DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

# Global State Queries in Stream Processing

*Author:*
Mitali PATIL

*Supervisor:*
Asterios KATSIFODIMOS

*Daily Co-Supervisors:*
George CHRISTODOULOU
Kyriakos PSARAKIS

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

*in the*

Web Information Systems Group
Software Technology

July 17, 2025

# Declaration of Authorship

I, Mitali PATIL, declare that this thesis titled, "Global State Queries in Stream Processing" 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.

Signed: Mitali Patil

Date: 16 th July 2025

DELFT UNIVERSITY OF TECHNOLOGY

# *Abstract*

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

**Global State Queries in Stream Processing**

by Mitali PATIL

While database systems have matured significantly over the past few decades, the rapid growth of real-time analytics to feed quick decision making has paved a way for multipurpose and high performant systems. As stream processing also matures, it is of interest to explore its full functional capabilities such as state management. Most streaming systems have inaccessible state for external systems to query, which limits the ability to drive value from the live mutable state data. In this thesis we present Q-Styx, a system that exposes the live state of stateful operators in a streaming engine for external queries. We introduce a global state store that maintains a copy of the distributed state across the system without the need of an external database. With strong isolation guarantees for consistent results, our implementation balances the tradeoffs between performance isolation and data freshness while exhibiting minimal impact on the core transactional capabilities of the streaming engine.

# *Acknowledgements*

I would like to firstly thank my supervisors Dr. Asterios Katsifodimos, George Christodoulou and Kyriakos Psarakis for their huge help, support and great ideas during my thesis. I especially acknowledge their feedback and invaluable patience.

I am also grateful for all my friends in Delft who made this place feel like home and inspired me. And lastly, to my family for their unconditional support throughout this journey.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **HTAP** | **H**brid **T**ransaction/ **A**nalytical **P**rocessing |
| **OLTP** | **O**n **L**ine **T**ransaction **P**rocessing |
| **OLAP** | **O**n **L**ine **A**nalytical **P**rocessing |
| **KV** | **K**ey **V**alue |
| **RDBMS** | **R**elational **D**atabase **M**anagement **S**ystem |
| **ACID** | **A**tomicity **C**onsistency **I**solation **D**urability |
| **DAG** | **D**irect **A**cyclic **G**raph |
| **ETL** | **E**xtract **T**ransform **L**oad |

# Chapter 1

# Introduction

Stream processing systems over the years have evolved to support real time analytics, central to this utility is the state management model. The need to manage state like a 'first class citizen' arises for many interactive and reactive data applications like Event-driven services , Online ML model training and inference, etc. As stream processors mature to support analytics on top of real time transaction processing-managing large scale, distributed and consistent state has become a challenge. This raises a significant architectural design question of how the state is managed, stored and can be queried [8]. A well rounded overview of stream processing systems drawn from, [11] states the programmability, architectures and open challenges of state management. In stream processing, state consists of all internal data required to produce correct output - which includes sliding windows , joins , aggregation counters and such. Steaming systems like Apache Flink [3], Spark [4] and Millwheel[1] introduced user-programmable state, allowing developers to define and manipulate state directly via API calls. These systems balance usability with system-managed persistence and scalability. The design choices of state management influence the way internal state can be queried externally. Being able to query the state allows the user to see an exposed view of the state amid transaction processing in the streaming system. This can be used for auditing, debugging and real-time analytics. For example - fraud detection application running on top of a streaming service can aggregate and filter state data to flag transactions above a certain amount or daily limit. However, querying live state externally presents several challenges. In systems with partitioned state, accessing a complete global view requires coordination across partitions. Since the state is continuously mutating, any attempt to access it must not interfere with ongoing transaction processing. At the same time, the retrieved state should be as fresh and consistent as possible. This is similar to querying a recent, but potentially slightly stale, version of the data. Several challenges exist to query live state, among them are data synchronisation, query optimisation, performance isolation and resource scheduling. Multipurpose systems like HTAP databases aim to solve these issues for OLTP and OLAP workloads. These architectural choices can be applied in streaming engines to serve state queries as a way to handle analytical workload for continuous stream of data.

## 1.1 Research Questions

This issue raises the following research questions:

1. How does integrating an active global state store affect the transactional stream processing performance?

2. How fast can the state store be updated in a distributed setting ?

3. How fresh can the fetched state data be for analytical workloads?
4. What consistency guarantees can be achievable when querying the state?

## 1.2 Contributions

In this thesis we propose Q-styx a system that queries the distributed state of a streaming processor. Q-Styx exposes the internal state of operators for external systems to query. Q-Styx is implemented in Styx [20], a streaming dataflow engine that guarantees serializable isolation and low latency performance. Our experimental evaluation suggests that Q-Styx adds only about 2ms in transaction latency in the 99 th percentile. We make the following contributions to current research:

- Design and implementation of a global state store, that exposes the state of a streaming processors transaction for querying, while minimally impacting Styx's throughput and latency.
- Retrieving all the distributed state data without the need for the data to be sinked to any external data store.
- Q-Styx guarantees Snapshot Serializable Isolation for consistent query results.
- Thorough evaluation of Q-Styx's performance using various state sizes, and input throughput. Using freshness scores to evaluate the staleness of data.
- And finally, we offer ways to improve the system further for better performance and scalability.

## 1.3 Outline

In chapter 2 we discuss the background and motivation for this thesis and go over some preliminary knowledge about state management and isolation levels required for the rest of the thesis. In chapter 3 we discuss the architecture and implementation of Q-Styx and in chapter 4 we evaluate the system performance based on the defined metrics. And finally in chapter 5 we conclude with some observations for future work.

# Chapter 2

# Background

## 2.1 Transactional and Analytical Processing

Traditional data processing systems separate transactional/operational data - that systems use for running the business which is stored in databases and analytical data - that is used for insights and is stored in buckets. Each of these OLTP and OLAP are designed to serve a distinct purpose. Moreover, advances in both OLAP and OLTP have driven the emergence of HTAP systems which are designed to handle mixed workloads and serve both purposes. HTAP databases are challenging to build for scale while maintaining consistent views of the data in all the replicas. Databases like [16], [13], [24] have various storage strategies and target different applications. For example, some applications like e-commerce require high scalability, while banking applications require high data freshness. It also depends on wether transactions or analytics get priority.

## 2.2 Motivation

Through ETL processes that may take hours sometimes upto days, the OLTP data is periodically moved to OLAP systems. This limits real time decision making that would benefit many downstream applications. For example, if an application uses Machine learning to generate insights based on some user activity, the transactional data is used for analytics and fed back to the transactional system. Stream processing systems fit into this kind of architecture. Our work aims to apply the concepts of HTAP systems into stateful stream processing. Stream processing works on a continuous stream of data. Instead of generating analytics from an external data store that could be the sink of the stream processor, we intend to query the live state of the data in the stream. Exposing the distributed state of a streaming system has applications like:

- Real-Time Monitoring: Queryable state would allow external applications to directly access and monitor current state of a stream processing job. This would eliminate the need to rely on ETL process and stale data. For example, in a billing system , the current quarterly/monthly/weekly totals can be provided to downstream applications quite quickly.
- Debugging : Inspecting the state at specific points in time or during specific streaming jobs can help identify issues and anomalies. This can be of great assist to developers when particularly dealing with complex computations in a distributed system. This also aids in our understanding of streaming topologies in depth.

- Look Aside Cache : Applications that use look aside cache pattern benefit from a queryable state store. Instead of querying an external database for fast changing data, the state store can serve the metrics for low latency applications like gaming.

To be able to query the live state of the data, we have to dive into the stream processors and state backends.

## 2.3 Preliminaries

Here we will discuss the background knowledge required for the rest of the thesis to follow.

A stateful stream processor processes continuous data streams while maintaining a memory of past events, enabling operations like aggregations, joins, and windowed analyses. Stream processing applications are structured as Directed Acyclic Graphs (DAGs), where each node represents a processing function—either stateless (e.g., filtering) or stateful (e.g., aggregations) that remember the result of previous execution. There are two types of "states" in stream processing : The metadata of stream processing , that tracks the progress of the processing task, such as checkpointing and offsets which is essential for fault tolerance and recovery. The other state is the intermediate data maintained between processing steps, enabling operations like aggregations and joins. When we refer to "state" in stream processing, we refer to the intermediate data.

### 2.3.1 State Management

Maintaining state in stream processing involves storing, updating, and recovering state in case of failures. Storing state requires allocating memory or disk space to hold intermediate processing data. To ensure fault tolerance, state must be persisted in durable storage, enabling recovery after crashes. This is achieved through a state store, which can range from simple in-memory key-value maps to embedded stores like RocksDB[7], or even external systems like Cassandra[15]. The state store acts as a repository for intermediate results between processing steps. Different architectures support different needs: in-memory state offers low-latency access and is ideal for frequent, fast updates but is limited by available RAM. Out-of-core state architectures, such as those using RocksDB, extend the state beyond memory by spilling to disk, allowing for much larger state sizes at the cost of disk I/O latency. In externalized state stores the state completely resides outside the processing engine, like Google's BigTable, as used in MillWheel[1] , providing strong durability and transactional guarantees but introducing higher access latencies. An important aspect of state maintenance is persistence granularity—the frequency and scope at which state is saved. Epoch-level persistence captures state snapshots either periodically or after a fixed number of records have been processed. This is typically implemented using asynchronous consistent snapshotting, such as the Chandy-Lamport algorithm [9], where each operator in the system saves a consistent copy of its state during an epoch. In contrast, batch-level persistence, seen in systems like Spark Streaming rely on a micro-batching approach where the state is only persisted at the end of each batch, after a group of records have been collected and processed. Another important aspect of state management is partitioned and non-partitioned(global) state. While processing a stream in parallel, partitioning the state is a standard approach.

The data is grouped by keys and mapped to respective logical partitions for computations. Global state on the other hand is maintained as a single state instance over the complete input stream. This kind of state is useful for global operations like counting total events, keys per operator or for all operators. However, this approach does not scale well. Most modern streaming dataflow systems offer built in state management as the alternative would be to essentially make all operators stateless and externalise all intermediate data.

### 2.3.2 Quering a State Store

State management features can also extend to being able to query the state store from outside the system. This provides a read access to the latest values computed by the stream processor. This state of the system can be used for ad-hoc analysis through a SQL like querying interface. From a traditional distributed systems viewpoint, this is similar to querying the stale replicas of the master node. Combined with pluggable state backends mentioned in 2.3.1, users can interact with system-managed and user defined state through APIs provided by the stream processor like in [1],[3], [2], [14]. Most dataflow programming models assume a key-value schema for input records and always associate state with a key meaning that the state is partitioned by a key and any query or update operate only on partition-local state for its key.

### 2.3.3 Styx

Styx builds on stateful streaming dataflow execution model inspired by systems like Apache Flink, that supports Stateful Functions-as-a-Service paradigm [21] with deterministic transactions [20]. Styx co-locates state with the function logic which eliminates the need for external data stores. This allows for high performance with end to end serializable transactions and exactly-once processing guarantees. During runtime , Styx structures an application as a directed dataflow graph where the vertices represent the operators (stateful entities) and edges represent the flow of events (function invocations). The deterministic sequencing of functions guarantees that each transaction's sequence and effect are repeatable under failure and replay. Hence, Styx is able to handle both parallel, data-intensive streaming workloads. Operators are partitioned across multiple cluster nodes, each partition holds a set of stateful entities. During an incoming event when a function is invoked, the local state is retrieved from the operators partition, the respective function is executed and the state is updated. Styx offers epoch level state persistence, meaning, the entire task graph is committed after each epoch is processed. A complete execution of an epoch is useful to support isolated queries for analytics on-top of data streaming [23].

### 2.3.4 Isolation Levels

In Distributed systems, the isolation property of the ACID (Atomicity, Consistency, Isolation and Durability) [12] model ensures that transactions appear to execute independently. [6] defines and improves upon the ANSI-SQL standard stating certain isolation levels based on possible anomalies listed in the table and explained below.

| Isolation level | Dirty write | Dirty read | Lost update | Fuzzy read | Phantom read |
|---|---|---|---|---|---|
| Read uncommitted | Not possible | Possible | Possible | Possible | Possible |
| Read committed | Not possible | Not possible | Possible | Possible | Possible |
| Repeatable read | Not possible | Not possible | Not possible | Not possible | Possible |
| Snapshot | Not possible | Not possible | Not possible | Not possible | Possible |
| Serializable | Not possible | Not possible | Not possible | Not possible | Not possible |

TABLE 2.1: Phenomena possible in different isolation levels [6].

- **Dirty write:** Occurs when two transactions modify the same item and if either rolls back, the final value is not correct.
- **Dirty read:** Occurs when a transaction reads uncommitted changes from another transaction that later on is rolled back. This leads to the first transaction reading an incorrect value.
- **Lost update:** Occurs when an update by one transaction overwrites an update made by another transaction. The initial update is lost.
- **Fuzzy read:** Occurs when a transaction reads the same value twice but sees different results due to a concurrent update from another transaction.
- **Phantom read:** Occurs when a transaction re-executes a query and sees a different set of rows because of inserts or deletes from other transactions.

Table 2.1 summarizes which phenomena can occur for different isolation levels. Stronger levels like *Serializable* prevent all anomalies, while weaker levels like *Read Uncommitted* allow dirty reads, lost updates, and other inconsistencies.

As mentioned in 2.3.3, Styx explicitly provides serializability, which is the strongest isolation level in the ACID model. In Styx, the deterministic transaction protocol sequences transactions in global order, tracks read/write sets and deterministically resolves conflicts. All state changes from a transaction appear atomic and isolated, even across distributed operator partitions. This isolation level is stronger than snapshot isolation, repeatable read, or read committed — because it prevents phenomena like write skew and phantom reads too. Styx is well suited for real-time global queries on its state store for the following reasons:

- Co-location of state and compute : Because the state lives inside the running dataflow graph , there is no external database we have to access.
- Determinstic Execution: Styx guarantees serializability and exactly once processing, during a query run on its state , the results will reflect a valid cut of the system without any dirty read(no uncommitted data is read due to epoch level persistence).

## 2.4 Related Work

Kafka Streams [14] allows users to interactively query the state store. This is implemented through RPCs(Remote Procedure Calls) and the know-how of which partitions the keys lie in. This results in a distributed database for state, embedded in an application. Moreover, kafka streams guarantees eventual consistency which could result in temporary incorrect results and edge cases in business logic. This is not ideal for applications that require strict consistency. Although kafka streams provide the ability to query the state, its a library and not a complete stream processing system with queryable state utility.
Apache Flink[3], on the other hand is a comprehensive stream processing system. The QueryableStateServer is responsible for serving the query from the state store after the state store is determined by the QueryableStateClientProxy. In Flink the state object when queried is directly accessed from a concurrent thread without any synchronisation or copying. The read patterns may become unsafe and cause the queryable state server to fail due to concurrent modifications.
Apache Samza [19] maintains state alongside the processing logic. It supports queries to its state however , not out of the box. Each task handles queries to its own local state store meaning , you need to route requests based on partition information to the correct local store. There's no global look up.

# Chapter 3

# Methodology

This chapter describes the architecture, implementation and the isolation levels of the qureyable state system in Styx [1], from here on out we will call it Q-Styx .[2] . Ad-hoc SQL queries can be submitted to the state store anytime, executed once, to provide insights into the systems current state which is the live state of the operators distributed across the streaming system.

## 3.1 Architecture

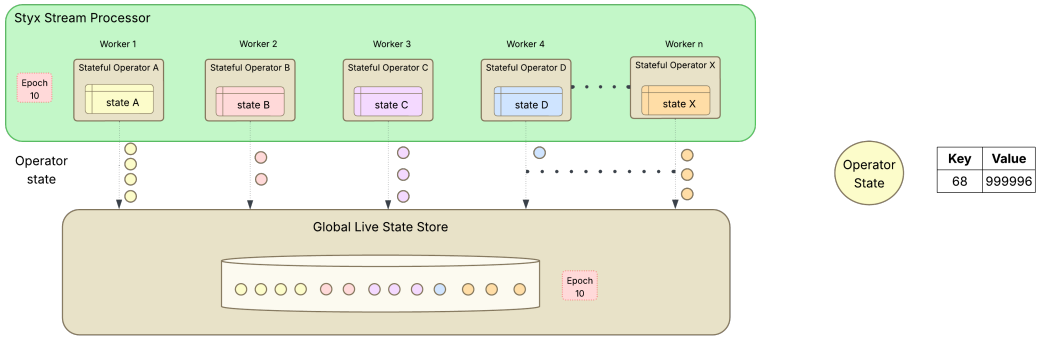


FIGURE 3.1: Global State Architecture

The high level architecture of the state store is depicted in Figure 3.1. There are two separate but tightly coupled systems. Q-Styx service has an in-memory global key-value state store. Q-Styx primarily runs two concurrent tasks - it ingests delta updates from Styx and processes external queries against the current state. The delta updates are state changes tracked per epoch. These deltas are merged to get a consistent global view of the Styx systems distributed state which represent the operators , partitions and their respective keys and values hence the name, global state store. Q-Styx reflects the latest committed state of the transactions in Styx providing low latency access of the state for external analytical queries.

### 3.1.1 Design Challenges

Extending Styx to support real-time analytical queries directly over its internal state introduces several fundamental architectural challenges. This resembles the goals

[1] https://github.com/delftdata/styx
[2] https://github.com/mitalipatil99/Q-Styx

of modern Hybrid Transactional and Analytical Processing (HTAP) systems, which aim to unify Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) workloads in a single runtime. As discussed in recent HTAP surveys and systems [17], [25], combining transactional processing with analytical queries involves a balance between data freshness, workload isolation, and system performance. Hence, it is critical to understand the challenges and trade offs under various architectures.

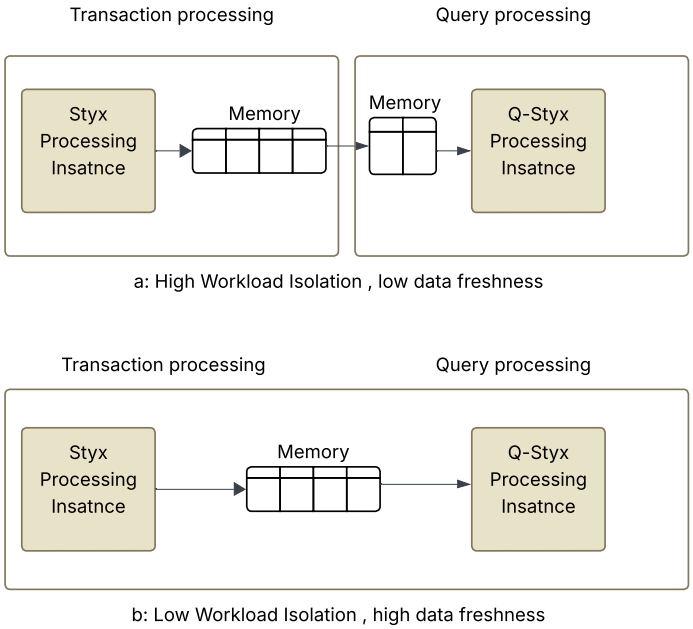


FIGURE 3.2: Workload Isolation vs Data Freshness

**Data Freshness vs Performance Isolation:** A key architectural challenge when integrating a queryable state store with Styx is managing the trade-off between data freshness and performance isolation. Data freshness refers to how up-to-date the query results are with respect to the latest committed transactions in the processing system. To support real-time analytics, Q-Styx must access the most recent state updates produced by Styx's transactional functions. This requires fast, frequent synchronization of state from the transactional runtime to the analytical store. Performance isolation, on the other hand, reflects how well the system can minimize interference between analytical workloads and transaction processing. In Styx, transactional functions continuously mutate state while ensuring strong serializability and exactly-once guarantees. Adding analytical queries introduces new load that can compete for CPU, memory, and locks, potentially slowing down transaction throughput. This creates a trade-off of depicted in Figure 3.2.

Running transactions and analytics in separate nodes or instances can achieve high performance isolation — since both processes would not contend for the same resources shown in part a of Figure 3.2. However, this comes at the cost of lower data freshness, because newly committed transactional state must be replicated to the analytical store, which introduces synchronization lag. Queries may then return slightly stale results if the replication is delayed.

Running transactions and analytics in the same memory space provides maximum data freshness, as both workloads operate directly on the live, mutable state as shown in part b of Figure 3.2 . However, this approach can degrade overall performance, because analytical queries share CPU cycles, memory bandwidth, and locks with the stateful functions, potentially slowing down the transaction processing.

Therefore, we need to carefully balance these requirements by: Providing low-latency access to the latest committed state for analytical queries while minimizing the performance impact on Styx's core transactional workload to maintain high throughput and strong consistency guarantees.

**Data Synchronization:** Another challenge is to decide when and how to synchronize state changes to the queryable state store. To minimize interference with Styx's transaction processing, it is necessary to maintain a separate replica of the state changes for analytical queries. However, this introduces a trade-off: If synchronization occurs too frequently, the analytical replica will always reflect the latest state, providing high data freshness for queries. However, frequent copying and merging of state changes can consume significant network bandwidth and can increase the merge overhead especially if the state changes are big. On the other hand, if the data replication updates are too infrequent, the state store may serve stale data, which will undermine the goal of real-time analytical querying.

## 3.2 Implementation

In the context of Styx, by addressing the above mentioned trade-offs to build Q-Styx on top of it, the following design choices have been made: We incrementally ship state changes after each epoch to another service over a TCP network to minimise the performance impact on Styx's transaction function calls. The state data is replicated to an in-memory KV store. This enforces isolation for analytical queries so that long running queries do not degrade Styx's performance and blow up memory for eg. to hold intermediate aggregation results. With a focus on data freshness, in the next chapter 4 we evaluate the performance degradation in Styx mentioned in section 4.2 , Data Synchronization Performance between the two systems mentioned in section 4.3 , the Freshness of results mentioned in section 4.4 and Query Performance evaluated in section 4.5

### 3.2.1 Global State Store

Since the stateful operators are located across the partitions in Styx, to support querying of live state, a global view of the system state is essential. We construct this view compacting all deltas up to the current epoch, effectively merging all state changes into a unified, consistent snapshot. This compaction enables external queries to access the complete operator state across all partitions and for all operators without requiring distributed coordination. The resulting global state store serves as an in-memory KV database that offers low-latency, local access to the current state. It eliminates the need for cross-node communication during query processing because now the operators and key's are co-located. Figure 3.3 illustrates transactions for an application with three operators(stock, payment and order) partitioned across 4 nodes. The global view store's all the KV pairs of all operators in Styx's transaction processing.

The source of truth for this state store are the committed transaction's state per epoch in Styx. During compaction the latest value per key is overwritten , reducing memory overhead while maintaining the latest state values. This design choice simplifies the complex coordination logic and partition alignment required for key to instance routing .

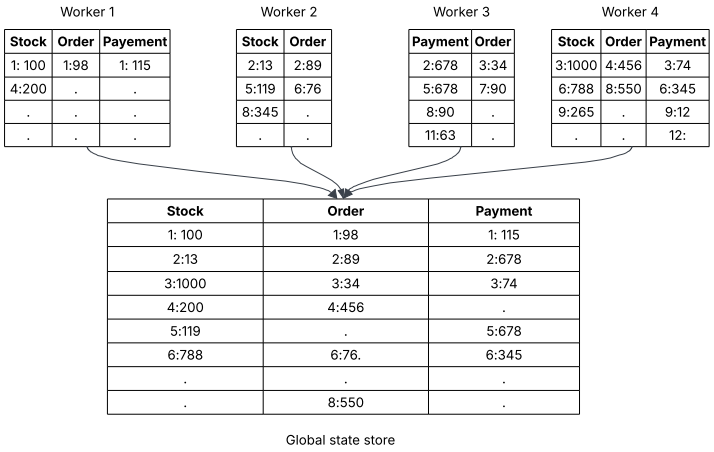


FIGURE 3.3: Global State Store Schema

### 3.2.2 Updating the State

Q-Styx maintains an In-Memory KV Data Structure that is updated incrementally using state changes (called delta maps) from distributed worker nodes. Each worker node processes a portion of the stream and reports its local state changes for each epoch. At the end of the transactional epoch in Styx, state updates are grouped, per worker dictionary containing a mapping of [operator, partition]-> key: values pairs of the respective epoch. These deltas are asynchronously sent over a TCP network to Q-Styx where the state is only updated in the store once all workers have submitted their deltas for the next epoch to be updated. The state store uses the same partitioned key as the operators in Styx to store the value. This non blocking operation does not interrupt the ongoing transaction processing. Moreover , this structure lends itself to fault tolerance ready since all the deltas are stored in a buffer until a full set is ready to be successfully merged. This also ensures atomic updates to the state store per epoch - either the entire epoch update is committed or its not.

### 3.2.3 Querying the Live State

The decoupled KV store can now be queried externally to retrieve the state of the operators during ongoing transactions. An SQL query can be used to get the state of a processing job. Clients can send structured query messages that include the operator and optionally the partition or the key. For example, *getKeyState/Stock/1245* , here we are querying to get the value of key 1245 from the Stock operator. With just the key in the query, the same hash partitioning mechanism in Styx is used to locate the partition for retrieval. The queries supported so far are mentioned in Section 4.1. Q-Styx responds asynchronously with the query results. This architecture ensures strong isolation between transaction and analytical processing .

## 3.3 Isolation levels

In this section we discuss the isolation levels achievable for Q-Styx. Referring to the isolation levels mentioned in Section 2.3.4, the following table shows the Q-Styx guarantees.

| System Guarantee | Dirty Read | Dirty Write | Fuzzy Read | Lost-Update | Phantom Read |
|---|---|---|---|---|---|
| Snapshot Isolation | Not Possible | Not Possible | Not Possible | Not Possible | Not Possible |

TABLE 3.1: Guarantee of Q-Styx

Q-Styx has Serializable Snapshot Isolation (SSI) by design, which provides strong consistency guarantees. Q-Styx's role is to serve queries against a committed state in Styx. Queries observe a consistent cut/snapshot of the state as the state is updated after an epoch is committed in Styx.

Since queries only see the committed data of the latest updated epoch state, Dirty Reads are not possible. Similarly, Dirty Write phenomenon cannot occur because the state is locked during the update process while the latest epoch is applied, preventing concurrent writes that might overwrite uncommitted data.

Queries served against a current committed epoch in the state store cannot see multiple versions of the same data; the state store is immutable for the duration of the query. Hence, if a query is being processed the transaction is locked to prevent fuzzy reads and to ensure consistency.

Because we lock the state store during an update and only process all the epoch state changes serially, There are no concurrency write conflicts and the queries are read only which prevents the Lost Update phenomenon.

Furthermore, when a query starts, it is served from the previously committed epoch view. Phantom Reads are avoided because the no changes are added to the state store and the keys in it during the query execution, even if the next epoch state is ready to be committed.

Q-Styx's design guarantees that queries always see and read a consistent isolated snapshot of Styx's state answering the Research Question 4.

# Chapter 4

# Evaluation

The evaluation of the query state system is performed by answering the following questions:

1. How does integrating the active global state store impact the performance of transactional stream processing in Styx. (Section: 4.2)
2. How fast is the state store updated.(Section: 4.3)
3. How fresh can the state data be for analytical workloads. (Section: 4.4)

## 4.1 Setup

**System Under Test.** In all the following evaluation we use two systems. Styx: ,stateful dataflow system and Q-Styx the query extension for Styx. Both the systems are implemented in Python 3.12 and use coroutines for asynchronous concurrent execution. Apache Kafka is used as a durable input and output queue for messages. Both are standalone containerized systems deployed using Docker.

**Workload/Benchmark.** We use the YCSB-T, [10] benchmark where each transaction performs two reads and two writes. For example for 1000 unique bank account keys, a simulation is performed to transact money from the debtors account to the creditors given sufficient credit exists for the payment. If not a rollback is performed. We select the debtor key based on an uniform distribution from a given key space size, for example 1k, 10K and 100k keys.

**Resources.** Throughout all the evaluation, Styx system was deployed with 4 worker nodes and 1 coordinator node, Q-Styx is deployed with 1 Queryengine node. Each node is allocated 1 CPU and 32GB of RAM respectively.

**Metrics.** We aim to observe the latency of the systems under test against varying transactions, queries and key sizes.

1. Input Throughput is defined as the number of transactions submitted to Styx per second (TPS). We expect the latency of transactions to increase as the load to the system increases.
2. Transaction Latency is defined by the time interval between a transaction submitted to Styx and the commit time of the same transaction. In Styx the latency timer starts when a transaction is submitted in the input queue (Kafka) and stops when the system reports the transaction as committed/aborted in the output queue.

3. Epoch State Update Latency is defined as the time interval between the completion of an epoch in Styx and the successful commit of the corresponding state updates in Q-Styx's state store. This metric assess the performance of state state synchronization mechanism between Styx and Q-Styx. To capture this metric we measure two timestamps per epoch:

   (a) Epoch Completion Time — when Styx finalizes all transactions in that epoch and commits them locally.

   (b) State Store Commit Time — when Q-Styx finishes applying and persisting the corresponding state updates.

   The difference between these timestamps represents that state propagation latency, which includes the Network communication delay (pushing deltas to Q-Styx), Serialization and Deserialization costs,and the processing overhead for updating the state store.

4. Query Throughput is defined by the number of analytical queries that are submitted to Q-Styx per second (QPS). Here, we expect Q-styx to maintain an acceptable latency as the the queries increase.

5. Freshness Score is defined as the recency of the data read by analytical queries. Following [18], we define the freshness score of an individual analytical query $q$, denoted as $F_q$. This metric measures the staleness of the snapshot visible to the query relative to the most recent committed state in the transactional engine.

   Formally, the freshness score of query $q$ is:

   $$F_q = A_{\text{start}} - T_{\text{first\_unseen}}$$

   where:

   - $A_{\text{start}}$ is the global clock time when the analytical query starts. In our system, this corresponds to the time when the query processing begins in Q-Styx.
   - $T_{\text{first\_unseen}}$ is the commit time of the *first transaction* whose updates are *not visible* to the snapshot read by $q$. In our system, this aligns with the epoch commit time in Styx for which the query is served by Q-Styx.

   By definition, $F_q = 0$ indicates that the snapshot seen by the query is fully up-to-date and includes all transactions committed before the query started. If $F_q > 0$, the snapshot is stale by the given time interval.

   In realistic workloads, multiple analytical queries run concurrently, each with its own freshness score. Therefore, the overall system freshness is an aggregated metric, such as the average or the $95^{\text{th}}$ percentile of all query freshness scores.

6. Query Latency is defined as the time interval between the submission of a state query to the Kafka queue in Q-Styx and the receipt of the corresponding response by the user. Here, we expect the query latency to increase with the query throughput.

**Queries.** The following point queries have been used in random generation order in Q-Styx to retrieve the state data from Styx.

1. `getKeyState/Operator/Key`, e.g., `getKeyState/Order/1245`
   This query fetches the state of the Operator's specified key.

2. `getOperatorState/Operator`, e.g., `getOperatorState/Payment`
   This query fetches the entire Operators state across all partitions.

3. `getOperatorPartitionState/Operator/Partition`,
   e.g., `getOperatorPartitionState/Stock/2`
   THis query fetches the state of a particular partition of the mentioned operator.

## 4.2 Evaluation of Transaction Latency: Impact of Q-Styx

To answer Research Question 1, we compare the baseline Styx runtime with Q-Styx, which asynchronously replicates the post-epoch committed state. We conducted controlled experiments focusing on transaction latency under varying input throughput and key spaces. The experiments were run for 240 seconds with a warmup of 10 seconds.

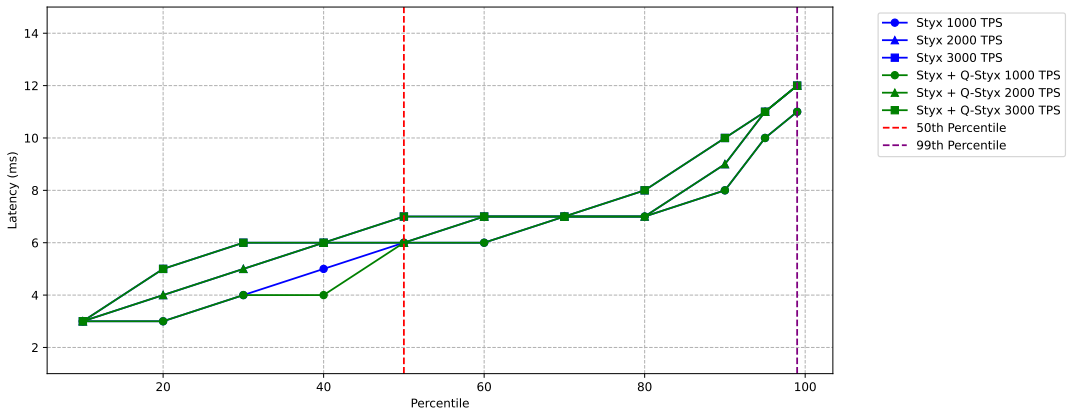


FIGURE 4.1: Latency vs Throughput

Figure 4.1 highlights the 50$^{th}$ and 99$^{th}$ percentile latencies of Styx runtime and Styx with Q-Styx enabled, tested at 1K, 2K and 3K TPS respectively with 10,000 unique keys. We expect the latency to increase as the throughput increases and baseline run of Styx shows a similar pattern. The overlapping greenline on blue in the reference figure indicates that adding Q-Styx shows almost no change in transaction latencies. Thus, we conclude that Q-Styx has a minimal to no impact on the core performance properties of Styx when the input throughput increases.

To further examine query impact of Q-Styx when we scale the data, we measure how the transaction latency changes as we increase the key space. When the key space is small, the total state per worker is limited — so the data transferred after each epoch is relatively small. However, when the key space grows, each worker's partition contains proportionally more key-value pairs. Consequently: More data must be serialized, transferred by Styx and, deserialized and ingested by the Q-Styx system. This replication happens asynchronously, but it shares bandwidth and CPU resources with the main transaction pipeline. The larger the key space, the more replication work is done in the background after each epoch boundary. Hence, the observed effect in the experiments is a slight increase in transaction latency as shown in Figure 4.2. The 99$^{th}$ percentile shows a 2ms increase in transaction latency for 100K keys.

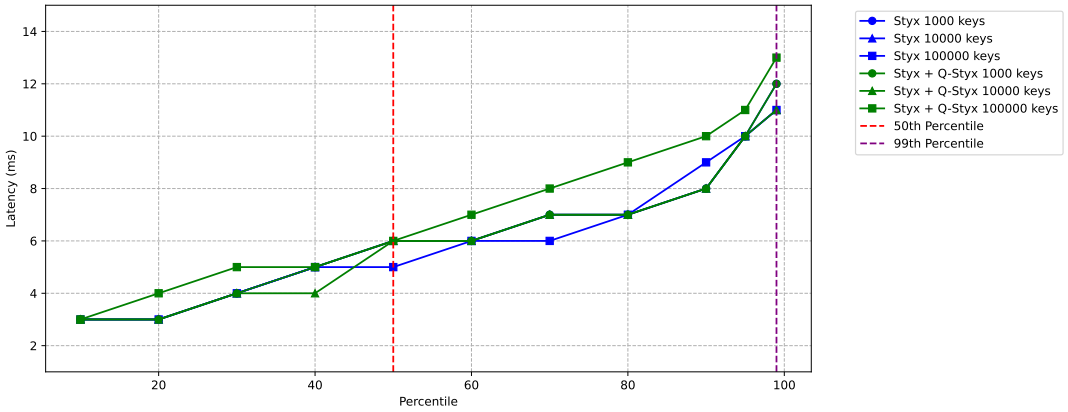


FIGURE 4.2: Latency vs key scale

This goes to show that Q-Styx can be integrated with Styx for live state query capability while keeping Styx's high throughput and low latency performance unchanged.

## 4.3 Data Synchronisation Performance

To answer Research Question 2, we compare the Epoch State Update Latency against an incremental query throughput and varying key space. This will in part help us understand how performant the two systems are for replicating the state data from Styx to Q-Styx.

Figure 4.3 shows the state propagation latency at the $50^{th}$ and $99^{th}$ percentiles across key spaces of 1K, 10K, and 100K unique keys, with query rates ranging from 1 to 5 QPS. For smaller key spaces, the state delta per epoch remains small, so state updates are consistently fast (typically under 4ms) and remain stable even as query rates increase. In contrast, larger key spaces lead to larger state deltas per epoch, which significantly degrades state update performance. This degradation is primarily due to the increased network overhead of transmitting larger deltas, combined with the serial bottleneck of waiting for state updates from all workers — which can only proceed as fast as the slowest worker. Additionally, each received delta must be deserialized and the full operator partition updated accordingly. All of this must be executed serially to preserve isolation guarantees. As a result, higher propagation latency directly impacts query's freshness results: slower epoch updates lead to more stale state in Q-Styx, reducing the effectiveness of real-time analytical queries.

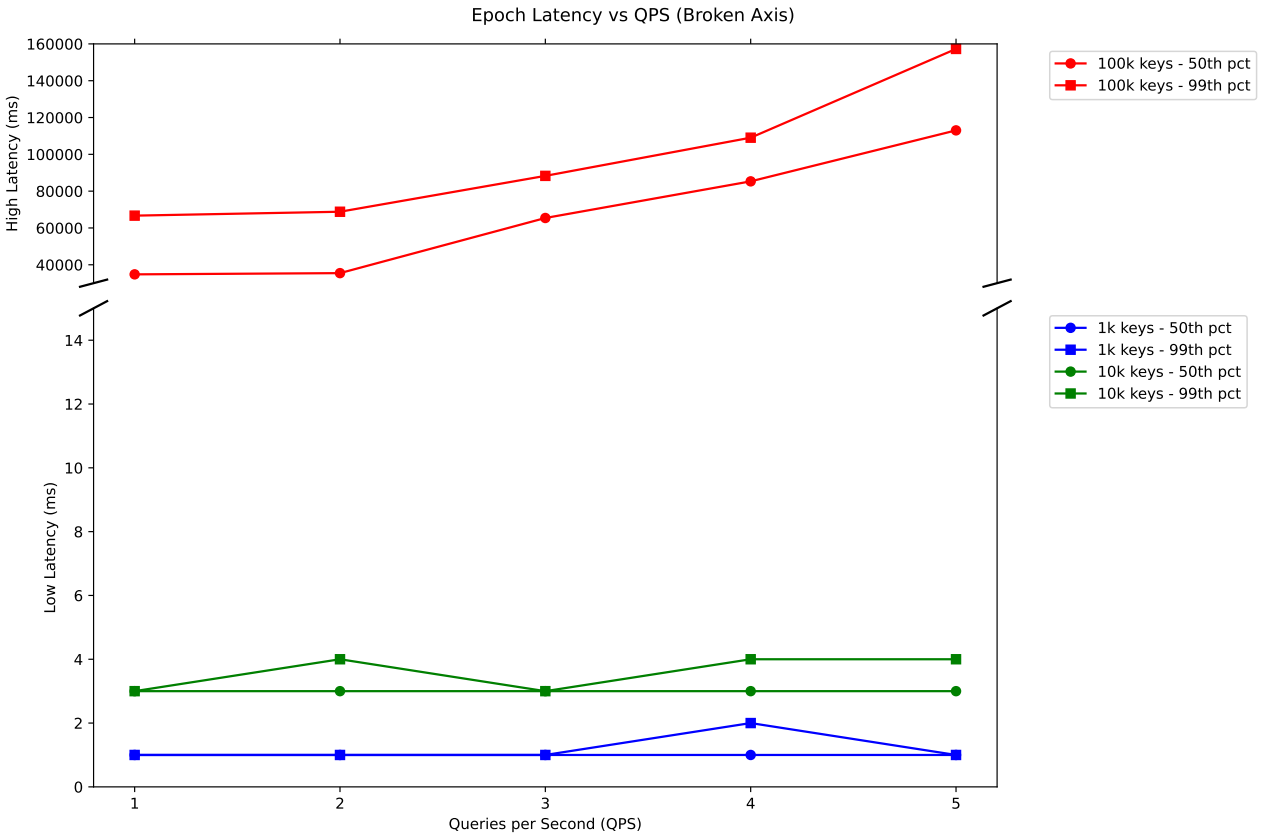


FIGURE 4.3: State Update Latency

## 4.4 Freshness Score

To answer research question 3 we measure the average freshness scores from query responses for incremental QPS and varying key spaces. In this evaluation , we ran the Styx engine with Q-Styx for 240 seconds while simultaneously processing the queries mentioned in the setup.

| # Keys | 1 QPS | 2 QPS | 3 QPS | 4 QPS | 5 QPS |
|---|---|---|---|---|---|
| 1k | 2.44 | 2.94 | 2.98 | 2.87 | 3.24 |
| 10k | 3.87 | 4.07 | 4.04 | 4.18 | 4.12 |
| 100k | 40409.31 | 76737.34 | 87393.24 | 89601.97 | 89222.86 |

TABLE 4.1: Average freshness score in milliseconds for different key spaces and query rates (QPS).

Lower freshness scores indicate better alignment between real-time transactional commits to Styx and the snapshots used by analytical workload in Q-Styx. Higher scores highlight delays in propagating state updates from Styx's core engine to Q-Styx, which can directly impact the timeliness of query results. The average freshness scores are shown in Table 4.1. The average scores reported for 1k, 10k unique keys is consistent and <=4ms which is good for most large scale applications. When the state size grows to 100K keys , there is a finer balance to maintain between updating the larger delta sizes to the store whilst processing queries.

## 4.5 Query Performance

In part, another evaluation criteria for Q-Styx's efficiency is its query performance which is a measure of its query latency. We evaluate the latency for the same configurations mentioned in the previous experiments.

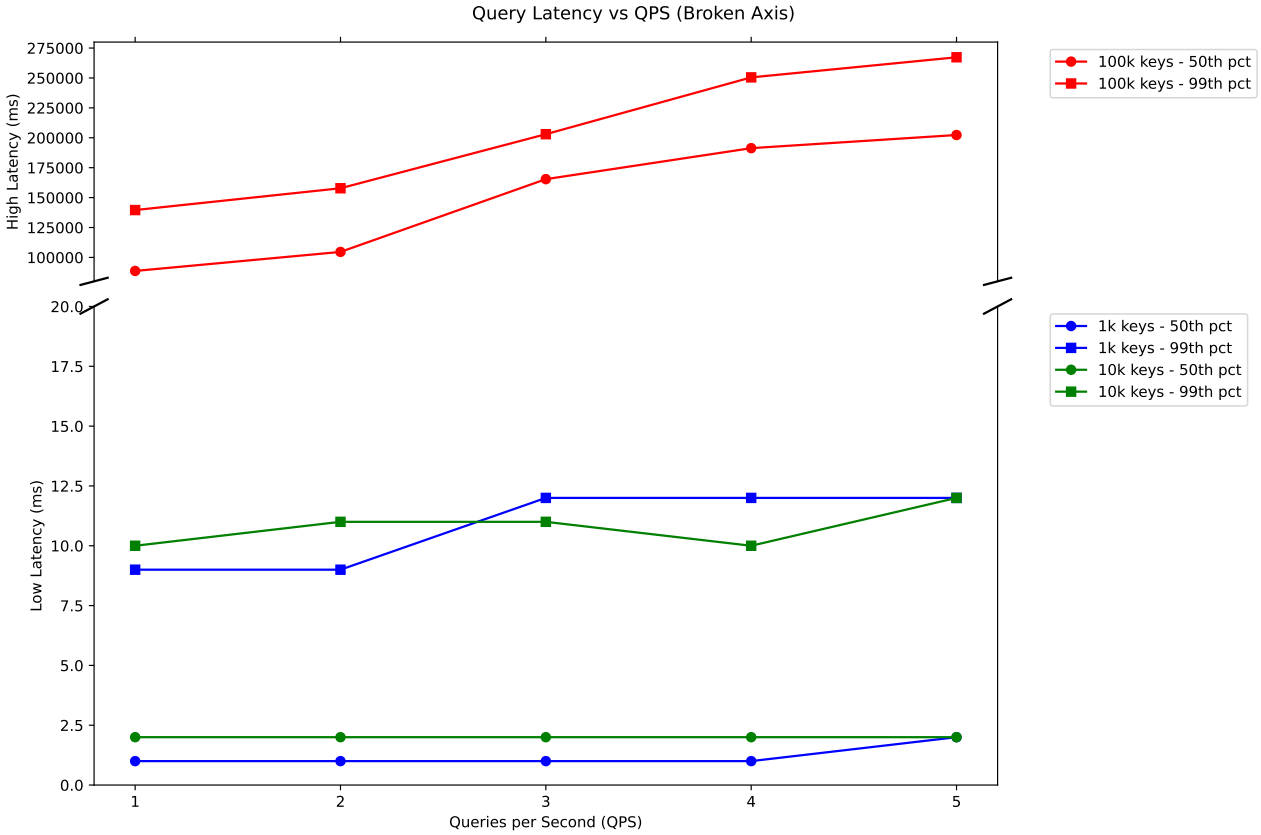


FIGURE 4.4: Query performance showing the 50th and 99th percentiles of query latencies for varying key space sizes.

Figure 4.4 illustrates the 50th and 99th percentile query latencies for key spaces of 1K, 10K, and 100K keys. We observe that query latency is influenced by the trade-off between freshness and responsiveness. Serving fresher data requires Q-Styx to prioritize applying new state store updates as soon as they are propagated from Styx. This synchronization step can temporarily delay query processing, as the state store must maintain isolation and consistency guarantees while applying incoming deltas. In contrast, serving queries over a stale state data can reduce response time but at the cost of freshness, because the system can immediately process queries without waiting for the latest updates to be applied.

## 4.6 Summary

| Factor | 1K Keys | 10K Keys | 100K Keys |
|---|---|---|---|
| State Delta Size | Small | Medium | Large |
| Network & State update Apply Cost | Low | Medium | High |
| Data Sync to Q-Styx | Quick | Acceptable | Slow |
| Freshness Score | Low | Still Low | **High** |

TABLE 4.2: Comparison of system factors across different key space sizes.

## 4.7 Limitations and Takeaways

In our current implementation, we use Python's asyncio library, [5] to execute concurrent tasks within Styx and Q-Styx. At its core, asyncio provides an event loop that orchestrates the execution of many tasks concurrently within a single-threaded context. This is especially effective for I/O-bound operations, such as network or disk I/O as it allows the program to continue to perform other tasks while waiting for the I/O tasks to complete.

However, when running CPU-bound operations inside a coroutine, asyncio can become a performance bottleneck. Operations like serialization, deserialization, compression, and decompression of large state deltas are CPU-intensive. Running these inside the event loop can block the coroutine. In our case, degrading the overall epoch state update performance for 100k keys. Additionally, creating too many tasks at a time leads to frequent context switching, which further reduces the performance. To limit excessive task creation, we could use semaphores to limit the number of active tasks at any time especially when we have limited resources.

For CPU-bound tasks in Q-Styx like processing state delta updates, a better approach is to use threads to exploit multiple CPU cores. One approach would be to use concurrent.futures.ProcessPoolExecutor. This would allow multiple processes to run in parallel, fully utilizing multi-core hardware. In our context, offloading the serialization and compression tasks to a process pool could potentially reduce epoch update latency, especially for large delta sizes like 100K keys.

Another potential performance bottleneck at larger key scale (e.g., 100K keys) is the use of locks for enforcing isolation guarantees when updating and reading from the state store. In our current implementation we use locks to prevent dirty reads and writes when we update the new epoch deltas. If the updates are bigger, the locks are held for longer which increases the latency for both reads and writes.

To improve query performance(query latency) for higher query throughput, we can adopt a multi-version concurrency control (MVCC) approach. In this method, we maintain two versions of the state store in memory. While one version copy is being updated with the latest committed epoch's state, the other version copy continues to serve queries allowing for atomic reads. Once the state update is complete, the system can switch a version marker to make the latest updated version copy active to serve queries, while the other copy processes the next state update. This eliminates the need for locking during reads and writes, allowing queries to be read without having to wait for the update to complete. MVCC can therefore significantly improve query latency and throughput under bigger update load.

We also explored py-linq [22] as a query language that would allow us to treat the in-memory KV state store as a collection to perform more expressive queries on. However, it requires the collection of objects to be an enumerable structure which would have required an extra step of conversion after every state update. For large delta sizes this is an added overhead that will decline the performance making it impractical for our application.

# Chapter 5

# Conclusion and Future Work

## 5.1 Future Work

While Q-Styx demonstrates a promising approach for supporting queries on live state synchronized with Styx's streaming transactional core, several directions remain open for future improvement and evaluation.

Firstly, the next step would be to evaluate Q-Styx on a realistic analytical and mixed workload benchmark like **HATtrick** or **CH-benCHmark** , which covers more complex query patterns beyond point lookups, including JOINS, FILTERS, and AGGREGATIONS. This will help validate the system's ability to serve hybrid workloads under realistic analytical queries.

Currently, Q-Styx materializes epoch-to-epoch state. If applications require access to historical state data upto a certain point in time, for example, queries that span not just the current epoch but also past committed epochs, then we can spill older state versions to disk. This would make Q-Styx's state store a hybrid, in-memory for fresh data with persistent storage for historical reads and fault tolerance in the solid state drive.

Another direction for higher query throughput and faster scans would be to adopt an in-memory columnar store design like SingleStore or Oracles IM column store. This would enable faster scans for workloads with large reads or wide table projections.

Another fine grain optimisation can involve locking only the rows that are being updated. Even if some keys have not changed for several epochs, queries that touch these keys in the current implementation must still wait for the entire epoch's update cycle to complete. If only rows being updated would be locked, and queries that read from unchanged rows could proceed immediately. This would reduce unnecessary blocking for keys that have not changed across epochs and further improve real-time freshness for queries.

These areas would make Q-Styx a more high performant In-Memory state store in conjunction with streaming processors for real time analytical queries.

## 5.2 Conclusion

This thesis presents Q Styx, a system supporting SQL like queries to the distributed state of a stream processing system (Styx). We are able to query the live state of Styx with a serializable snapshot isolation guarantee. We find that Q-Styx adds very little overhead to Styx transaction latency and has good performance characteristics from thousands to tens of thousands unique keys. And most importantly, Q-Styx exposes intermediate state querying capabilities for numerous applications.

# Bibliography


[1] Tyler Akidau et al. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Very Large Data Bases*. 2013, pp. 734–746.

[2] Apache Beam. *Apache Beam*. https://beam.apache.org/. Accessed: 2024-07-03. 2024.

[3] *Apache Flink*. Open-source, distributed streaming and batch data processing engine. 2023. URL: https://flink.apache.org.

[4] *Apache Spark*. Open-source distributed computing framework. 2011. URL: https://spark.apache.org/.

[5] *Asynchronous I/O*. https://docs.python.org/3/library/asyncio.html.

[6] Hal Berenson et al. "A critique of ANSI SQL isolation levels". In: *SIGMOD Rec.* 24.2 (May 1995), 1–10. ISSN: 0163-5808. DOI: 10.1145/568271.223785. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/568271.223785.

[7] Dhruba Borthakur et al. *RocksDB*. https://rocksdb.org/. Accessed: 2024-06-24. 2013.

[8] Paris Carbone et al. "Beyond Analytics: The Evolution of Stream Processing Systems". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 2651–2658. ISBN: 9781450367356. DOI: 10.1145/3318464.3383131. URL: https://doi.org/10.1145/3318464.3383131.

[9] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of a Distributed System". In: *ACM Transactions on Computer Systems* (1985), pp. 63–75. URL: https://www.microsoft.com/en-us/research/publication/distributed-snapshots-determining-global-states-distributed-system/.

[10] Akon Dey et al. "YCSB+T: Benchmarking web-scale transactional databases". In: Mar. 2014, pp. 223–230. ISBN: 978-1-4799-3481-2. DOI: 10.1109/ICDEW.2014.6818330.

[11] Marios Fragkoulis et al. *A Survey on the Evolution of Stream Processing Systems*. 2023. arXiv: 2008.00842 [cs.DC]. URL: https://arxiv.org/abs/2008.00842.

[12] Theo Haerder and Andreas Reuter. "Principles of transaction-oriented database recovery". In: *ACM Comput. Surv.* 15.4 (Dec. 1983), 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/289.291.

[13] Dongxu Huang et al. "TiDB: a Raft-based HTAP database". In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), 3072–3084. ISSN: 2150-8097. DOI: 10.14778/3415478.3415535. URL: https://doi-org.tudelft.idm.oclc.org/10.14778/3415478.3415535.

[14] Apache Kafka. "Interactive Queries". In: *Kafka Documentation* (2024). https://kafka.apache.org/3
guide/interactive-queries.html.

[15] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *Operating Systems Design and Implementation (OSDI)*. Available at https://cassandra.apache.org/. USENIX Association, 2009.

[16] Juchang Lee et al. "Parallel replication across formats for scaling out mixed OLTP/OLAP workloads in main-memory databases". In: *The VLDB Journal* 27.3 (June 2018), 421–444. ISSN: 1066-8888. DOI: 10.1007/s00778-018-0503-z. URL: https://doi-org.tudelft.idm.oclc.org/10.1007/s00778-018-0503-z.

[17] Guoliang Li and Chao Zhang. "HTAP Databases: What is New and What is Next". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, 2483–2488. ISBN: 9781450392495. DOI: 10.1145/3514221.3522565. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3514221.3522565.

[18] Elena Milkai et al. "How Good is My HTAP System?" In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, 1810–1824. ISBN: 9781450392495. DOI: 10.1145/3514221.3526148. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3514221.3526148.

[19] Shadi A. Noghabi et al. "Samza: stateful scalable stream processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), 1634–1645. ISSN: 2150-8097. DOI: 10.14778/3137765.3137770. URL: https://doi-org.tudelft.idm.oclc.org/10.14778/3137765.3137770.

[20] Kyriakos Psarakis et al. *Styx: Transactional Stateful Functions on Streaming Dataflows*. 2025. arXiv: 2312.06893 [cs.DC]. URL: https://arxiv.org/abs/2312.06893.

[21] Carlo Puliafito et al. "Stateful Function as a Service at the Edge". In: *Computer* 55.9 (Sept. 2022), 54–64. ISSN: 1558-0814. DOI: 10.1109/mc.2021.3138690. URL: http://dx.doi.org/10.1109/MC.2021.3138690.

[22] *py-linq*. https://pypi.org/project/py-linq/.

[23] Jim Verheijde et al. "S-QUERY: Opening the Black Box of Internal Stream Processor State". In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 1314–1327. DOI: 10.1109/ICDE53745.2022.00103.

[24] Jianying Wang et al. "PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba". In: *Proc. ACM Manag. Data* 1.2 (June 2023). DOI: 10.1145/3589785. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3589785.

[25] Chao Zhang et al. "HTAP Databases: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 36.11 (2024), pp. 6410–6429. DOI: 10.1109/TKDE.2024.3389693.