, 5M, 9M events/s. X-axis shows percentiles on an inverted log scale; y-axis shows the latency in milliseconds.

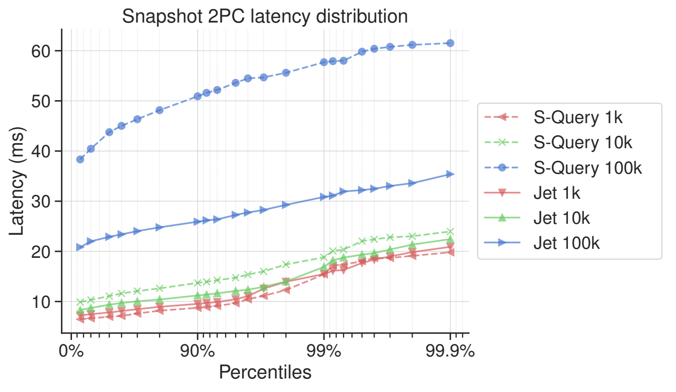


Fig. 10: Latency distribution of snapshot mechanism between S-QUERY and Jet for 1K/10K/100K unique keys across a 7 node cluster. X-axis shows the percentiles on an inverted log scale; y-axis shows the latency in milliseconds.

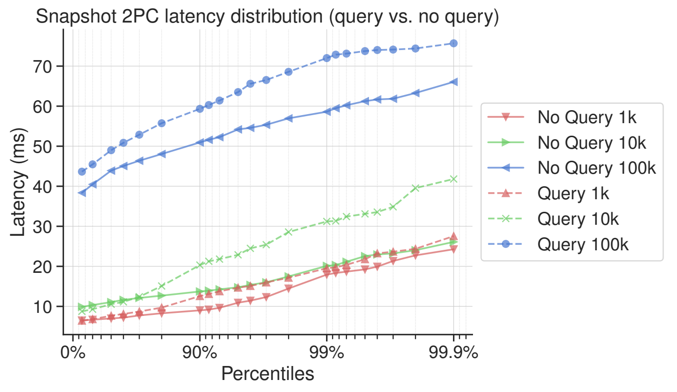


Fig. 11: Latency distribution of snapshot mechanism on 7 node cluster with and without S-QUERY queries for different number of unique keys. X-axis shows percentiles on an inverted log scale; y-axis shows the latency in milliseconds.

C. *Effect of Snapshotting Mechanism on System Performance*

Because S-QUERY introduces changes to the snapshot creation process of Jet, it is important to measure the effect of the snapshotting mechanism on the system's performance with and without query execution on the state. For each of the two configurations we compare the time it takes to commit a snapshot with exactly-once processing guarantees between S-QUERY and Jet considering different snapshot state sizes. The measured snapshot latency determines how much time an operator spends in processing records as opposed to taking snapshots.

S-QUERY Operation Without Queries. Figure 10 shows the snapshot creation time for both S-QUERY and Jet when varying the number of unique keys in the snapshot state. S-QUERY achieves virtually equal latency performance to Jet for 1K keys and is only 2-4ms slower than Jet for 10K keys throughout the distribution. Even for 100K keys of state, S-QUERY is merely 19-27ms slower than Jet. While the difference might seem considerable, the overhead would be unnoticeable to most applications. To further illustrate the impact, the snapshot interval is set at 1 second, which is already very low. The 50th percentile of the worst case, i.e. 100K keys, has a snapshot latency of 44ms for S-QUERY and 23ms for Jet. The implication is that S-QUERY would support a mere 2% lower sustainable throughput than Jet at the worst case, i.e. for 100K unique keys and 1 second snapshot interval. For a smaller number of keys the difference is almost negligible. Consequently, S-QUERY's impact on the performance of the streaming system is minimal.

Query Execution. The execution of S-QUERY queries can affect the snapshot creation mechanism. We measure the latency distribution of the snapshot mechanism as before with and without queries being executed on the snapshot state. For the experiments we use Query 1, which is a relatively expensive query including both a JOIN and GROUP BY clause. Figure 11 shows the impact of queries on the snapshot 2PC latency. For 1K and 10K keys the impact is negligible until the 70th percentile. For 10K keys a small difference appears from the 80th percentile onwards, which increases up to 20ms, while for 100k there is a similar difference throughout the distribution. In general, the impact is up to 14ms across all unique key configurations. Notably, the time it takes to commit a snapshot is much smaller than the snapshot interval itself. Thus, consecutive queries from multiple threads do not affect the performance of a streaming job significantly.

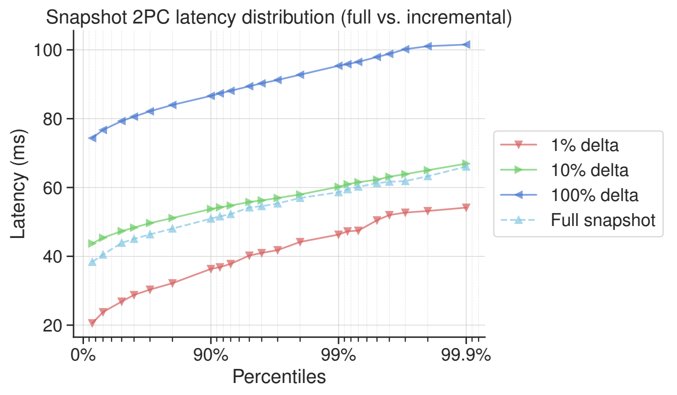


Fig. 12: Latency distribution of incremental/full snapshots on 7 node cluster with 100k unique keys for different snapshot delta ratios. x-axis shows percentiles on an inverted log scale, y-axis shows the latency in milliseconds.

Incremental snapshots. Figure 12 presents the performance of creating incremental snapshots of varying size by tweaking the amount of changes introduced. As expected, the performance of incremental snapshots depends significantly on the snapshot change/delta ratio: for a modest delta ratio, incremental snapshots are considerably more efficient, but when the delta ratio is high they incur non-trivial overhead compared to full snapshots due to the housekeeping involved.

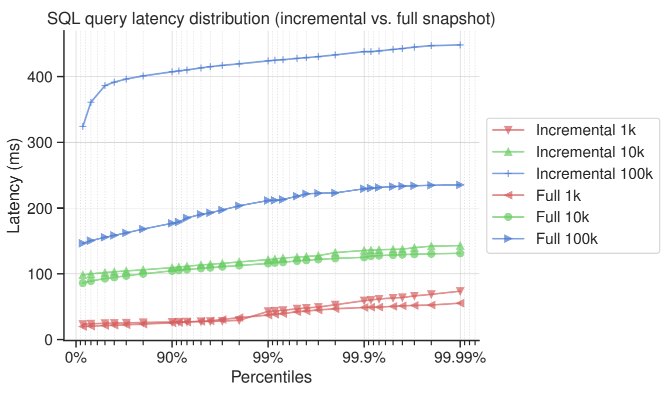


Fig. 13: Query execution latencies on a 7 node cluster for incremental and full snapshots with 1K/10K/100K unique keys. X-axis shows percentiles on inverted log scale; y-axis shows latency in milliseconds.

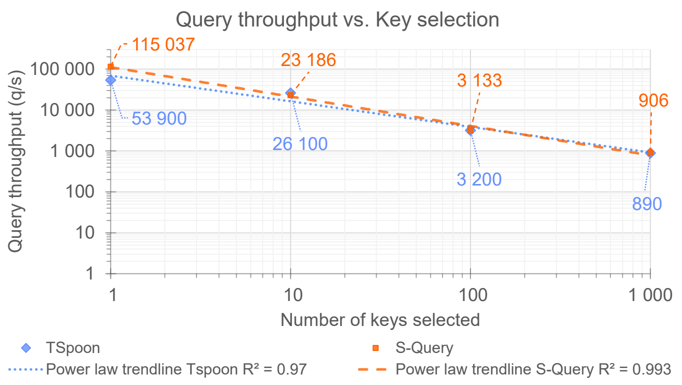


Fig. 14: Throughput of direct object queries between S-QUERY and TSpoon. X-axis shows the number of keys selected in a query at log scale; y-axis shows the throughput measured in queries/s at log scale.

D. Query Performance

We measure the query execution performance of S-QUERY through the SQL and direct object interfaces.

SQL Interface. In Figure 13 we measure query performance for varying state sizes in terms of latency, which is quantified as the time it takes to execute a query and retrieve the complete result set. We use Query 1 in this experiment. As expected, the query execution latency increases with larger state size as a query has to process more state entries. Surprisingly, the performance of queries executed on incremental snapshots is identical to that of full snapshots for 1K and 10K unique keys even though S-QUERY has to join all the incremental snapshots involved. That said, for 100K unique keys, queries on incremental snapshots manifest almost five times additional latency. In summary, incremental snapshots present a tradeoff between savings in snapshot state size and overhead in query execution, and their use should be judged based on application requirements and the amount of state changes introduced across keys.

Finally, the latency distributions remain relatively flat even in the higher percentiles providing the opportunity for reliable SLA agreements. The snapshot ID retrieval times (not shown in the plot) have a median of around 1-2ms and go up to 40ms. They increase proportionally with state size, but this can also be attributed to the increased system load owed to the larger state size. The performance of the queries is reasonable given that latency is measured end-to-end, from query submission to result reception, including network delays. To illustrate, the query on 100K keys works on a dataset of size 22.4MB. Over a 10 Gbit/s network connection it takes 18ms at best to execute the query without taking into account network overhead.

Direct Object Interface. We compare the performance of S-QUERY to TSpoon [8], a system offering simple state queries. S-QUERY queries the rider location operator from the Delivery Hero SE use case from Section VIII, whose state consists of two doubles (coordinates) and the time it was last updated. An external query from the direct object interface retrieves the total state of the rider location operator. The experiment is executed on a 3-node cluster (48 vCPUs). On a fourth identical node, 180 threads are used to send queries to the 3-node cluster in parallel. On the other hand, TSpoon used a double value denoting an account balance as state in their experiment [8] which was performed on a cluster of 50 vCPUs. Figure 14 plots the query throughput of direct object access queries when selecting 1, 10, 100, and 1000 keys out of 100K unique keys for both S-QUERY and TSpoon.

The plot suggests a power law trendline, which fits the data points with an R^2 of 0.993 and 0.97, indicating that the query throughput follows a power law distribution indeed. A power law distribution is reasonable since selecting more keys takes proportional time thereby resulting in proportionally less throughput. Compared to the equivalent experiment from TSpoon [8], S-QUERY outperforms TSpoon by a factor of two when querying 1 key while performing similarly for the other key selections even though S-QUERY is slightly disadvantaged in terms of state size per key and number of vCPUs available for query processing.

E. Scalability

Since modern streaming systems are typically horizontally scalable, it is important to test S-QUERY's scalability. For this purpose, we identify the sustainable throughput for each experiment configuration, which is the throughput at which the system achieves the highest sustainable performance with steady latency. We define different experiment configurations by varying the degree of parallelism and the snapshot interval. The snapshot interval affects the amount of time available for processing and, consequently, S-QUERY's scalability. For the experiment we use query 6 of the NEXMark benchmark as the streaming job and execute 10 JOIN queries per second on the state of the job's operators. This setting simulates a realistic query workload.

The results of the scalability experiment are shown in Figure 15. The existing horizontal scalability of Jet [10] is

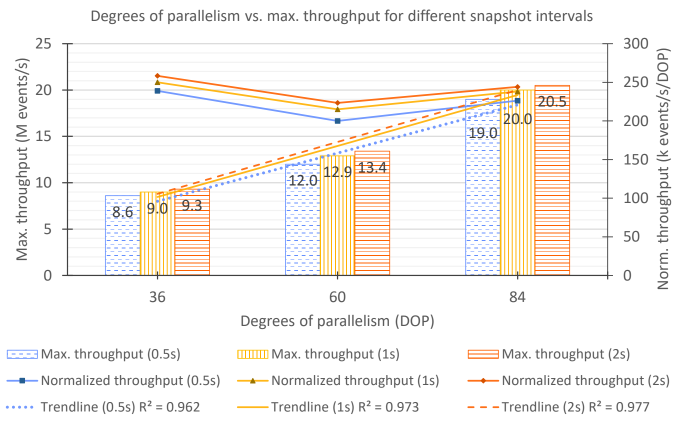


Fig. 15: Scalability experiment results, DOP (degrees of parallelism) on x-axis, max. throughput on left y-axis, normalized throughput on right y-axis. Bars from left to right correspond to 0.5s, 1s, and 2s snapshot intervals respectively.

clearly replicated in our experiment. The R^2 of the trendlines are always higher than 0.96, which means that the sustainable throughput follows a linear relationship with the degree of parallelism conveyed as number of Jet threads. Thus, S-QUERY is horizontally scalable.

The experiment also shows that the snapshot interval has a small but measurable effect on the sustainable throughput. The reason for this is that a smaller snapshot interval results in more time being spent on taking a state snapshot vs. actual processing, which in turn results in decreased sustainable throughput.

X. RELATED WORK

Existing pieces of work focus on state access across operators within a stream processing system, often referred to as transactional stream processing [8], [38]–[42], while others support state queries from external sources [8], [9], [14], [41], [42]. We compare and contrast these pieces of work to S-QUERY.

A. *Transactional Stream Processing*

The Unified Transaction Model (UTM) [38] is one of the first systems supporting transactions in a stateful stream processing system. Using a transaction manager with a strict 2-phase locking mechanism, it provides in-order access to storage resources and rollback functionality, resulting in consistent histories. S-Store [39] supports transactional stream processing by controlling shared state access inside the system with ACID guarantees and exactly-once processing. In contrast, S-QUERY focuses on exposing the live uncommitted state and past consistent state with serializable isolation of a shared-nothing distributed streaming system to external applications. In such systems, shared state access is only possible between live state updates and queries. Transactional streaming workflows over shared state can be a lot more complex and require a full-blown transactional database engine to support. Thus, the clear distinction of use cases drives also a clear separation of concerns. Finally, TStream [40] improves S-Store by dynamically restructuring transactions into operation chains, which can be processed in parallel eliminating lock contention [40].

B. *Queryable State*

TSpoon [8], FlowDB [41], Apache Flink [9], and Apache Samza [14] provide key-value lookup queries to the state of one or more operators of a distributed streming system programmatically while PipeFabric [42] provides ad-hoc queries on the live state of a single-node system.

Queries in TSpoon [8], which is based on FlowDB [41], are treated as read-only transactions. They access transactional parts of the dataflow graph following a transaction commit or abort ensuring sequential execution. Apache Flink [9] provides a simple key-based API for state queries where state has to be explicitly defined as queryable in the code and queries need to specify data types upfront. No synchronization or consistency guarantee is provided and queries may unexpectedly fail. Apache Samza [14] allows streaming jobs to access the state of other jobs using the programmatic Table API in order to enrich streams especially via joins. The Table API supports an in-memory store, RocksDB, and other custom remote tables. Remote tables can allow external applications to query the state programmatically. Finally, PipeFabric [42] uses multi-version concurrency control that guarantees snapshot isolation using commit timestamps for read and write operations.

S-QUERY exposes safe access to the live state and consistent access to past states of a distributed streaming system in a way that higher level query interfaces, such as SQL, can transparently leverage to provide rich ad-hoc queries to external applications. In addition, S-QUERY optimizes snapshot sizes via incremental snapshot support and the access path to the state via co-partitioning of the compute and state layers.

XI. CONCLUSIONS

In this paper we present S-QUERY, a novel approach supporting queries to the distributed state of a stream processing system. S-QUERY is able to query both the live and the snapshot state of a streaming system providing complementary isolation levels, including serializable isolation, and performance characteristics. We evaluate the performance of S-QUERY on the NEXMark benchmark as well as on a real workload from Delivery Hero SE, a global company offering Q-commerce services. We find that S-QUERY adds little overhead to a streaming system when querying the snapshot state, it is horizontally scalable, and is able to perform tens to thousands of queries per second depending on query selectivity. Most importantly, S-QUERY paves the way for novel capabilities by substituting caching layers and intermediate databases commonly used by applications. It also introduces new use cases in the streaming domain, such as auditing and advanced debugging.

Acknowledgements. This work has been partially funded by the H2020 project OpertusMundi No. 870228, and the ICAI "AI for Fintech Lab" project.

References

[1] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," 2020. [Online]. Available: https://arxiv.org/abs/2008.00842

[2] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 725–736.

[3] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: Consistent stateful distributed stream processing," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017.

[4] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 2651–2658.

[5] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos, "Distributed transactions on serverless stateful functions," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 31–42.

[6] A. Akhter, M. Fragkoulis, and A. Katsifodimos, "Stateful functions as a service in action," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 1890–1893, Aug. 2019.

[7] A. Katsifodimos and M. Fragkoulis, "Operational stream processing: Towards scalable and consistent event-driven applications," in *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi, Eds. OpenProceedings.org, 2019, pp. 682–685.

[8] A. Margara, L. Affetti, and G. Cugola, "TSpoon: Transactions on a stream processor," *Journal of Parallel and Distributed Computing*, vol. 140, pp. 65–79, 2020.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[10] C. Gencer, M. Topolnik, V. Durina, E. Demirci, E. B. Kahveci, A. G. O. Lukás, J. Bartók, G. Gierlach, F. Hartman, U. Yilmaz, M. Dogan, M. Mandouh, M. Fragkoulis, and A. Katsifodimos, "Hazelcast Jet: Low-latency stream processing at the 99.99th percentile," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, 2021.

[11] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos, "Clonos: Consistent causal recovery for highly-available streaming dataflows," in *SIGMOD*, 2021.

[12] European Union, "Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation)," *Official Journal L119*, vol. 59, pp. 1–88, 05 2016.

[13] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce, "Consistent regions: Guaranteed tuple processing in IBM Streams," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1341–1352, Sep. 2016.

[14] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at LinkedIn," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017.

[15] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[16] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative API for real-time applications in Apache Spark," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 601–613.

[17] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at Facebook," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1087–1098.

[18] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[20] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[21] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds. USENIX Association, 2014, pp. 305–319.

[22] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[23] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[24] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[25] S. Dong, M. D. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *CIDR*, 2017.

[26] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 275–290.

[27] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[28] American National Standards Institute, *ANSI X3.135-1992: Information Systems — Database Language — SQL*, November 1992.

[29] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 1–10.

[30] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica, "Lineage stash: Fault tolerance off the critical path," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 338–352.

[31] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 13–24.

[32] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 827–838.

[33] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[34] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang, "Spanner: Google's globally-distributed database," in *OSDI*, 2012.

[35] J. L. Carlson, "Database row caching," in *Redis in Action*. USA: Manning Publications Co., 2013, pp. 31–34, ISBN 9781617290855.

[36] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, N. Feamster and J. C. Mogul, Eds. USENIX Association, 2013, pp. 385–398.

[37] P. Tucker, K. Tufte, V. Papadimos, and D. Maier, "NEXMark—a benchmark for queries over data streams," Technical report, OGI School of Science & Engineering at OHSU, Tech. Rep., 2002.

[38] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul, "Transactional stream processing," in *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, E. A. Rundensteiner, V. Markl, I. Manolescu, S. Amer-Yahia, F. Naumann, and I. Ari, Eds. ACM, 2012, pp. 204–215.

[39] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang, "S-Store: Streaming meets transaction processing," *Proc. VLDB Endow.*, vol. 8, no. 13, p. 2134–2145, Sep. 2015.

[40] S. Zhang, Y. Wu, F. Zhang, and B. He, "Towards concurrent stateful stream processing on multicore processors," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1537–1548.

[41] L. Affetti, A. Margara, and G. Cugola, "FlowDB: Integrating stream processing and consistent state management," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 134–145.

[42] P. Götze and K. Sattler, "Snapshot isolation for transactional stream processing," in *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi, Eds. OpenProceedings.org, 2019, pp. 650–653.