DELFT UNIVERSITY OF TECHNOLOGY

MASTER'S THESIS

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

Author:
Jim VERHEIJDE

Thesis advisor:
Dr. Asterios KATSIFODIMOS

Daily supervisor:
Dr. Marios FRAGKOULIS

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

August 20, 2021

Declaration of Authorship

I, Jim VERHEIJDE, declare that this thesis titled, "S-QUERY: Opening the Black Box of Internal Stream Processor State" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

2021-08-20

Date

Signature

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

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

by Jim VERHEIJDE

At the moment we are witnessing the maturation of distributed streaming dataflow systems whose use-cases have departed from the mere analysis of streaming windows and complex-event processing, as they now extend to cloud applications, workflows and even e-commerce. The state of streaming operators has been so far hidden from external applications. In this thesis it is argued that exposing this internal state to outside applications by making it queryable, opens the road for novel use-cases. To this end, we introduce S-QUERY: a system and reference architecture where the state of stream operators can be queried - either live or through snapshots, achieving different isolation levels. It is shown how this 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 snapshot configuration adds only up to 5ms in latency in the 99.99^{th} percentile and no increase in $0\text{-}90^{th}$ percentiles. S-QUERY scales horizontally, allowing for sustainable throughput to scale linearly with nodes in the cluster.

Acknowledgements

This thesis would not have been possible without the help and support of several people. Therefore, I would like to acknowledge them before the start of this thesis.

A large thank you to my supervisors, Dr. Asterios Katsifodimos and Dr. Marios Fragkoulis for the guidance and the great ideas contributing to the end result. And of course the hard work on the paper version of this thesis.

I'd also like to thank Vassilios Karakoidas and Dimitrios Kokmadis from Delivery Hero for taking interest in this work, contributing to the paper, and providing resources for the experiments.

And of course I'd like to thank my family for their unconditional support while writing this thesis.

Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iii

1 Introduction 1

2 Why do we need Queryable State? 3

3 Preliminaries 5
3.1 Streaming Model . . . 5
3.2 State Management & Fault Tolerance . . . 5
3.3 Key-Value Stores & Relation to Stream Processors . . . 6
3.4 Isolation levels in distributed query systems . . . 6

4 Approach Overview 8
4.1 Architecture . . . 8
4.1.1 Colocating State & Compute . . . 8

5 Exposing Internal State to External Systems 10
5.1 Storing Operator State in a KV Store . . . 10
5.2 Modeling & Storing State Externally . . . 10
5.3 Querying Live & Snapshot State . . . 12

6 Isolation levels 13
6.1 Querying Live state . . . 13
6.2 Querying Snapshot State . . . 13

7 Implementation 15
7.1 Platform choice . . . 15
7.2 State structure . . . 15
7.3 Fault-tolerance . . . 15
7.4 Query system . . . 16
7.4.1 Direct object interface . . . 17
7.4.2 SQL interface . . . 17

8 Use Case: Q Commerce in Delivery Hero 18

9 Evaluation 22
9.1 Experimental Setup . . . 22
9.2 Overhead Experiments . . . 23
9.3 Effect of Snapshotting Mechanism on System Performance . . . 23
9.4 Query Performance . . . 26

9.5 Scalability 27

10 Related Work **30**
10.1 Transactional Stream Processing 30
10.2 Queryable State 30

11 Future work **32**

12 Conclusions **33**

Bibliography **34**

List of Figures

2.1 Stream processing pipeline with stateful 'average' operator. Input items are 10, 30, 5 respectively. Output 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. . . . 4

3.1 DAG distribution for two stateful transformations on a keyed stream with 100 unique keys 5
3.2 Marker alignment phase on checkpoint for operator with two input channels. Red squares are markers, circles are records. 6

4.1 S-QUERY system architecture. Stream operators store state in state store which is in turn queried by the query system. Query system only queries snapshot 8 as 9 is still in progress (dashed outline=in progress). 9

5.1 S-QUERY stream operator state representation for both live and snapshot state including queries. 11

6.1 Live state isolation level example. 14
6.2 Snapshot state isolation level example. 14

7.1 Jet state partition structure across a 3 node cluster [17] 16
7.2 Jet state partition rebalancing [17]. Dashed outline are the states from (failed) node 1. 16

8.1 Traditional caching architecture 19
8.2 Stream-based architecture, events are processed by stateful operators with corresponding state inside. External systems are able to query the internal state. 20

9.1 NEXMark query 6 latency distribution, 3-node cluster. x-axis shows percentiles on an inverted log scale, y-axis shows the latency in milliseconds. 24
9.2 NEXMark query 6 latency distribution. Vanilla vs. S-QUERY, 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. 24
9.3 Latency distribution of snapshot mechanism between original Jet and S-QUERY 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. 25
9.4 Latency distribution of snapshot mechanism on 7 node cluster with and without S-QUERY queries for different # of unique keys. x-axis shows percentiles on an inverted log scale, y-axis shows the latency in milliseconds. 26

9.5 Query/snapshot ID latencies on 7 node cluster for different # of unique keys. x-axis shows percentiles on inverted log scale, y-axis shows latency in milliseconds. 27
9.6 Direct object query throughput. Throughput in queries/s on the log scale y-axis, number of keys selected in the query shown on the log scale x-axis. Data labels are S-QUERY on top, TSpoon on the bottom. . 28
9.7 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. 29

List of Tables

3.1 Phenomena possible in different isolation levels [5]. ✓ denotes possible, ✗ denotes not possible. 7

5.1 Live state structure 10
5.2 Snapshot state structure 11

7.1 IMap state structure in Jet 15

8.1 SQL query results for Query 1 and Query 2 20
8.2 SQL query results for Query 3 and Query 4 21

9.1 c5.4xlarge node properties 22

List of Abbreviations

S-QUERY	**S**tate/**S**tream-query
RDBMS	**R**elational **D**atabase **M**anagement **S**ystem
KV	**K**ey **V**alue
ACID	**A**tomicity **C**onsistency **I**solation **D**urability
DAG	**D**irected **A**cyclic **G**raph
IMDG	**I**n **M**emory **D**ata **G**rid
SSID	**S**nap**S**hot **ID**entifier
2PC	**2** **P**hase **C**ommit
DOP	**D**egrees **O**f **P**arallelism

1 Introduction

Over the last decade stream processing systems have evolved from experimental engines producing approximate results to production-ready sophisticated platforms providing correct executions of long-running jobs on hundreds of nodes even in face of failures [16]. State management in particular, enabled important advancements in fault tolerance and scalability by partitioning state and enforcing global coordinated checkpoints [15, 9]. 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 [8] and running cloud applications based on microservices and stateful functions [12, 2, 21]. 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, gender, and preferences.

The problem of querying the state of distributed streaming systems externally presents important challenges. Streaming systems perform continuous processing and thus their state also mutates all the time. Accessing the state in order to answer an external query entails some form of access synchronization. In addition, external access is bound to affect the processing progress of the system as operators of the streaming system compete with query processing for access to the state. Alternatively, the complexities of accessing live state can be avoided by leveraging the system's checkpointed state to answer external queries without obstructing the normal processing of the system. In this case, however, query results do not correspond to the system's current state. This is similar to directing database queries to replicas of primary nodes, which hold a stale version of the data. Surprisingly external queries to the state of streaming systems have received scant attention in the literature. Systems such as FlowDB [1] and TSpoon [24] are able to query the system's state. However, these systems are limited to querying the object directly by key and are thus not able to natively perform more complicated operations such as joining two states, or filtering by and aggregating the state values.

Having discussed the problem of querying state in stream processing systems, the following research question and subquestions are formulated:

RQ 1 How can state be exposed from stateful stream operators?

- RQ 1.1 How can it minimally impact the stream processing performance itself?
- RQ 1.2 How could this work in a distributed setting?
- RQ 1.3 What consistency guarantees are achievable when querying the state?

In this thesis, S-QUERY is proposed, a query interface to the distributed state of a stream processing system. S-QUERY exposes the internal distributed state of a streaming system through an external SQL query interface. S-QUERY is implemented in Hazelcast Jet [17], a distributed streaming system that optimizes low-latency performance. Our experimental evaluation suggests that snapshot configuration adds only up to 5ms in latency in the 99.99th percentile and no increase in 0-90th percentiles. In addition, S-QUERY scales horizontally, allowing for sustainable throughput to scale linearly with nodes in the cluster. Overall, this thesis makes the following contributions to the state of the art:

- design and implementation of S-QUERY, the first system that exposes an SQL-like interface for querying the state of distributed streaming systems, all while minimally impacting their throughput and latency.
- S-QUERY provides different isolation levels in a configurable manner, demonstrating the achievable isolation levels for consistent query results, including serializable isolation.
- thorough performance evaluation with queries of the NEXMark benchmark highlighting the tradeoffs between state size, checkpoint frequency, isolation levels, performance, and scalability.
- application of S-QUERY for real-time reflection of order delivery in Delivery Hero SE, a global company offering quick (Q) commerce services.

The structure of this thesis goes as follows. Section 2 describes a series of example use cases for queryable state in stream processing systems, next Section 3 provides preliminary background knowledge. Section 4 gives an overview of our approach to the problem with S-QUERY. In Section 5, the state structure used in S-QUERY is explained. Next, Section 6 demonstrates the impact of the state configuration on the isolation levels of S-QUERY. In Section 7 we describe S-QUERY's implementation. Section 8 shows a real world use case in Delivery Hero SE demonstrating the novel functionality of S-QUERY. The evaluation of S-QUERY follows in Section 9. In Section 10 several similar and related works are discussed and compared to S-QUERY. Finally, in Section 11 matters for future consideration are presented with concluding remarks in Section 12.

2 Why do we need Queryable State?

The capability of exposing and querying the distributed state of a streaming system, which is referred to as queryable state throughout the thesis, introduces novel and important advancements and use cases that are highlighted 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 sharply escalated 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 to applications faster.

In order to scale traditional relational database managemnent systems (RDBMS), caching is a quick to implement solution to provide improved 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. Keeping a cache consistent in a distributed setting, which 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 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 [14], 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 [14]. Thus, organizations using streaming systems need

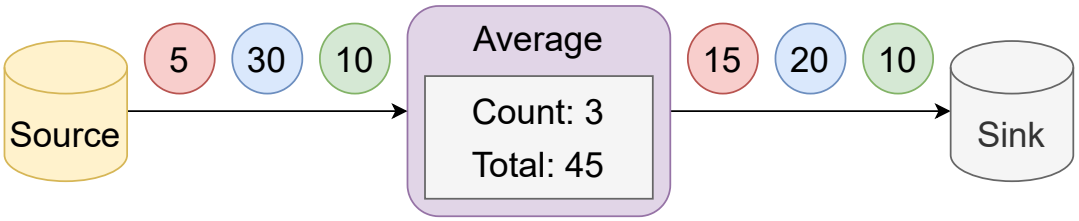


FIGURE 2.1: Stream processing pipeline with stateful 'average' operator. Input items are 10, 30, 5 respectively. Output corresponding to the input items have the same colors (and same order as in the input stage). Internal state of the operator is in the rectangle inside the operator.

to provide even their internal state on request. Therefore, the ability to query the internal state of a streaming system greatly facilitates regulation compliance.

Debugging. Currently, debugging stateful streaming topologies is a difficult process. With state being internal, a streaming system's input and output are the main observable clues. With queryable state however, it is possible to have an overview of this state or isolate a very specific 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 2.1. 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.

Having shown these various use-cases, it is clear that having queryable state is useful and in some cases even necessary.

3 Preliminaries

This section will lay out the required background knowledge for this thesis. Topics explained are the basics of stream processing including state management with key-value stores, and isolation levels in query systems.

3.1 Streaming Model

Stream processing queries and applications jobs are modeled as a directed acyclic graph (DAG) of operators, which is sometimes also called a dataflow graph. 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 vertex applying another operation. 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 operators across nodes when they are partitioned by key range. This is depicted in Figure 3.1, where the stream is partitioned according to a key allowing load balancing across the two nodes.

3.2 State Management & Fault Tolerance

Virtually all modern stream processing systems, including Apache Flink [7], Apache Spark [3], Jet [17], Apache Samza, Apache Pulsar, and IBM Streams [20] have converged to a common state management approach that guarantees at-least- or exactly-once processing based on the seminal Chandy-Lamport's distributed snapshots protocol [11] adapted to stream processing [9, 29]. In short, the most common state management approach involves periodic coordinated checkpoints (also called snapshots) that are taken when special markers that flow through the topology of a

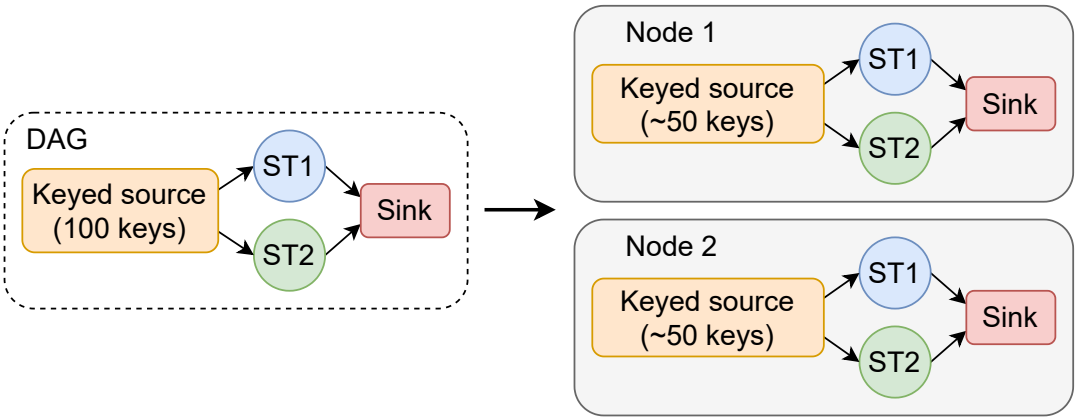
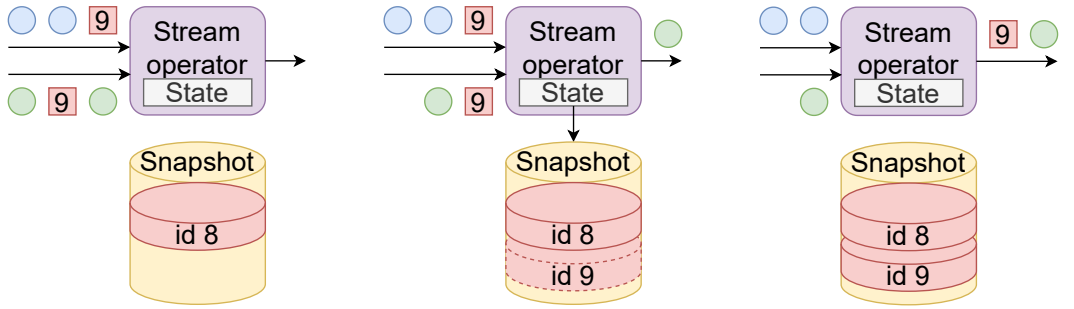


FIGURE 3.1: DAG distribution for two stateful transformations on a keyed stream with 100 unique keys



(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

FIGURE 3.2: Marker alignment phase on checkpoint for operator with two input channels. Red squares are markers, circles are records.

dataflow graph, instruct the operators to snapshot their state. The state 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 that the processing of each input record will be recorded to an operator's state exactly-once.

Figure 3.2 depicts this process. First, Figure 3.2a depicts an operator with two input channels is shown. 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 3.2b. 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 3.2c. S-QUERY takes advantage of such a checkpointing mechanism found in many streaming systems in order to make the state of the stream operator available for querying.

3.3 Key-Value Stores & Relation to Stream Processors

Modern key-value (KV) stores such as Cassandra[22] and DynamoDB[13] basically 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., hashing). They also keep replicas of each key value pair for fault tolerance and scalability. Finally, KV stores typically support a dialect of SQL, which allows external applications to query the KV store. Since S-QUERY stores its state in a KV store, that state is also immediately made available for querying. Section 5 details how this is done.

3.4 Isolation levels in distributed query systems

In database systems, ACID is used to describe the atomicity, consistency, isolation, and durability of database transactions [19]. The isolation level is an important part

TABLE 3.1: Phenomena possible in different isolation levels [5]. ✓ denotes possible, ✗ denotes not possible.

Isolation level \ Phenomena	dirty write	dirty Read	lost update	fuzzy read	phantom read	read skew	write skew
None	✓	✓	✓	✓	✓	✓	✓
Read uncommitted	✗	✓	✓	✓	✓	✓	✓
Read committed	✗	✗	✓	✓	✓	✓	✓
Repeatable read	✗	✗	✗	✗	✓	✗	✗
Snapshot	✗	✗	✗	✗	✓	✗	✓
Serializable	✗	✗	✗	✗	✗	✗	✗

of ACID, as it details how well transactions are visible to other transactions. The isolation level of a query system is determined by the possibility for certain phenomena: dirty writes, dirty reads, lost updates, fuzzy reads, phantom reads, and read/write skew [5]. These phenomena are used in the analysis as they improve upon the original ANSI SQL isolation levels by taking into account snapshots of data [5]. Below a short summary is provided of these phenomena, ordered from worst (*dirty write*) to least worst (*write skew*):

1. **Dirty write.** This occurs if a transaction T1 modifies an item, after this transaction T2 then also modifies the same item. Then if T1 or T2 performs a rollback, there is no clear correct value [5].

2. **Dirty read.** This phenomenon occurs when a transaction T1 reads a value while another transaction T2 rolls back that value to a previous version. T1 then has an out of date result which is called a dirty read [5].

3. **Lost update.** This happens when a transaction T1 reads a data item, T2 then updates it, and T1 updates it again based on its first read [5]. The effect is that the update by T2 is then lost by the overwrite of T1.

4. **Fuzzy read.** This occurs when a transaction T1 reads the same item twice, but gets a different result for both reads. This can be is caused by another transaction T2 modifying the item between the two reads of T1 [5].

5. **Phantom read.** This can occur when a transaction T1 reads a set of items according to a criterion while another transaction T2 also updates the items matching that criterion. If T1 queries the set of items twice, the size of this set can change because T2 changed the items matching this criterion [5].

6. **Read/write skew.** Read/write skew occurs when there are constraints between items. Take for example constraint C between items A and B. If T1 reads A, and T2 updates A and B to new values, when T1 reads B constraint C might be violated, this is *read skew*. Another issue arises when T1 reads both A and B, T2 does the same and writes A. If T1 then writes B, constraint C could be violated, this is *write skew* [5].

The isolation level is determined by the phenomena that can happen and checking the corresponding isolation level in Table 3.1.

4 Approach Overview

S-QUERY is an SQL interface 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 by configuration. Queries to snapshot state can refer, analyze, and compare specific versions of snapshot state. S-QUERY can be introduced to any streaming system employing a checkpoint-based state management approach.

4.1 Architecture

Figure 4.1 depicts the high level system 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, in-memory key-value store (or any partitioned database system, for that matter). Any change in the stateful operator state is directly reflected in the *state store* and updates the live state stored there. At the same time, the state store holds the snapshots that are triggered by the checkpointing mechanism [9, 29] 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 4.1, snapshot with id 9 is still being processed, so 8 is the latest snapshot id which corresponds to the latest completed checkpoint.

4.1.1 Colocating State & Compute

RQ 1.2 asks how it is possible to query state in a distributed system where state is fragmented across different nodes. This is solved in S-QUERY as follows: Figure 4.1 shows that both the live state of each operator as well as the respective checkpoints are colocated with the compute of the same partitioning. This is one of the main design decision behind 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 have to *i*) share the same partitioning function and *ii*) the system's scheduler needs to make sure that state and compute of the same partition are colocated.

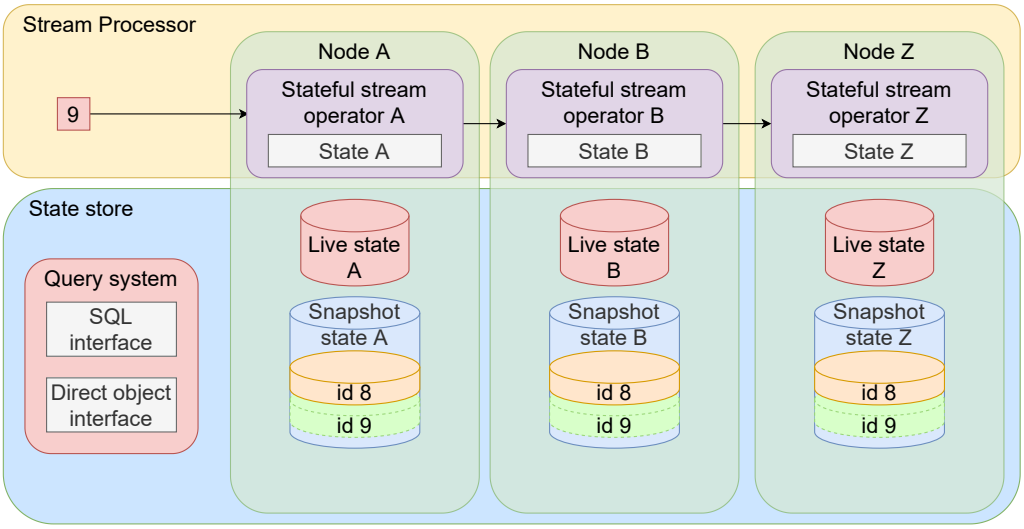


FIGURE 4.1: S-QUERY system architecture. Stream operators store state in state store which is in turn queried by the query system. Query system only queries snapshot 8 as 9 is still in progress (dashed outline=in progress).

5 Exposing Internal State to External Systems

This section describes how S-QUERY utilizes a KV store in order to make the state of a stream operator externally visible and queryable while minimally impacting the stream processing capabilities, to answer RQ 1.1.

5.1 Storing Operator State in a KV Store

The 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 4.1. This way, instead of performing remote calls for each change to the operator state (help 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 [23] or Raft [28]. 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.

5.2 Modeling & Storing State Externally

S-QUERY uses two tables per stateful vertex in the dataflow graph: one for live state, and one for snapshot state as depicted in Figure 5.1. It is important to note here that we assume that the state of an operator is in the form of a Map (e.g., a Java HashMap), 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 table is shown in Table 5.1. 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 table is named after the operator whose live state it holds. Finally, the name of a table is used to address SQL queries to the live state of the corresponding operator. As seen in Figure 5.1, the operator (vertex) is called `average` and that matches the name of the table. Note that if the `average` vertex has multiple instances i.e., it runs partitioned in multiple nodes, the table called `average` will contain all the KV pairs of the state of all these operators across the cluster.

TABLE 5.1: Live state structure

Key	Value
Key	State object

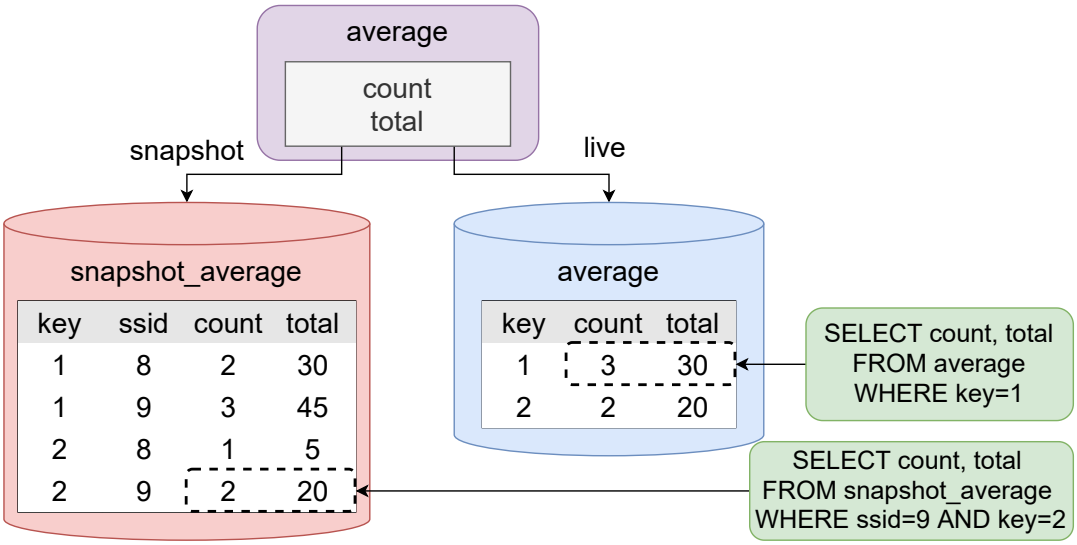


FIGURE 5.1: S-QUERY stream operator state representation for both live and snapshot state including queries.

TABLE 5.2: Snapshot state structure

Key		Value
Key	Snapshot ID	State object

Storing Snapshots. The way that snapshot state is stored, can be seen in Table 5.2. 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_<vertex name>`, where `<vertex name>` is the name of the stateful operator in the DAG for which the snapshot state is stored. For example, if the vertex 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 table, 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.

As discussed in Section 3, the checkpointing mechanism of most modern streaming systems follows a two-phase commit (2PC) snapshot process: once all operators have completed taking a snapshot of their state, they inform the snapshot manager and the snapshot manager directs the operators to commit their snapshots only if all operators have succeeded. After this snapshot 2PC process is done, S-QUERY adds an additional step: it updates a counter to reflect the latest committed snapshot ID. This latest snapshot ID is needed for identifying which is the latest completed snapshot across the distributed system.

5.3 Querying Live & Snapshot State

The tables 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 5.1, an SQL query can be used to query the state of running operator, or the state at the moment when snapshots have been taken at different moments in time, during the lifetime of the streaming job.

RQ 1.1 asks how queryable state can be achieved while minimally impacting the stream processing performance itself. This is achieved by S-QUERY as it designed to leverage the existing 2PC process to store the state in an easy to query form/location, therefore minimally impacting the stream processing performance itself when querying snapshot state. The next section describes the different isolation levels that can be achieved with S-QUERY.

6 Isolation levels

Since S-QUERY exposes the internal state of a stream processor to the outside world, this section discusses the isolation levels that one can achieve with S-QUERY to answer RQ 1.3. Since S-QUERY can query both the live state of a stream processing job as well as its snapshot state created and updated by periodic checkpoints. The two capabilities entail different performance characteristics and consistency guarantees.

6.1 Querying Live state

S-QUERY locks each entry of the table being accessed in order to safeguard from concurrent updates to the live state by the job being processed. The stateful operators use the same locking mechanism when they access and mutate the state.

Consider the case where the internal, live state of an operator is read before a checkpoint is taken. and that the operator is not deterministic [4, 29]. The replay after the failure could yield different results. This is exactly the dirty read phenomenon, which according to Table 3.1 results in the read uncommitted isolation level.

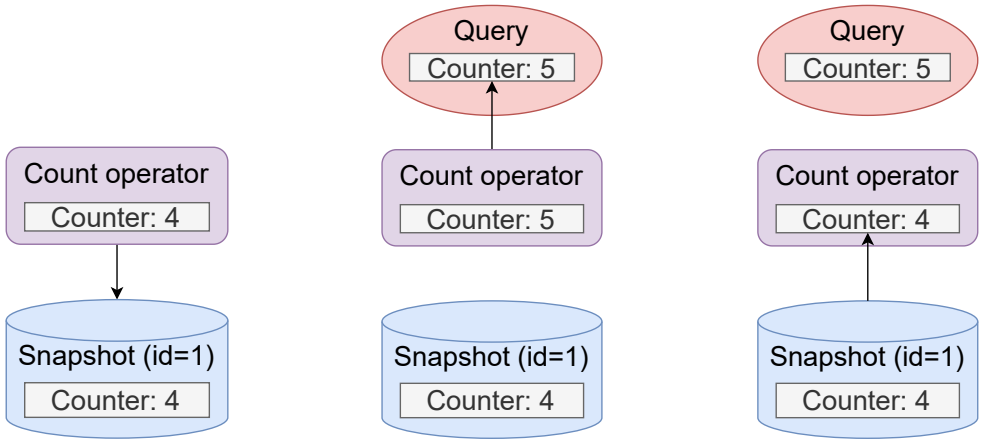
Thus, live state queries provide *read uncommitted* isolation level because of the possibility of dirty reads. Take for instance the following example where the state of a stream operator that counts incoming records is 4. At that point a checkpoint is taken that creates snapshot with id 1 (Figure 6.1a). Following the checkpoint the state becomes 5. Then, a query on the live state returns 5 (Figure 6.1b). 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 operator recovered its state from the latest snapshot (Figure 6.1c). Therefore, this example demonstrates the possibility of a dirty read.

6.2 Querying Snapshot State

Querying the live state tampers with the processing of a job because it requires access synchronization. Alternatively, queries can be answered by accessing the snapshot state of a job, which offers safe unrestricted data access. However, the data in the snapshot state are stale as the job's live state continues to be modified.

The only pitfall regards which snapshot is being queried. If a query to snapshot state is being processed and at that same time a new snapshot is committed, which takes the place of the previous one, then phantom reads are possible. For this reason S-QUERY keeps track of the snapshot id in the form of a distributed atomic integer value and always reads it before executing a query on the snapshot state. This way, phantom reads are impossible because the query is guaranteed to be processed on the latest available snapshot atomically.

In addition, read and write skew are not possible in this situation as there is exactly one partition responsible for writing the state, meaning multiple partitions never write to the same location.

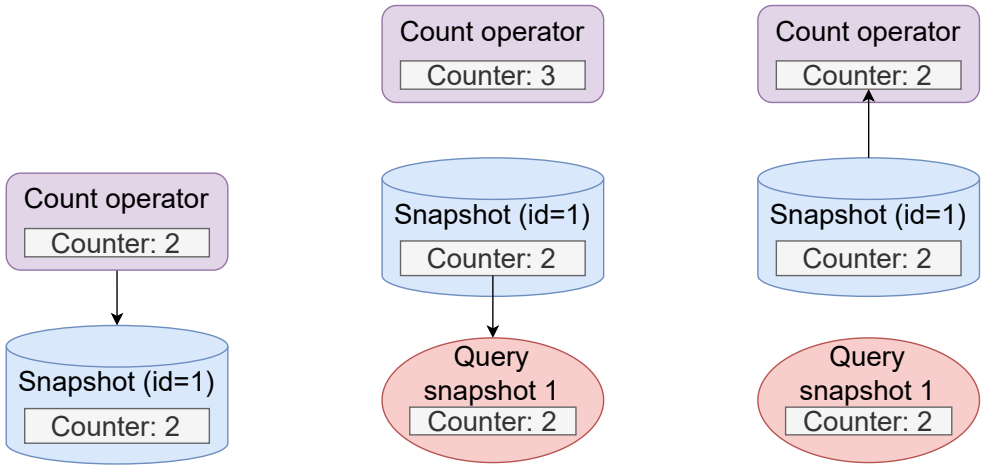


(A) Initial state is 4 and snapshot is made with id 1

(B) Live state is updated to 5, query result is 5

(C) Snapshot 1 restored, query result is a dirty read

FIGURE 6.1: Live state isolation level example.



(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

FIGURE 6.2: Snapshot state isolation level example.

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 6.2a). 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 6.2b). 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 6.2c). Dirty reads are never possible in this setting and phantom reads can only happen if no latest snapshot ID is specified in the query.

As no phenomena as described in Section 3.4 can occur, under this scheme, S-QUERY provides *serializable* isolation.

7 Implementation

This section will explain how S-QUERY was implemented, what implementation choices were made and why.

7.1 Platform choice

The decision was made to implement S-QUERY on top of Jet in order to leverage Jet's state management approach, its neat integration with the IMDG, and Hazelcast's SQL interface to IMDG. Since Jet implements Chandy-Lamport-style coordinated checkpoints, snapshot state captures a consistent distributed snapshot of the system's overall state. Jet leverages the IMDG as a fast store for state snapshots. In the rest of the thesis, original Jet is used to refer to the unmodified version of Jet, and S-QUERY to signify the system developed as a result of this thesis. In our implementation IMDG's IMap shares the same partitioning scheme as the keyed state of Jet in order to hold state snapshots on the same node as the corresponding keyed state. This is an important property as the state does not need to travel across the network when writing snapshots or live state.

7.2 State structure

Original Jet used to store the state of each operator in a single key-value pair in IMDG, where the key was metadata and the value a blob structure encapsulating the key-value pairs of the operator's state (see Table 7.1). As a result, this structure was not queryable. In contrast, S-QUERY exposes the state of each operator stored in the KV structure depicted in Table 5.1 and Table 5.2 as first-class key-value pairs (see Section 5.2), allowing each operator to be queried externally.

7.3 Fault-tolerance

To protect data in IMDG from failures, it implements an Available under network Partition (AP) mechanism. While the state is kept locally as the primary source, Jet also keeps one backup replica on a different node for fault-tolerance. This is shown in Figure 7.1, where the state is partitioned across a 3 node cluster. In Figure 7.2 node 1 has failed and node 2 and 3 rebalance the state across the new cluster to

TABLE 7.1: IMap state structure in Jet

IMap Key				IMap Value
partition N	snapshot ID	vertex name	sequence number	128KB array of key-state pairs in partition N

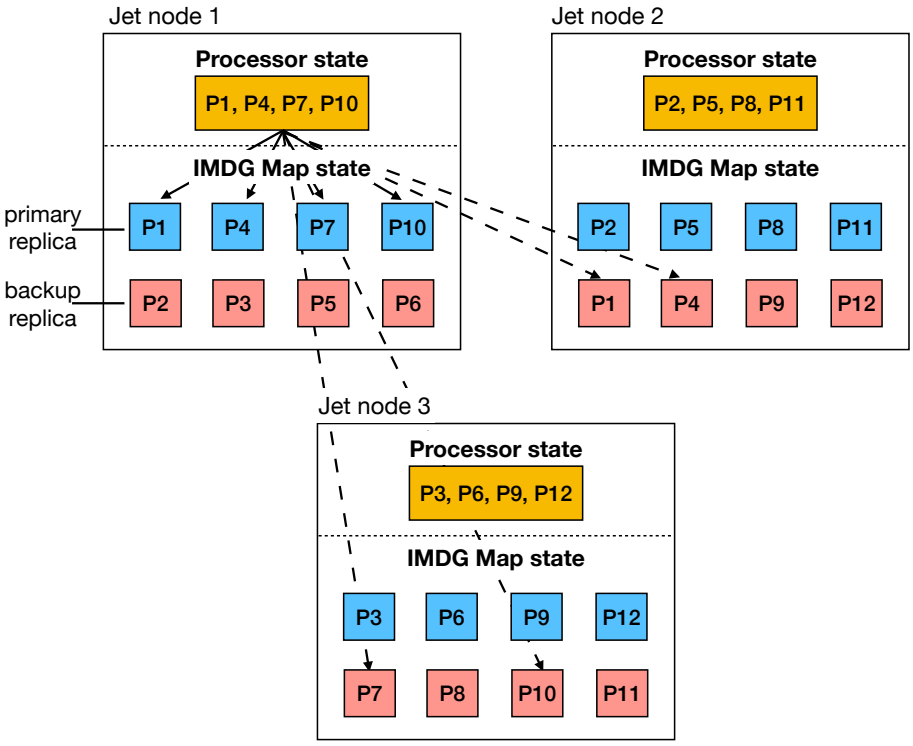


FIGURE 7.1: Jet state partition structure across a 3 node cluster [17]

keep availability at 100%. In addition, a Consistency under Network partitions (CP) subsystem using Raft [28] is used. This subsystem preserves strongly consistent distributed data structures across Jet nodes such as the atomic integer containing the latest snapshot id (see Section 5.3).

7.4 Query system

The query system is the gateway between the S-QUERY and external systems that want to inspect the stream processor state.

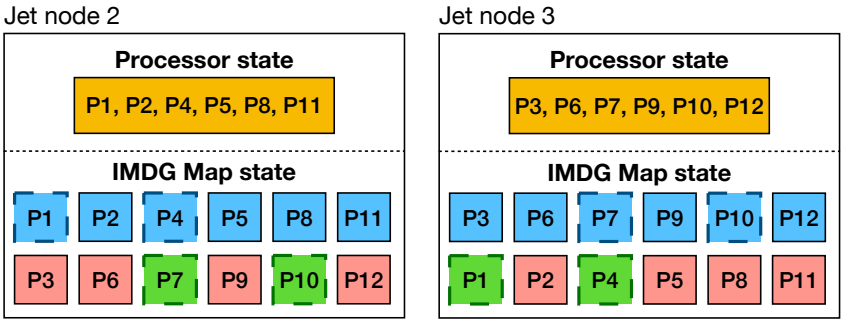


FIGURE 7.2: Jet state partition rebalancing [17]. Dashed outline are the states from (failed) node 1.

7.4.1 Direct object interface

In S-QUERY, this gateway is facilitated by using the Hazelcast client libraries. Hazelcast has client libraries for several programming languages that can interact with a Hazelcast cluster (which runs Jet). These libraries can access the IMDG which store the state. This is done using operations such as `get(key)` or `get(predicate)`, resulting in state entry objects native to the programming language of the client library. As this method uses native objects, this is called the direct object interface. As the state is accessible in a custom program, there are virtually endless possibilities in what one can do with the state, such as using it as a backend for a visualizing website, a reporting tool, etc.

7.4.2 SQL interface

Hazelcast IMDG features an SQL interface, which allows applications to query IMap data structures in IMDG. The SQL interface supports a subset of SQL, such as the WHERE, ORDER BY, SUM, and COUNT clauses[1]. Essentially, querying the state becomes as simple as querying a traditional relational database system. The JOIN functionality is not officially released yet, but it was already available in a development branch.[2] Unfortunately, this branch was not compatible even with original Jet, so custom modifications were made in S-QUERY Jet to make them compatible.[3,4]

[1] https://docs.hazelcast.com/imdg/4.2/sql/expressions.html
[2] https://github.com/hazelcast/hazelcast/tree/sql
[3] https://github.com/Jimver/hazelcast/tree/sql-jet-4.4
[4] https://github.com/Jimver/hazelcast-jet/tree/4.4-statefulP

8 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 utmost importance. Therefore, databases hosting daily data about points of sale, 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 8.1.

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 [10] or Memcache [26]. 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 8.2. In this thesis 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. Four real queries are used to evaluate the expressiveness and performance of S-QUERY, Query 1, Query 2, Query 3 and Query 4 with (partial) results in Table 8.1 and Table 8.2. 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.

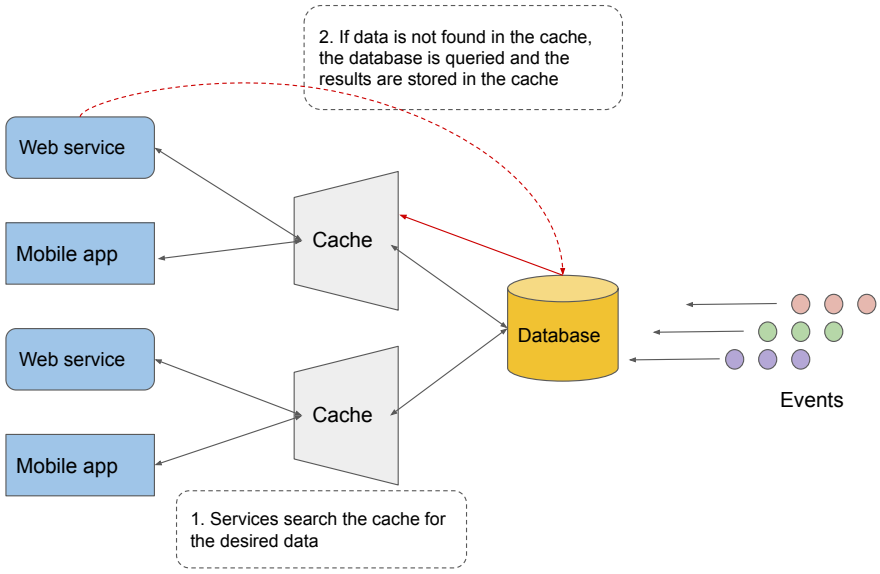


FIGURE 8.1: Traditional caching architecture

```
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?

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, vendor category and delivery zone (postal code range).

Details of the performance of this workload can be found in the evaluation (Section 9).

```
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?

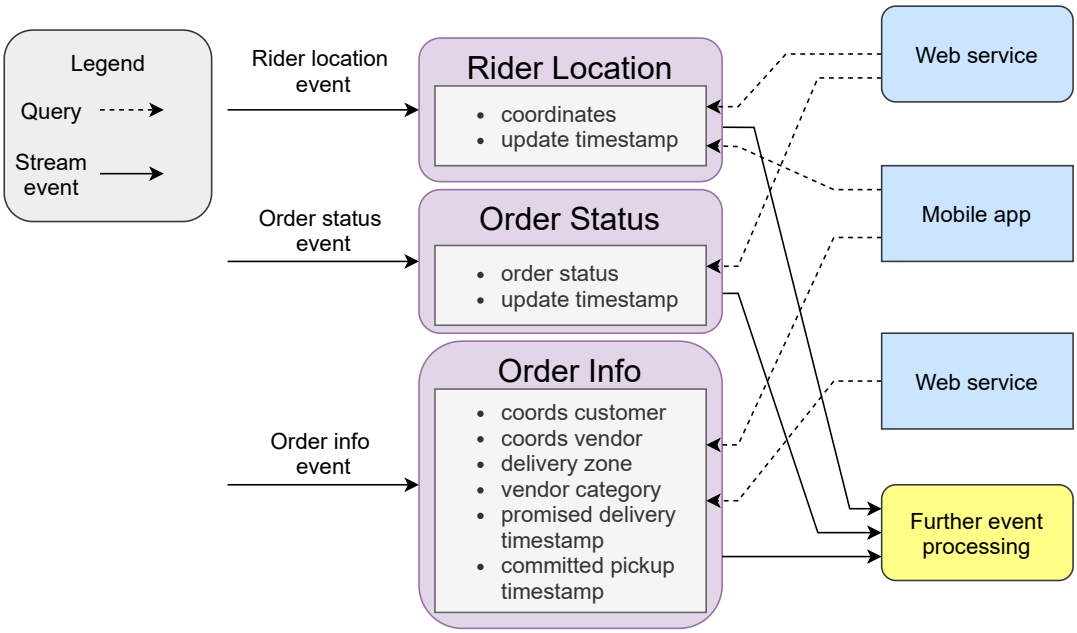


FIGURE 8.2: Stream-based architecture, events are processed by stateful operators with corresponding state inside. External systems are able to query the internal state.

```
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 8.1: SQL query results for Query 1 and Query 2

(A) Order late in preparation per area (Query 1)

Count(*)	deliveryZone
121	9700-9747
113	8200-8245
132	9400-9408
134	6500-6546
118	3500-3585
118	4800-4839
113	5600-5658
137	4200-4208
120	2600-2629
⋮	⋮

(B) Orders ready for pickup per shop category (Query 2)

Count(*)	vendorCategory
3630	lunch
3631	sushi
3553	groceries
3675	pizza
3647	breakfast

TABLE 8.2: SQL query results for Query 3 and Query 4

(A) Orders being prepared per area (Query 3)

Count(*)	deliveryZone
484	6200-6229
495	7500-7547
475	8900-8941
513	1000-1099
483	8200-8245
527	9700-9747
516	9400-9408
517	6500-6546
521	3500-3585
⋮	⋮

(B) Orders in transit per area (Query 4)

Count(*)	deliveryZone
1588	6200-6229
1510	7500-7547
1525	8900-8941
1534	1000-1099
1496	8200-8245
1518	9700-9747
1565	9400-9408
1527	6500-6546
1542	3500-3585
⋮	⋮

9 Evaluation

S-QUERY is evaluated on a) a real workload capturing real-time views of online Q-commerce orders with deliveries provided by Delivery Hero SE and b) NEXMark [30], the de facto benchmark in stream processing. The focus of the evaluation lies on four different dimensions of S-QUERY's operation. First, we measure S-QUERY's overhead to Jet in terms of latency (Section 9.2. Second, we analyze the overhead of S-QUERY to Jet's snapshot mechanism with and without query execution (Section 9.3). 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.[1] We also compare S-QUERY's query performance against TSpoon [24] (Section 9.4). Finally, we study the scalability of S-QUERY by measuring the system's throughput with different cluster sizes (Section 9.5).

9.1 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 are detailed in Table 9.1. 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 [17]. In this setup, the 4 CPUs used for garbage collection are also used to process S-QUERY queries.

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.[2,3] 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.

TABLE 9.1: 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)

[1] https://github.com/Jimver/S-Query-examples
[2] https://beam.apache.org/documentation/sdks/java/testing/nexmark/
[3] https://github.com/hazelcast/big-data-benchmark

The snapshot performance experiments are performed on the Delivery Hero SE workload (Section 8) 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 totaling 48 vCPUs to compare to related work, TSpoon [24].

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, the 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.

9.2 Overhead Experiments

The experiment results for query 6 of NEXMark are shown in Figure 9.1. 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) original 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 for the snapshot state configuration is equivalent to the original Jet configuration. Therefore, for the remaining experiments, we focus our evaluation on the snapshot state configuration.

Figure 9.2 compares S-QUERY's snapshot state configuration to original 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 original Jet until around the 90^{th} 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 5ms more latency. In conclusion, S-QUERY achieves similar low latency performance as original Jet demonstrating that the state snapshot configuration of S-QUERY has very little impact on the system's latency and throughput.

9.3 Effect of Snapshotting Mechanism on System Performance

Because S-QUERY introduces changes to the snapshot creation process of original 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 original Jet considering different

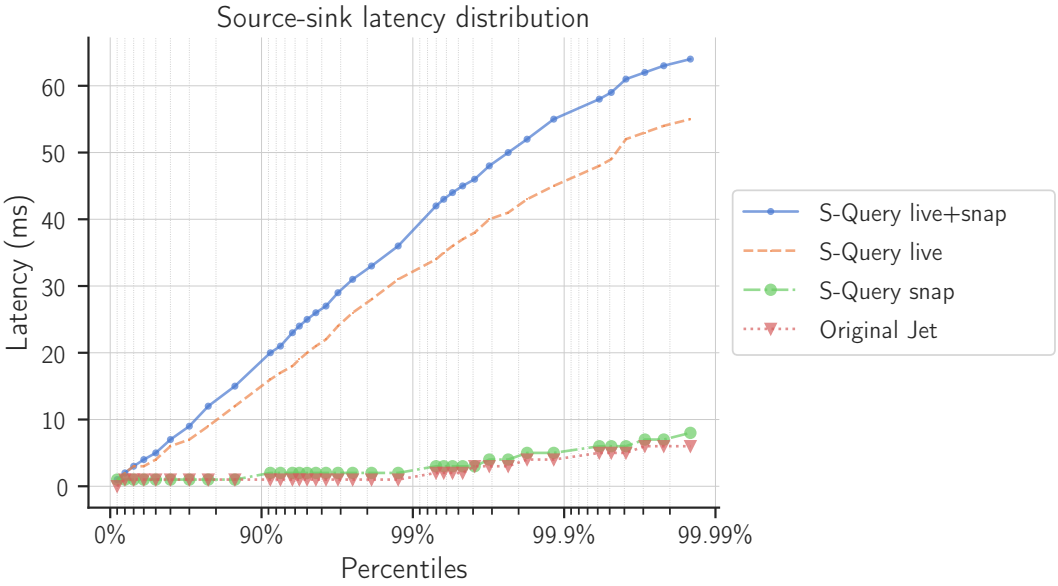


FIGURE 9.1: NEXMark query 6 latency distribution, 3-node cluster. x-axis shows percentiles on an inverted log scale, y-axis shows the latency in milliseconds.

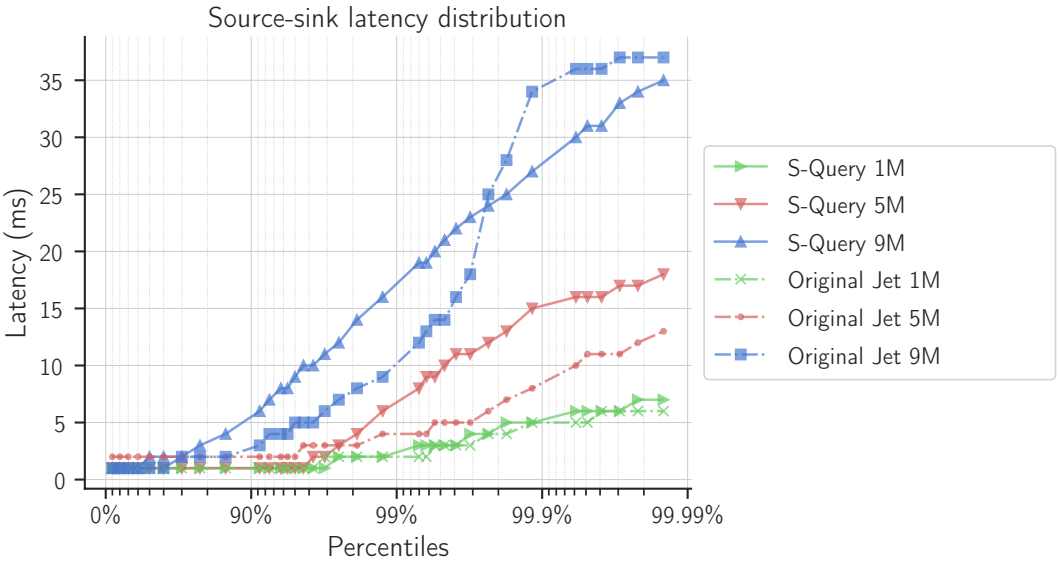


FIGURE 9.2: NEXMark query 6 latency distribution. Vanilla vs. S-QUERY, 3-node cluster at 1M, 5M, 9M events/s. x-axis shows percentiles on an inverted log scale, y-axis shows the latency in milliseconds.

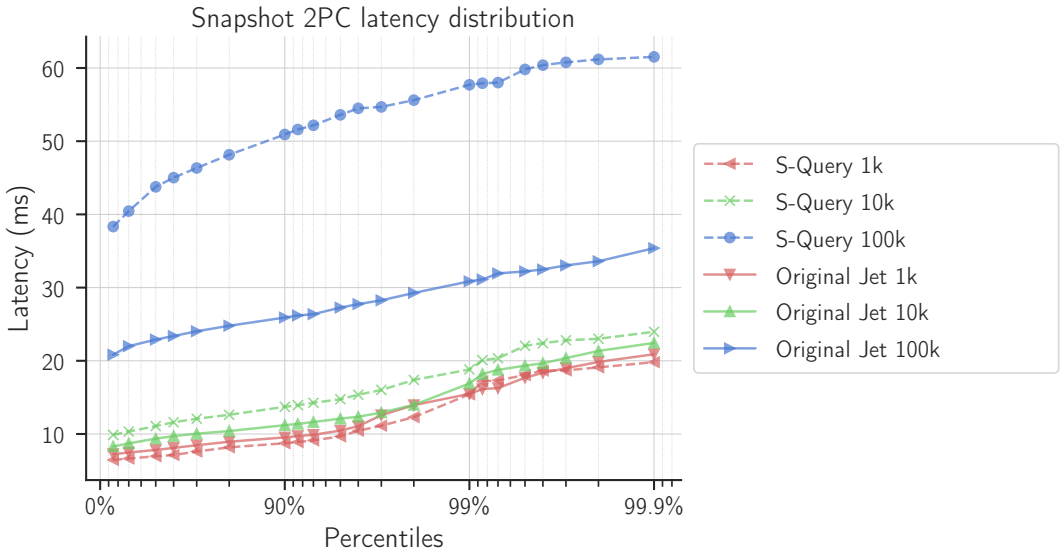


FIGURE 9.3: Latency distribution of snapshot mechanism between original Jet and S-QUERY for 1K/10K/100K unique keys across a 7 node cluster. x-axis shows the percentiles on an inverted log scale, y-axis shows the latency in milliseconds.

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 9.3 shows the snapshot creation time for both original Jet and S-QUERY ranging the number of unique keys in the snapshot state. S-QUERY achieves virtually equal latency performance to original Jet for 1K keys and is only 2-4ms slower than original Jet for 10K keys throughout the distribution. Even for the 100k keys of state, S-QUERY is merely 19-27ms slower than original Jet. While the difference might seem considerable, the overhead would be unnoticeable to most applications. To further illustrate the impact, the snapshot interval s_i is set at 1 second, which is already very low. The 50^{th} percentile of the worst case (100K keys), has a snapshot latency s_l of 44ms for S-QUERY and 23ms for original Jet. The ratio r between time available for processing and total time (processing + snapshot) is:

$$r = (s_i/(s_i + s_l))$$

Then $r_{\text{S-QUERY}} = 1/1.044 \approx 0.958$ and $r_{vanilla} = 1/1.023 \approx 0.978$. Comparing these two, S-QUERY would only process $1 - r_{\text{S-QUERY}}/r_{vanilla} \approx 0,02011 = 2.01\%$ less than original Jet. The implication is that in theory the sustainable throughput would be 2.01% lower for S-QUERY than original Jet, which is still manageable, as this is for the worst case (100k unique keys, 1 second snapshot interval). For smaller number of keys the difference becomes almost negligible: 0.25% for 10K and close to 0 for 1K. Consequently, S-QUERY's impact on the performance of the streaming system is minimal.

Query Execution. The execution of S-QUERY queries can potentially affect the snapshot creation mechanism. The latency distribution of the snapshot mechanism is measured as before with and without queries being executed on the snapshot state. For the experiments Query 1 is used, which is a relatively expensive query including both a JOIN and GROUP BY clause. Figure 9.4 shows the impact of queries on the snapshot 2PC latency. For 1K and 10K keys the impact is zero until the 70^{th} percentile. For 10K a small difference appears from 80% onwards, which increases

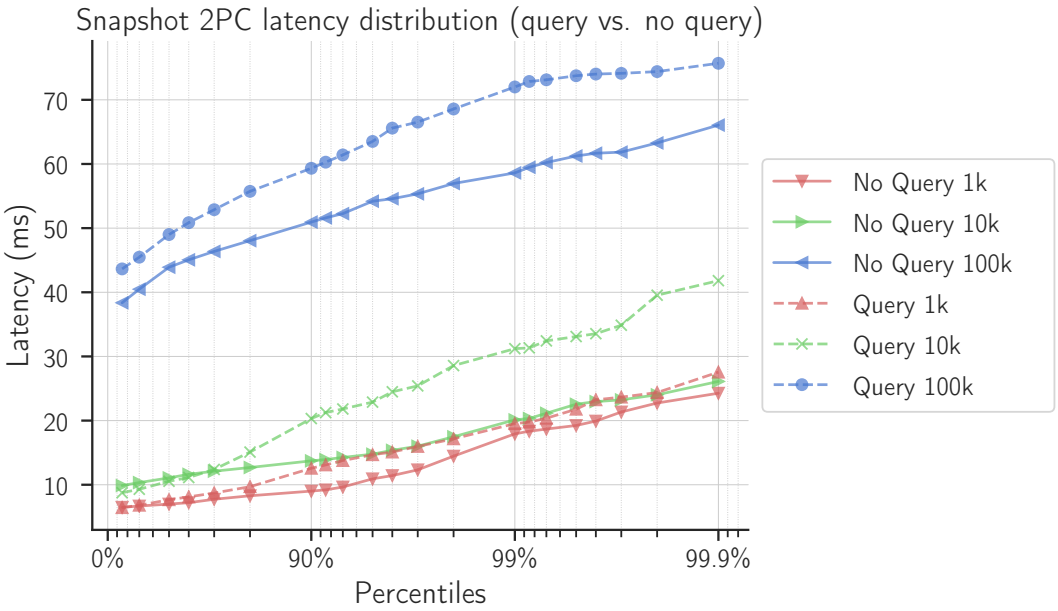


FIGURE 9.4: Latency distribution of snapshot mechanism on 7 node cluster with and without S-QUERY queries for different # of unique keys. x-axis shows percentiles on an inverted log scale, y-axis shows the latency in milliseconds.

up to 16ms, while for 100k there is a similar difference throughout the entire distribution. In general, the impact is no more than 16ms across all unique key configurations. Notably, the time it takes to commit a snapshot is much smaller than the snapshot interval itself. Thus, querying non-stop from multiple threads does not affect the performance of a streaming job significantly.

9.4 Query Performance

The query execution performance of S-QUERY is measured through its SQL interface and through its direct object interface.

SQL Interface. The performance is measured in terms of latency by the time it takes to execute a query on the state and the time required to get the latest snapshot ID across the distributed system, results are shown in Figure 9.5. This experiment uses Query 1 as the SQL query. As expected, the query execution latency increases with larger state size as a query has to process more state entries. However, the latency distributions remain relatively flat even in the higher percentiles. The snapshot ID times slightly increase as the state size increases, but this is owed to the increased system load that is related to the larger state size. The performance of the queries is decent given that these latencies represent the time between the moment before query submission and complete 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 in the best case to execute the query without taking into account network overhead. Notably, a relatively flat latency distribution provides the opportunity for reliable SLA agreements.

Direct Object Interface. In this experiment the performance of S-QUERY is compared to TSpoon [24], another stream processing system offering simple state queries. The rider location operator from the Delivery Hero SE use case from Section 8 is used as the query target, whose state consists of two doubles (coordinates)

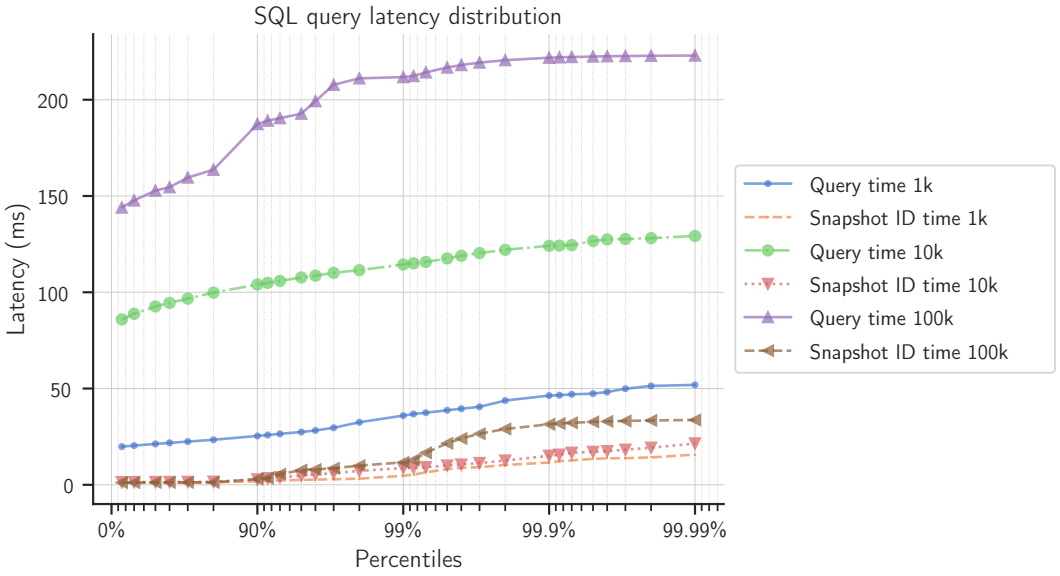


FIGURE 9.5: Query/snapshot ID latencies on 7 node cluster for different # of unique keys. x-axis shows percentiles on inverted log scale, y-axis shows latency in milliseconds.

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 totaling 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 one double (account balance) as state in their experiment [24], which was conducted on a cluster with 50 vCPUs. Figure 9.6 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 same plot also shows 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. A power law distribution is reasonable since selecting more keys takes more time thereby resulting in less throughput. In addition, throughput will never become zero as long as the queries finish, which explains the long tail being non zero, and justifies the power law distribution. Compared to the equivalent experiment from TSpoon [24], 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.

9.5 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. Different experiment configurations are defined by varying the degree of parallelism in terms of number of nodes in the cluster and the snapshot interval. The snapshot interval affects the amount of time available for processing, which can have an effect on S-QUERY's scalability. For the experiment, query 6 of the NEXMark benchmark is

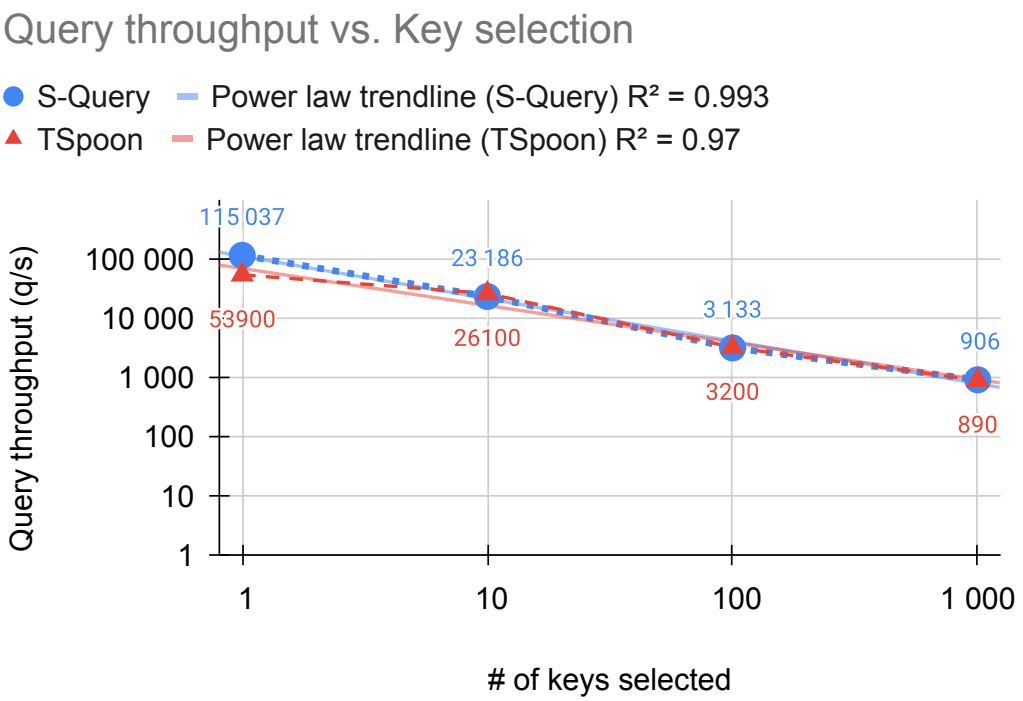


FIGURE 9.6: Direct object query throughput. Throughput in queries/s on the log scale y-axis, number of keys selected in the query shown on the log scale x-axis. Data labels are S-QUERY on top, TSpoon on the bottom.

used as the streaming job and execute 10 JOIN queries per second on the state of the job's operators. This setting simulates a realistic workload.

The results of the scalability experiment are shown in Figure 9.7. The existing horizontal scalability of Jet [17] is 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 the snapshot interval is not the interval between the start times of two consecutive snapshots, but the interval between the end of the current snapshot and the start of the next one. Therefore, having 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.

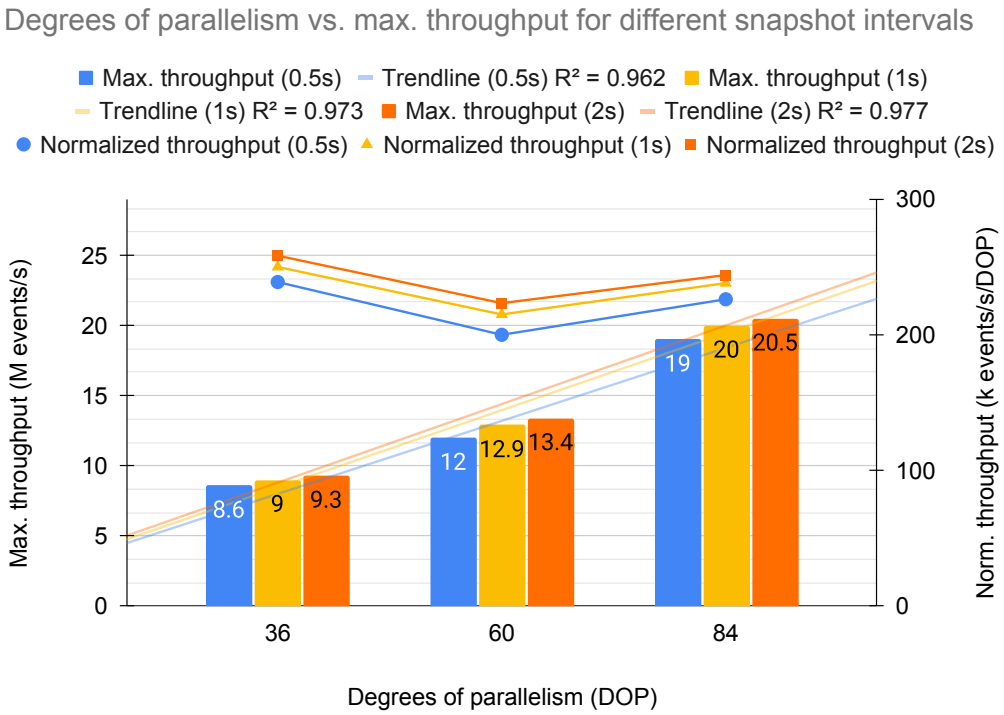


FIGURE 9.7: Scalability experiment results, DOP (degrees of parallelism) on x-axis, max. throughput on left y-axis, normalized throughput on right y-axis. Bars from left to right correspond to 0.5s, 1s, and 2s snapshot intervals respectively.

10 Related Work

Existing pieces of work focus on state access across operators within a stream processing system, often referred to as transactional stream processing [6, 25, 31, 1, 24, 18], while others support state queries from external sources [7, 27, 1, 24, 18]. We compare and contrast these pieces of work to S-QUERY.

10.1 Transactional Stream Processing

The Unified Transaction Model (UTM) [6] is one of the first systems supporting transactions in a stateful stream processing system. Using a transaction manager with a strong strict 2-phase locking mechanism, it provides in-order access to storage resources and rollback functionality, resulting in consistent histories. S-Store [25] supports transactional stream processing with shared state access inside the system. S-Store and S-QUERY both use an in-memory data structure for storing the state. Finally, TStream [31] improves upon S-Store by dynamically restructuring transactions into operation chains, which can be processed in parallel eliminating lock contention [31]. S-QUERY differs from transactional streaming systems in that it focuses on external queries to the state instead of transactions on the state inside the system.

10.2 Queryable State

Apache Flink [7] provides a simple API for state queries. Flink uses a client(proxy)-server model where the client sends a query to the client proxy, which in turn forwards it to the correct state server where it gets processed and the result is returned. Flink only features a programmatic interface for queries. In addition, it requires to specify data types upfront[1], which is not the case with S-QUERY. Furthermore, queries in Flink always return the live state, so as discussed in Section 6.1, this allows for dirty reads when the system recovers from a failure. Consequently, Flink only supports the read-uncommitted isolation level when querying state.

Apache Samza [27] allows streaming jobs to access the state of other streaming jobs using the Table API, which supports both an in-memory store, RocksDB, or other custom remote tables.[2] Remote tables would allow external applications to query the state, but the state would need to be transported over the network, incurring a heavy performance penalty compared to in-memory state. Queries are key-value lookups.

FlowDB [1] supports queries to the internal state from external components. It uses non-exclusive locks to read the live state of multiple operators. However, FlowDB only works with live state, thus it supports read-uncommitted isolation level at best, when considering failures where the state needs to be recovered from checkpoints. In addition, the queries are simple key lookup operations.

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/datastream/fault-tolerance/queryable_state/

[2] https://samza.apache.org/learn/documentation/1.6.0/api/table-api.html

PipeFabric [18] supports ad-hoc queries on the state by external parties. PipeFabric uses an MVCC-based state management system combined with commit timestamps to write and read the state thereby achieving snapshot isolation. However, the query system only allows key-value lookups on the state. In addition, it is only suitable for single node applications.

TSpoon [24] focuses more on transactions between operators than external queries. TSpoon requires queries to be programmed in advance as it is bound to each stateful operator. Thus it cannot support ad-hoc queries. In addition, a query is a get operation for the whole state of a specific key. There are no built-in predicates to filter results or join multiple results together, compared to S-QUERY.

S-QUERY is the first system offering an SQL interface and two different query modes with complementary properties and isolation levels.

11 Future work

There are still opportunities for optimization in the S-QUERY implementation in Jet. As discussed in Section 9.2, the live state configuration of S-QUERY is not viable for high throughput streams as putting the state to the IMap every time the state changes causes too much overhead. Therefore, a big opportunity for improvement lies in that area.

A similar but less severe performance point is the increased snapshot latency. While it doesn't have a significant impact on the stream processing operation, it is still desirable to have no impact or make it even faster. The cause of the increased snapshot latency is the increased amount of entries being put to the IMap. Even though the IMap is very fast, it still has some overhead causing the snapshots to be slower compared to the bunched up key strategy in original Jet. This could be solved by having a dedicated and efficient `putAllSnapshot()` implementation on the IMap without unneeded functionality such as data validation, as it is an internal method not available to end users. This was partly implemented in S-QUERY but could be further improved upon.

Another small issue is that querying the latest snapshot state now requires first getting the latest snapshot ID and then performing the query, however this is not ideal as the extra request for the latest snapshot ID incurs some extra latency. Therefore, having the SQL engine automatically retrieving this ID locally on the node could eliminate this extra step.

As a final recommendation for future research, it is deemed useful to implement S-QUERY on top of other stream processing systems, as it has now only been implemented on top of Jet. Such further implementations would allow for interesting comparisons on the performance characteristics of S-QUERY between different stream processing systems.

12 Conclusions

This thesis presents S-QUERY, a novel system supporting SQL 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, is horizontally scalable, and is able to perform thousands to tens of 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.

Bibliography

[1] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. "FlowDB: Integrating Stream Processing and Consistent State Management". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*. DEBS '17. Barcelona, Spain: Association for Computing Machinery, 2017, 134–145. ISBN: 9781450350655. DOI: 10.1145/3093742.3093929. URL: https://doi.org/10.1145/3093742.3093929.

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. "Stateful Functions as a Service in Action". In: *Proc. VLDB Endow.* 12.12 (Aug. 2019), 1890–1893. ISSN: 2150-8097. DOI: 10.14778/3352063.3352092. URL: https://doi.org/10.14778/3352063.3352092.

[3] Michael Armbrust et al. "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 601–613. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190664. URL: http://doi.acm.org/10.1145/3183713.3190664.

[4] Magdalena Balazinska et al. "Fault-tolerance in the Borealis Distributed Stream Processing System". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 13–24. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066160. URL: http://doi.acm.org/10.1145/1066157.1066160.

[5] Hal Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: Association for Computing Machinery, 1995, 1–10. ISBN: 0897917316. DOI: 10.1145/223784.223785. URL: https://doi.org/10.1145/223784.223785.

[6] Irina Botan et al. "Transactional stream processing". In: *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*. Ed. by Elke A. Rundensteiner et al. ACM, 2012, pp. 204–215. DOI: 10.1145/2247596.2247622. URL: https://doi.org/10.1145/2247596.2247622.

[7] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38. URL: http://sites.computer.org/debull/A15dec/p28.pdf.

[8] Paris Carbone et al. "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*. Ed. by David Maier et al. ACM, 2020, pp. 2651–2658. ISBN: 9781450367356. DOI: 10.1145/3318464.3383131. URL: https://doi.org/10.1145/3318464.3383131.

[9] Paris Carbone et al. "State Management in Apache Flink&Reg;: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729. ISSN: 2150-8097. DOI: 10.14778/3137765.3137777. URL: https://doi.org/10.14778/3137765.3137777.

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

[11] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456. URL: https://doi.org/10.1145/214451.214456.

[12] Martijn De Heus et al. "Distributed transactions on serverless stateful functions". In: *DEBS*. 2021.

[13] Giuseppe DeCandia et al. "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[14] 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)". In: *Official Journal L119* 59 (May 2016), pp. 1–88. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC.

[15] Raul Castro Fernandez et al. "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*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 725–736. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465282. URL: http://doi.acm.org/10.1145/2463676.2465282.

[16] Marios Fragkoulis et al. *A Survey on the Evolution of Stream Processing Systems*. 2020. arXiv: 2008.00842. URL: https://arxiv.org/abs/2008.00842.

[17] Can Gencer et al. "Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile". In: *Proceedings of the VLDB Endowment* 14.12 (2021).

[18] Philipp Götze and Kai-Uwe 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*. Ed. by Melanie Herschel et al. OpenProceedings.org, 2019, pp. 650–653. DOI: 10.5441/002/edbt.2019.78. URL: https://doi.org/10.5441/002/edbt.2019.78.

[19] Theo Härder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[20] Gabriela Jacques-Silva et al. "Consistent Regions: Guaranteed Tuple Processing in IBM Streams". In: *Proc. VLDB Endow.* 9.13 (Sept. 2016), pp. 1341–1352. ISSN: 2150-8097. DOI: 10.14778/3007263.3007272. URL: https://doi.org/10.14778/3007263.3007272.

[21] Asterios Katsifodimos and Marios 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*. Ed. by Melanie Herschel et al. OpenProceedings.org, 2019, pp. 682–685. DOI: 10.5441/002/edbt.2019.86. URL: https://doi.org/10.5441/002/edbt.2019.86.

[22] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[23] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[24] Alessandro Margara, Lorenzo Affetti, and Gianpaolo Cugola. "TSpoon: Transactions on a stream processor". In: *Journal of Parallel and Distributed Computing* 140 (2020), pp. 65–79. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.03.003. URL: https://doi.org/10.1016/j.jpdc.2020.03.003.

[25] John Meehan et al. "S-Store: Streaming Meets Transaction Processing". In: *Proc. VLDB Endow.* 8.13 (Sept. 2015), 2134–2145. ISSN: 2150-8097. DOI: 10.14778/2831360.2831367. URL: https://doi.org/10.14778/2831360.2831367.

[26] Rajesh Nishtala et al. "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*. Ed. by Nick Feamster and Jeffrey C. Mogul. USENIX Association, 2013, pp. 385–398. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[27] Shadi A. Noghabi et al. "Samza: Stateful Scalable Stream Processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1634–1645. ISSN: 2150-8097. DOI: 10.14778/3137765.3137770. URL: https://doi.org/10.14778/3137765.3137770.

[28] Diego Ongaro and John 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*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[29] Pedro F Silvestre et al. "Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows". In: *SIGMOD*. 2021.

[30] P Tucker et al. *NEXMark—A Benchmark for Queries over Data Streams*. Tech. rep. Technical report, OGI School of Science & Engineering at OHSU, 2002.

[31] Shuhao Zhang et al. "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. DOI: 10.1109/ICDE48307.2020.00136. URL: https://doi.org/10.1109/ICDE48307.2020.00136.