Global-State Querying in Stream Processing using Snapshots

Author:
Smruti KSHIRSAGAR

Supervisor:
Dr. Asterios KATSIFODIMOS

MASTER'S THESIS

submitted in partial fulfillment of the requirements
for the degree

MASTER OF SCIENCE, COMPUTER SCIENCE

from

TU Delft

in the

Data-Intensive Systems Research Group
Faculty EEMCS
Delft University of Technology
Delft, Netherlands

defended on

17th July, 2025

Student Number:	5938643
Project Duration:	November 2024 – July 2025
Thesis Committee:	Dr. Asterios Katsifodimos
	Dr. Georgios Iosifidis
	Dr. Burcu Özkan
Daily Supervisors:	Dr. George Christodoulou
	Kyriakos Psarakis

Declaration of Authorship

I, Smruti KSHIRSAGAR, declare that this thesis titled, "Global-State Querying in Stream Processing using Snapshots" 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: SK

Date: 10/07/2025

Abstract

Global-State Querying in Stream Processing using Snapshots

Stateful Functions-as-a-Service (SFaaS) platforms, such as Styx, are emerging as powerful abstractions for building distributed, serverless cloud applications. By combining the abilities of FaaS with strong transactional guarantees, they enable complex, stateful workflows without requiring developers to manage infrastructure. However, they lack built-in support for analytical queries across distributed function state. This thesis addresses that gap by proposing H-Styx, whose hybrid architecture extends Styx with a snapshot-based Query Engine, enabling near-real-time OLAP queries over global state while maintaining performance isolation for transactions. The Query Engine integrates seamlessly into the Styx architecture, leveraging periodic snapshots transmitted via a loosely-coupled, asynchronous interface. It ingests partitioned state from object store MinIO into columnar database DuckDB, supports incremental delta loads, and delivers results over a Kafka-based interface to achieve scalable, low-latency analytical querying while employing robust fault tolerance.

Empirical evaluation demonstrates that H-Styx preserves transactional throughput and latency under hybrid workloads, while significantly outperforming a baseline HTAP architecture (Postgres with Streaming Replication) on analytical throughput and providing superior workload isolation. These results validate the feasibility of supporting hybrid transactional and analytical processing in SFaaS environments. Overall, H-Styx bridges a crucial capability gap in SFaaS, enabling more powerful data-driven applications in distributed, event-driven architectures.

Acknowledgements

This thesis marks the conclusion of my Master's studies at TU Delft, as well as the closing chapter of my student life. I have thoroughly enjoyed working on this project, exploring the underlying technologies at the intersection of data-intensive systems and cloud computing, fields that have long fascinated me. Having the opportunity to contribute to Styx, a relevant platform with strong potential for the future of distributed serverless computing, and to work with emerging, impactful frameworks, has been deeply rewarding. I take great pride in what I have accomplished through this work.

This achievement has been far from a solo undertaking. I would like to extend my sincere gratitude to my thesis advisor, Dr. Asterios Katsifodimos, for granting me the opportunity to work on a project that so perfectly aligned with my interests, and for his invaluable guidance and feedback throughout the process. I am also deeply grateful to Kyriakos Psarakis and George Christodoulou for their continuous support, approachability, and insightful direction, which were essential in keeping my work focused and streamlined. This project would not have been possible without their contributions.

As the last member to join the circle of Master's graduates among my nearest and dearest, I wish to take a moment to express my appreciation to them all. My heartfelt thanks go to my parents, whose unwavering faith gave me the confidence to pursue and achieve this goal; and to my sister, for always being my trusted counselor. I am especially grateful to my three pillars from afar: Saachi, for the wry humour and wellness check-ins; Meghana, for being my enthusiastic and unconditional cheerleader; and Ritwik, for being my source of mental fortitude. To all my friends and extended family, who have offered nothing but encouragement along the way, I am deeply thankful. Finally, to the *vrienden* I made here in Delft, thank you for the countless memories and for making this chapter of my life so much easier and more joyful.

This has been an unforgettable journey, from navigating life abroad to immersing myself in cutting-edge research and technological innovation. I will always look back on it with great fondness and take away a lifetime of learning.

Contents

List of Figures

List of Tables

List of Abbreviations

FaaS	**F**unctions **a**s **a** **S**ervice
HTAP	**H**ybrid **T**ransaction/**A**nalytical **P**rocessing
MVCC	**M**ulti-**V**ersion **C**oncurrency **C**ontrol
SFaaS	**S**tateful **F**unctions **a**s **a** **S**ervice
2PC	**2** **P**hase **C**ommit

Chapter 1

Introduction

Modern applications increasingly rely on distributed, event-driven architectures [33], microservices, and streaming dataflows [14, 12] to meet the growing demands for elasticity, scalability, and fault tolerance in dynamic cloud environments [11, 30]. These paradigms enable real-time responsiveness, promote modular design, and allow individual components to scale independently. Frameworks based on serverless computing and Function-as-a-Service (FaaS) [21] have further accelerated this shift by abstracting away infrastructure concerns and offering a lightweight, event-driven programming model focused solely on application logic. This approach allows developers to compose large-scale systems from stateless functions [49] that process a wide range of events, simplifying deployment, scaling, and system maintenance in cloud-native environments.

Traditional FaaS platforms are inherently stateless. Due to this design, they fail to meet the requirements of complex workflows that involve stateful coordination across multiple function invocations, fine-grained consistency guarantees, or long-running transactions [39]. To address this, the paradigm of Stateful Functions-as-a-Service (SFaaS) has emerged. SFaaS allows developers to define functions that can retain local, durable state while still benefiting from the serverless deployment model. These functions can coordinate with each other in complex workflows while preserving consistency properties traditionally guaranteed by OLTP systems [58, 25].

Styx emerges as state-of-the-art among the SFaaS solutions. By adopting a deterministic, streaming dataflow execution model, Styx achieves serializable transaction semantics across distributed stateful functions while maintaining near-linear scalability [40]. In Styx, state changes are driven by event streams of transactions, and consistent snapshots are asynchronously created for recovery and replay, bypassing the blocking problems of classic 2PC [2]. This streaming model fits naturally with event-driven architectures and allows state to be partitioned, parallelized, and recovered with minimal coordination overhead [10]. Styx outperforms other SFaaS solutions, such as Beldi [58] and Boki [25], in terms of both latency and throughput.

While Styx excels at OLTP-style workloads, it lacks built-in support for analytical queries across multiple entities and transactions, limiting its utility for real-world applications due to the opaque nature of function state. Bridging this gap requires rethinking how distributed function state can be exposed, queried, and analyzed without compromising the system's transactional guarantees. This challenge motivates the core research problem addressed in this thesis.

1.1 Problem Statement

Despite its strengths in transactional processing, Styx leaves a critical gap in functionality – it does not support querying over distributed function state. Each function manages its state in isolation, which makes it difficult, if not impossible, to obtain a global, consistent view of the system or to run analytical queries across the function network [35]. This limitation becomes especially problematic in real-world use cases that require monitoring, observability, reporting, or mixed analytical processing workloads. Although Styx periodically persists snapshots for fault tolerance, these are stored in a binary format meant for recovery, not analysis. There is no built-in mechanism to access or transform these snapshots into a queryable format. Unlike traditional row- or column-store databases, which use query plans to efficiently access and process data [22, 31], Styx lacks indexing, projection, and filtering mechanisms over its state.

This challenge is closely related to the one addressed by Hybrid Transactional and Analytical Processing (HTAP) systems [38], which aim to efficiently combine real-time transactional updates with powerful analytical capabilities [57]. HTAP systems rely on architectures that isolate analytical workloads from transactional ones while maintaining fresh data access. Bringing this paradigm to SFaaS systems like Styx is particularly difficult due to the distributed, encapsulated nature of function state and the lack of structured query interfaces.

This thesis addresses the above challenge in the Styx system by extending its snapshotting mechanism using HTAP principles to support analytical querying within the SFaaS runtime. A dedicated analytical module can operate independently of the transactional path, ensuring workload isolation and minimal interference with active transactions. By introducing built-in queryability into Styx, the extended system, H-Styx, can serve hybrid workloads across domains such as banking, e-commerce, and workflow observability, where both strong consistency and analytical insights are critical [8]. The solution bridges the gap between the flexibility of serverless architectures and the analytical power of database systems, effectively turning SFaaS into a viable platform for hybrid workloads.

1.2 Research Questions

In this thesis, we build an analytical framework for Styx and evaluate its performance. By integrating analytical capabilities into a transactional SFaaS platform, we aim to explore the feasibility of unified OLTP and OLAP processing within event-driven, serverless architectures. We aim to answer the following research questions:

- RQ1: Can periodic blob store snapshots be leveraged without using traditional ETL to query the global distributed state of a streaming system while ensuring minimal impact on ongoing transactional processing?
- RQ2: What is the impact of integrating a snapshot-based query engine on the transactional throughput, latency, and consistency of the underlying SFaaS system?
- RQ3: How does the proposed hybrid architecture compare to a baseline HTAP system (Postgres with Streaming Replication) in terms of workload isolation, data freshness, and both transactional and analytical throughput and latency?

1.3 Contributions

This work proposes a hybrid architecture that enables distributed function state in SFaaS platforms to be queried in a consistent, low-latency, and scalable manner. Drawing on lessons from HTAP systems, delta stores, and cloud-native data lake designs, this thesis extends Styx with a lightweight mechanism for OLAP-style analytics over transactional state.

Specifically, we present **H-Styx** (Hybrid Styx), a hybrid architecture that integrates a snapshot-based Query Engine into Styx, enabling near-real-time analytical queries over distributed function state. The design preserves Styx's core transactional properties, including strict consistency and workload isolation, while introducing minimal overhead. It introduces a delta store architecture that enables near real-time queries over the distributed function state, without compromising Styx's transactional guarantees or requiring external data infrastructure. The delta store maintains an incremental log of state changes between snapshots, similar to modern data lake systems such as Delta Lake [4]. This design enables efficient access to both recent and historical states without the overhead of reconstructing full snapshots at query time. To support analytics, this delta store exposes a columnar interface that enables OLAP-style operations such as filtering, aggregation, and joins over distributed state fragments [27].

Experimental results confirm the feasibility and effectiveness of this approach for hybrid workloads in event-driven serverless environments. H-Styx demonstrates that the benefits of HTAP can be practically extended to SFaaS platforms, closing the gap between distributed transactional functions and powerful analytical processing. This enables a unified platform for modern, data-intensive applications.

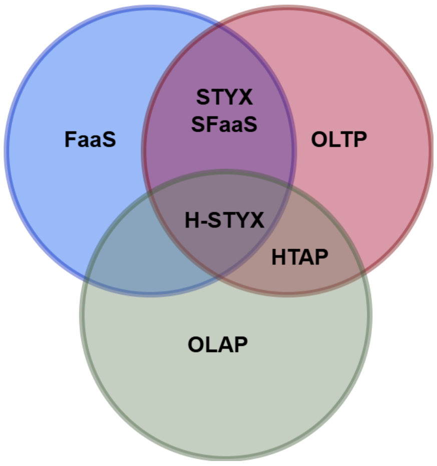


FIGURE 1.1: H-Styx at the intersection of three systems

The code for H-Styx can be found here.

1.4 Structure of Report

The remainder of this report is structured as follows. Chapter 2 discusses the background on Styx and the theoretical foundations of HTAP systems. Chapter 3 presents the architecture of the proposed solution. The experimental setup used to evaluate H-Styx is described in Chapter 4, and the results of these experiments are presented in Chapter 5. An in-depth discussion and analysis of the results, along with identified limitations, are provided in Chapter 6. Finally, conclusions and directions for future work are summarized in Chapter 7.

Chapter 2

Background

This chapter reviews the existing literature on SFaaS systems, focusing on Styx [40], as well as HTAP architectures. It discusses the parallels between OLTP and SFaaS and explores how the principles of HTAP can be applied within an SFaaS system to enable near real-time querying capabilities.

2.1 Stateful Functions-as-a-Service

Stateful Functions-as-a-Service aims to serve use cases involving strict consistency within low-latency workflows and microservices architecture [1]. SFaaS defines functions that have the ability to retain their own state and also call other functions. This differs from traditional FaaS, which is extremely popular in serverless computing [5], where each function call is run in a new runtime environment, and persistence is only available through external storage solutions such as databases. The disaggregation of storage and compute resources introduces significant latency and network overhead. It also restricts the ability of FaaS to serve concurrent, multi-step workflows requiring stateful coordination [37]. Orchestration services like Step Functions [7] provide limited coordination for FaaS like Lambda Functions [6], but latency is still a problem, as they suffer from high startup latency due to on-demand instantiation of function environments [49]. These limitations can be addressed in SFaaS [54, 3].

Several systems have attempted to implement SFaaS with varying degrees of success. Some of these solutions in literature are Crucial [9], Cloudburst [48], Beldi [58], and Boki [25]. Crucial provides strong consistency via a distributed shared object layer and supports fine-grained synchronization, but lacks functionalities like rollback or transactional isolation. Similarly, Cloudburst with HydroCache [54] provides causal consistency by forming a DAG, but does not isolate the DAG workflows. Beldi and Boki provide ACID properties aligning with our use case, but suffer from high latency due to the utilization of costly Two-Phase Commit [2]. Styx has been shown to outperform them in terms of both latency and throughput, alongside exhibiting near-linear scalability.

2.2 Styx: Distributed Consistent Transactions

Styx [40] enables OLTP-style workloads by executing distributed stateful functions with end-to-end serializability and exactly-once guarantees. Unlike traditional OLTP systems, Styx co-locates computation with in-memory state and uses a deterministic execution model based on streaming dataflows. This architecture eliminates the need for external databases, making it a modern, cloud-native approach to transactional processing. Table 2.1 shows how Styx compares to OLTP systems.

2.2.1 Architecture Overview

The Styx deployment consists of a coordinator and a configurable number of workers that are managed by it. The coordinator manages and monitors the workers and the runtime

TABLE 2.1: Comparison of Styx to OLTP systems

Feature	Traditional OLTP Systems	Styx
Execution model	Thread-based [28]	Streaming dataflows with stateful function call-graphs
Transaction model	Serializable or snapshot isolation with 2PC	Serializable transactions via deterministic protocol
ACID Guarantees	Yes	Yes
Concurrency control	Locking or timestamp-based	Deterministic sequencing with optimistic locks
Scalability	Limited by central coordination [17]	Almost linear due to horizontal scale-out
Programming Model	SQL, stored procedures, APIs	Async or sync Python API

state of the cluster. It also handles fault tolerance. The workers encapsulate the logic of low-level primitives such as locking, transaction coordination, retries, and idempotency checks, ensuring that users do not need to handle this in their code. Transactions are ingested in the form of Kafka messages and sequenced. Kafka [23] contributes to the durability of Styx, as it deterministically replays messages even in the event of a failure. Incoming transactions are partitioned among workers that utilize coroutines for asynchronous processing, resulting in low latency.

Users interact with Styx through the `Styx` package, specifically via the `StyxClient` class and its methods. By creating an instance of this class, users can perform various operations, such as initializing the Styx system, loading data, submitting a dataflow graph, and executing transactions. The client internally manages the network with Styx infrastructure for running these methods, such as converting transaction requests to Kafka messages.

The key terms within the Styx programming model that will be used in our project are as follows:

- **Entity and State:** An entity is analogous to an object in object-oriented programming. Its state consists of a unique key and a set of attribute values. Each entity is isolated and has access only to its own state.
- **Function:** A function mutates the state of an entity during a transaction. An entity can have multiple functions across the different transaction types.
- **Operator:** An operator is a node in the dataflow graph that executes a function on an entity, thereby changing the state of the entity. Each operator maintains the state of its entity and provides access to its functions.
- **Dataflow Graph:** The dataflow (or stateflow) graph contains all operators and their relations based on the defined functions and transactions. It captures the structure of computation and state transitions across entities. Figure 4.1 shows the dataflow graph created for our experiment.
- **Transaction:** A transaction begins with the invocation of an entry function on an entity and may involve subsequent function calls across entities, forming an arbitrary call graph that defines the control and data flow of that transaction.

2.2.2 Snapshotting Mechanism

Incremental snapshots are taken at the end of each transactional epoch on a per-worker basis, and together they represent a consistent global state across all operators. Styx asynchronously takes these snapshots, leading to negligible impact on transactional throughput. These snapshots are persisted to MinIO object store in a binary serialized format. Each worker writes the state of entities modified by transactions to MinIO for every operator.

2.2.3 Limitations

Although Styx can replay snapshots and utilize the data within transactions, the isolation of each entity makes it impossible to run analytical queries on the global state or view the state of multiple entities simultaneously. This leaves room for building an analytical framework that enables querying capabilities, allowing us to run aggregations, joins, and other OLAP queries on the transactional data, making this black box more transparent.

2.3 Hybrid Transactional and Analytical Processing

Hybrid Transactional and Analytical Processing (HTAP) describes systems designed to unify transactional (OLTP) and analytical (OLAP) workloads within a single database engine [38]. These systems aim to avoid the traditional separation of transaction processing and analytics, enabling low-latency analytical insights over fresh transactional data. By combining the strengths of row-based storage (for fast transactional writes) with columnar storage (for efficient analytical scans), HTAP systems achieve hybrid performance on mixed workloads [29].

Integrating these two workload types introduces several challenges:

- Data synchronization: Ensuring timely propagation of transactional updates to analytical views with minimal overhead is difficult, yet essential for maintaining freshness [57].
- Data consistency: Analytical queries must operate on a consistent version of data while transactional workloads continue to update it.
- Gap of write efficiency: Transforming row-oriented transactional data into columnar analytical layouts (e.g., via log replay or delta merge) can be costly, increasing latency and reducing analytical freshness.
- Performance Isolation: Running OLTP and OLAP workloads on the same infrastructure risks resource contention, making it hard to guarantee that neither workload starves the other [46].

2.3.1 HTAP Architectures

Traditional HTAP systems integrate a row store for transactions and a column store for analytics within a unified architecture [38]. This tight coupling reduces the need for external data movement, enabling low-latency access to recent transactional data for analytical queries. However, such integration introduces significant coordination overhead, making it difficult to maintain workload isolation. Synchronization mechanisms such as delta merges or log-based replay are typically used to keep the OLAP store in sync with the OLTP layer, but these can result in latency spikes and temporarily stale query results when updates lag behind.

Cloud-native HTAP systems leverage disaggregated storage and compute, allowing separate scaling for transactional and analytical workloads [19]. Features such as log-based transaction persistence and serverless query execution improve elasticity and fault tolerance, while exposing near-real-time snapshots for analytical queries without blocking OLTP workloads.

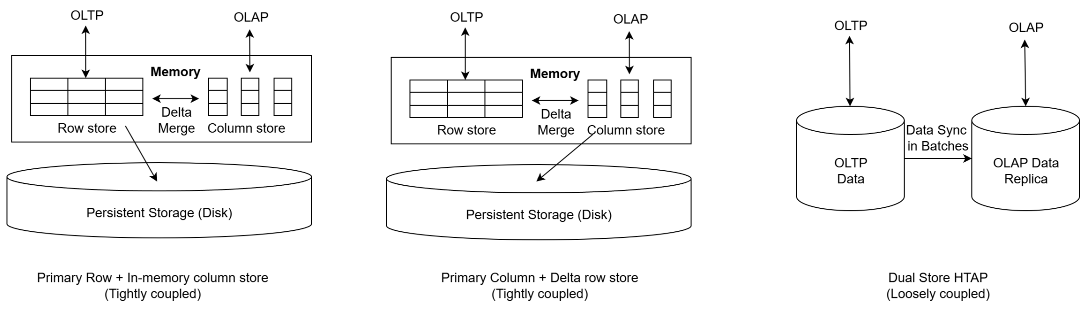

OLTP
OLAP
Memory
Delta Merge
Row store
Column store
Persistent Storage (Disk)
Primary Row + In-memory column store (Tightly coupled)
OLTP
OLAP
Memory
Delta Merge
Row store
Column store
Persistent Storage (Disk)
Primary Column + Delta row store (Tightly coupled)
OLTP
OLAP
OLTP Data
Data Sync in Batches
OLAP Data Replica
Dual Store HTAP (Loosely coupled)

FIGURE 2.1: HTAP Architecture Patterns

The core trade-off in the design of an HTAP system is between performance isolation and data freshness. Tightly coupled designs (e.g., primary row store with an integrated column store, or vice versa) offer more up-to-date analytical views but risk interference with transactional workloads, especially under high contention [47]. Loosely coupled dual-store architectures, on the other hand, improve isolation by separating the two execution paths entirely, where OLTP and OLAP operate on different data views and infrastructure layers, often with asynchronous synchronization [29]. While this can introduce some analytical staleness, it provides stronger guarantees around transactional latency and throughput, which is especially important in distributed, event-driven systems like SFaaS.

The dual-store model is therefore particularly well-suited to the serverless and streaming nature of platforms like Styx. It aligns with cloud-native principles such as disaggregated compute and storage, enabling independent scaling of transactional and analytical workloads. By applying a Copy-on-Write mechanism to incremental snapshot deltas [47], H-Styx can expose analytical views over consistent, recent state without locking or delaying ongoing transactions. These snapshots are inherently consistent due to Styx's deterministic execution model, making them an ideal basis for OLAP-style query engines.

2.3.2 Benchmarks

Benchmarking HTAP systems is challenging, as it is difficult to define a single unifying metric that captures the diverse capabilities of hybrid workloads [26]. Table 2.2 summarizes key HTAP benchmarks from the literature, namely CH-BenCHmark [16], HTAPBench [15], HATtrick [34], and HyBench [55].

TABLE 2.2: HTAP Benchmarks

Benchmark	Transactions	Queries	Metrics
CH-BenCHmark	TPC-C	modified TPC-H	Referenced throughput
HTAPBench	TPC-C	modified TPC-H	Referenced throughput
HATtrick	Custom	SSB	Throughput frontier, freshness
HyBench	Custom	Custom	Hybrid score, freshness score

CH-BenCHmark and HTAPBench evaluate mixed workloads by combining TPC-C [28] transactions with adapted TPC-H queries, focusing on total throughput within a single system but lacking comparative performance metrics across systems. HyBench defines custom hybrid workloads with both transactional and analytical characteristics, introducing a unified hybrid score but based on a synthetic schema, which does not generalize well to delta store architectures due to the introduction of analytical transactions.

HATtrick, in contrast, extends the SSB [36] schema with two read-write transactions, and introduces the *throughput frontier* metric to thoroughly analyze workload isolation and performance trade-offs. Additionally, HATtrick defines a freshness metric, a critical aspect of HTAP systems.

2.3.3 Limitations

The discourse on HTAP presents certain gaps that provide scope for a more sophisticated system. Traditional HTAP systems tightly couple transactional and analytical layers via synchronous mechanisms, causing latency spikes under mixed workloads due to blocking and coordination overhead [43]. Systems using row-to-column conversions introduce additional latency and stale analytical views, impacting freshness [29]. OLAP workloads can starve OLTP operations, as HTAP often lacks mechanisms to ensure isolation across workload types [56].

2.4 Styx as an HTAP System

Styx proposes a deterministic, asynchronous, and distributed execution model to address the fundamental limitations in current HTAP systems as mentioned above. Its transaction processing engine, combined with a snapshot-based storage layer, enables consistent, low-latency transactional guarantees. The asynchronous streaming dataflow foundation reduces coordination bottlenecks and prevents blocking of resources. This separation, while maintaining strong serializable semantics, is one of the core innovations that can allow Styx to serve hybrid workloads efficiently.

From this literature review, it is evident that Styx SFaaS represents a promising evolution of cloud-based OLTP systems. In the HTAP domain, Styx can align with a delta store architecture, where data is incrementally updated and exposed through near-real-time snapshots. Delta store patterns provide a flexible middle ground between pure OLTP row stores and columnar OLAP systems, enabling data synchronization and consistency without row-to-column conversion overheads, while also supporting strong workload isolation. Styx's snapshot store can be seen as a natural implementation of delta-store principles and can thereby support OLAP functionalities, resulting in a distributed HTAP system.

Chapter 3

Methodology

In the previous chapter, we introduced the concept of aggregating delta snapshots from the durable store to enable near-real-time data querying. Building on this, we design a Query Engine that provides an interface for executing analytical SQL queries over the entity states maintained in Styx. This chapter presents the proposed solution of H-Styx in detail, outlining the architecture and design of the Query Engine.

3.1 Design Decisions

The development of the Query Engine was a multi-step process. In this section, we discuss the design decisions that were made and the rationale behind them to implement the final solution.

3.1.1 Data Processing Framework

Staying consistent with Styx design, which provides co-location of state and data, a set of embedded analytical data processing frameworks were considered to serve the queries. These included DuckDB [42, 53], Pandas [32, 50], Polars [24] and PyArrow [41]. Table 3.1 compares these frameworks.

TABLE 3.1: In-process Analytical Frameworks

Feature	DuckDB	Pandas	Polars	PyArrow
Data Storage	SQL Tables	Dataframes	Dataframes	Columnar Dataframes
Handle Large Data	Out-of-core	In-memory only	Limited out-of-core	Out-of-core
Parallelism	Yes	No	Yes	Yes
Memory Efficiency	High	Low	High	Very High
Aggregations/Joins	Yes	Manual	Manual	Manual
SQL Support	Yes	No	Minimal	No
Optimized Querying	Yes	Memory-bound	Yes	External support

Based on this initial analysis, DuckDB was chosen as the framework for building the Query Engine. DuckDB supports micro-batch data updates, providing easy and familar SQL format for DDL and DML. It employs Multi-Version Concurrency Control (MVCC) to allow concurrent reads and writes, making it convenient to perform frequent delta loads while simultaneously running read queries on consistent data. This ensures low-latency reads and writes, along with ACID guarantees, unlike traditional OLAP databases. Additionally, it provides Python API, allowing ease of use. These features make DuckDB extremely suitable for our Query Engine.

3.1.2 Data Storage

DuckDB supports ad-hoc querying directly on files in various formats such as CSV, Parquet, and JSON, including those stored externally in file systems like MinIO. This capability enabled consideration of two possible architectural approaches:

- Performing delta loads and writing the results back as queryable Parquet files [51] in MinIO, which DuckDB could then read on demand; or
- Creating DuckDB tables and persisting them internally as a `.db` file within the Query Engine component.

The first approach restricts access to many of DuckDB's advanced OLAP capabilities, such as caching, indexing, and constraint enforcement. Consequently, the second approach was selected. A key motivation for this decision was the ability to enforce primary key constraints and MVCC, which are essential for maintaining data correctness and consistency across delta loads. The primary key column(s) correspond to the unique entity key in Styx. This architecture supports nested column structures, enabling efficient querying over complex data types.

3.1.3 Loading Binary Snapshots into DuckDB

The Query Engine processes binary snapshots retrieved from MinIO entirely in-memory before loading them into DuckDB. DuckDB's Python API supports multiple input formats for data ingestion, including Pandas DataFrames, Polars DataFrames, and NumPy arrays. Since the ingestion process involves typecasting and schema mapping, DataFrame-based formats were preferred due to their structured columnar representation.

To determine the most efficient DataFrame library for this workload, a microbenchmark was conducted comparing the time required to construct a DataFrame from deserialized snapshot data using both Pandas and Polars. The benchmark tested various row counts, representative of actual snapshot sizes encountered during typical operations. The results are presented in Table 3.2.

TABLE 3.2: Time taken for DataFrame Creation

Number of Rows	Pandas	Polars
10000	0.02s	0.02s
50000	0.2s	0.11s
100000	0.5s	0.4s
1500000 (init load)	22.11s	4.65s

Based on this performance comparison, Polars was chosen for DataFrame construction due to its consistently lower latency across all scales, particularly in the initialization case. Within the Query Engine, binary snapshot partitions are aggregated per operator. Since each operator corresponds to a table in the analytical schema, a separate DataFrame is created per table during each delta load. Once DataFrames for all tables have been constructed, the data loading process into DuckDB begins.

The data load into DuckDB is performed via serial inserts using a database cursor. Data is upserted into all applicable tables (i.e., operators present in the snapshot), and the cursor is only committed after all insertions have completed. This approach guarantees read consistency for queries and enables robust failure handling, which is discussed further in Section 3.5.

3.1.4 Schema Inference

Automatic data type inference is supported by both Polars and DuckDB, making it convenient for simple datasets that contain a single key and lack complex structures. However, in cases where the data includes composite keys, automatic inference becomes challenging, as unpacking and interpreting such keys correctly is not straightforward. In Styx, data is stored in a key-value format using Python dictionaries, where keys are strings. For composite keys, multiple components are concatenated into a single string using a delimiter to ensure the uniqueness of the entity. To simplify query writing and improve clarity, the schema explicitly uses user-defined column names and keys. This approach allows users to reference meaningful column identifiers in their queries and add constraints, which is why it was chosen over relying solely on automatic inference.

3.2 Architecture of Query Engine

Based on the design choices discussed above, the Styx architecture was extended to integrate the Query Engine component. The updated high-level system architecture is illustrated in Figure 3.1. To enable this integration, only minor modifications were required in the existing Styx components, specifically within the Styx Coordinator. These changes involved two key enhancements: first, when the Coordinator receives a Stateflow Graph during system start-up from the client, it transmits the initialized data model to the Query Engine; second, it sends notification to the Query Engine upon the successful completion of each snapshot process. These notifications allow the Query Engine to remain synchronized with the latest global state, ensuring that analytical queries are executed on fresh and consistent data. The Query Engine was added into the existing Styx network, which utilizes TCP socket connection for internal communication, listening for messages from the Coordinator. Simultaneously, it listens for queries from the user on a Kafka topic using a Kafka consumer, and provides query results on another Kafka topic, for the client to consume. Thus, the Query Engine continuously listens for internal and external communication by asynchronously switching between these two message consumption tasks.

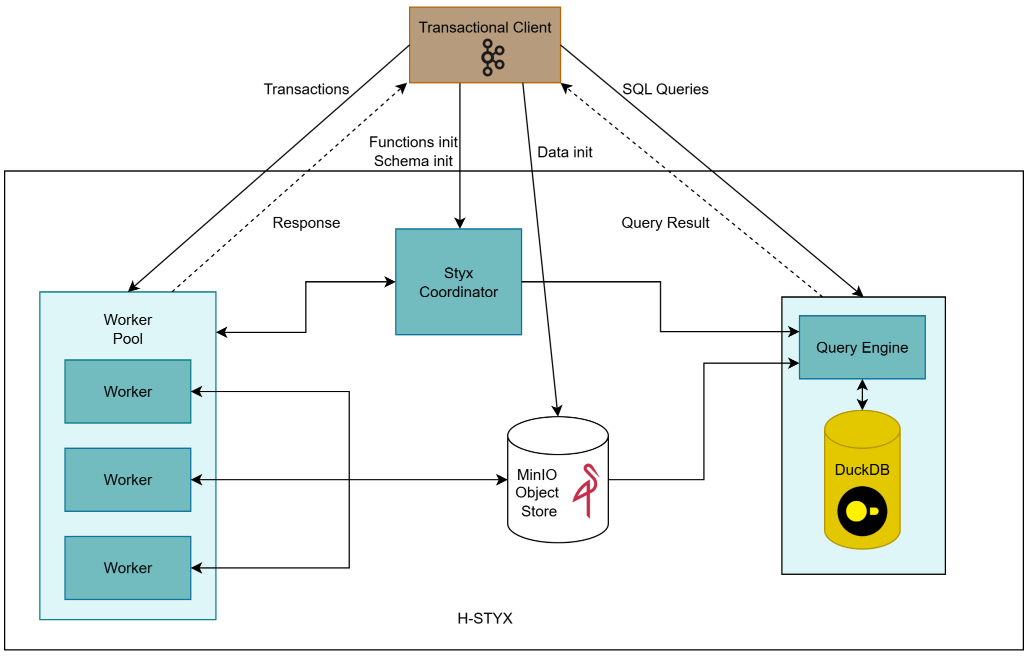


FIGURE 3.1: Styx Architecture with Query Engine

3.3 Query Engine Workflow

The `StyxClient` was enhanced to handle incoming analytical queries.

The Query Engine's workflow is logically divided into three main stages, each responsible for a distinct part of the query execution process.

3.3.1 Initialization

During Styx initialization, the client bulk loads data to MinIO. After this, the client submits the stateflow graph, which includes user-defined functions attached to the operator definitions. In this process, an additional attribute is provided to each operator to define its corresponding analytical schema. Since each operator represents a logical entity, it is appropriate to specify the schema at the operator level. A schema definition sample is shown in 3.1.

LISTING 3.1: Example Schema Declaration

```
ycsb_operator.set_analytical_schema([
    {
        "column_name": "id",
        "data_type": "BIGINT",
        "primary_key": True
    },
    {
        "column_name": "value",
        "data_type": "BIGINT",
        "nullable": False,
    }
])
```

`StyxClient` sends this stateflow graph to the Coordinator. Then the Query Engine receives the initialized data model from the Styx Coordinator. The schema definition allows for the following configurations:

- Data Type: SQL data type of the column
- Primary Key: Boolean value to determine whether a column is a primary key. In case of composite keys, it should be set to `True` for all such columns. It is `False` by default.
- Nullable: Boolean value to add `NOT NULL` constraint.
- Datetime Format: Styx stores all data as strings; therefore, this configuration allows the creation of columns with appropriate types such as `datetime` or `timestamp`, which are essential for analytical queries.
- Nested Columns: In cases where the user wishes to normalize complex data by breaking it into smaller, normalized tables, this setting enables the creation of separate tables for nested columns, with each maintaining a reference to the original parent table.

Once the schema is defined in the Query Engine, it proceeds to load the initial snapshots generated during the bulk initialization phase into DuckDB. This process utilizes chunking to efficiently handle large volumes of data. According to DuckDB best practices, constraints and indexes should be applied only after the bulk data load is complete to optimize performance [20]; this approach is followed here as well. After all data has been loaded, the necessary indexes are created, which completes the initialization process. This step is illustrated in Figure 3.2.

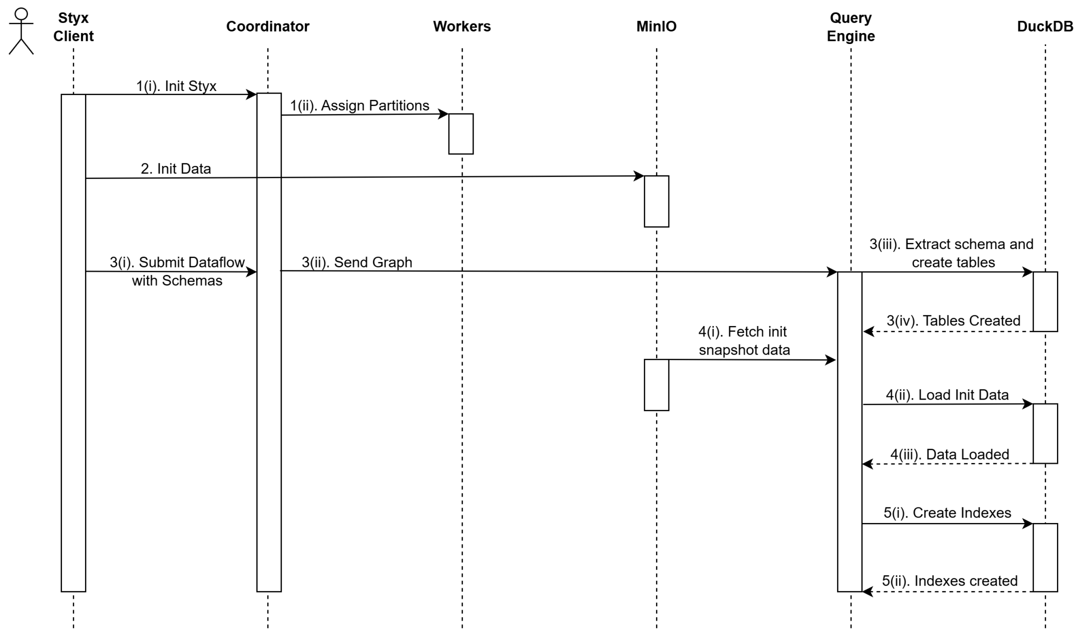


FIGURE 3.2: Init Workflow in Query Engine

3.3.2 Delta Load

Styx workers periodically persist the state of their respective operator partitions to durable storage in MinIO. Once the Coordinator receives confirmation that all workers have successfully committed their snapshots, it notifies the Query Engine by sending the corresponding snapshot ID. The Query Engine then collects snapshot files from all operator partitions associated with that snapshot and aggregates them by operator. The data is deserialized and converted into dataframes, which are subsequently ingested into DuckDB.

If a primary key constraint is defined for a table, the engine performs upserts based on this key; otherwise, it performs simple inserts. The snapshot load is only committed after data from every operator partition has been fully loaded into the corresponding DuckDB tables. Due to DuckDB's MVCC support, any queries executed by the Query Engine during this loading phase will run on the previous snapshot version ensuring consistent reads. This step is shown in Figure 3.3.

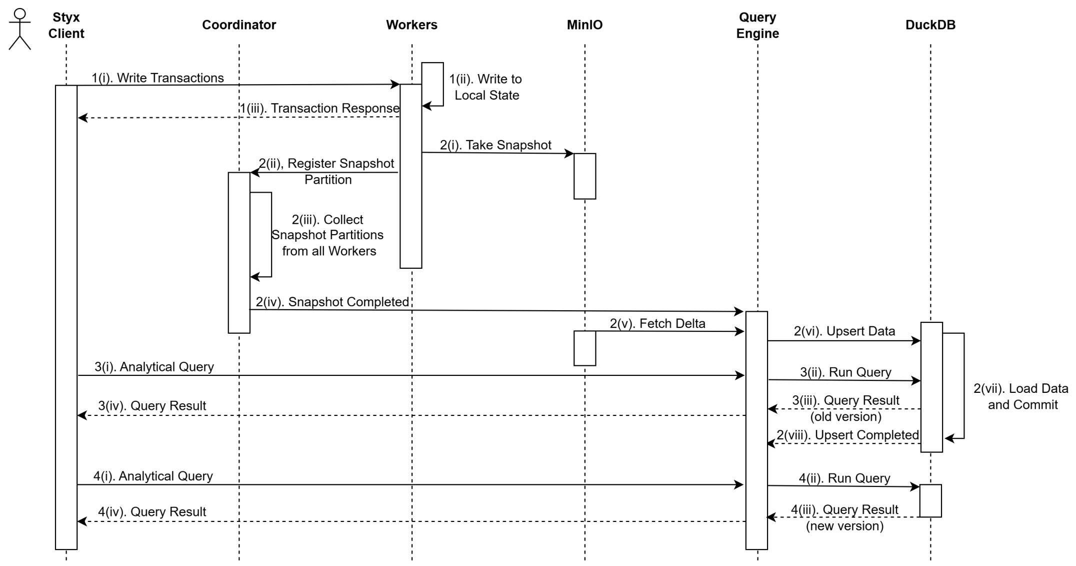

Styx Client
Coordinator
Workers
MinIO
Query Engine
DuckDB
1(i). Write Transactions
1(ii). Write to Local State
1(iii). Transaction Response
2(i). Take Snapshot
2(ii), Register Snapshot Partition
2(iii). Collect Snapshot Partitions from all Workers
2(iv). Snapshot Completed
2(v). Fetch Delta
2(vi). Upsert Data
2(vii). Load Data and Commit
3(i). Analytical Query
3(ii). Run Query
3(iii). Query Result (old version)
3(iv). Query Result
2(viii). Upsert Completed
4(i). Analytical Query
4(ii). Run Query
4(iii). Query Result (new version)
4(iv). Query Result

FIGURE 3.3: Read-Write Workflow in Query Engine

3.3.3 Handling Client Queries

`StyxClient` contains a function `send_event()` which serves as a function call for a particular entity, thus performing a transaction for that entity. It returns an object of `StyxFuture` class, which contains a unique request ID. The Kafka consumer on the user side can map the transactions to their eventual outputs using this request ID. We employed a similar mechanism for queries. We created a `send_query()` method that accepts a SQL query as input, and returns a `StyxFuture` instance, with a unique query ID which can then be mapped to the query result. Query Engine introduces concurrency to serve multiple read queries in parallel, reducing read latency. Transaction outputs are published by Styx to dedicated Kafka operator output topics (named with the `--OUT` suffix) . Similarly, all query responses produced by the Query Engine are sent to a single Kafka topic named `query-engine--OUT`.

3.4 Fault Tolerance

The Query Engine is designed to be fault tolerant. Both the DuckDB database file and the stateflow graph received from the coordinator are persisted on a mounted volume, ensuring durability across restarts. To track progress, a dedicated `RECOVERY` table is maintained within DuckDB. This table stores the last committed snapshot ID. When the Query Engine receives a new snapshot ID from the coordinator, it compares it against the recorded value. In the event of a failure and subsequent restart, the Query Engine reinitializes using the locally persisted stateflow graph. Upon receiving a new snapshot ID, it determines whether any intermediate snapshots were missed by comparing it to the last known committed ID. Since the snapshot IDs are monotonically increasing, Query Engine can detect gaps and load missing snapshots in order. This recovery mechanism guarantees that the DuckDB state becomes eventually consistent with the snapshot store, without data loss.

Furthermore, queries sent by clients are published to a Kafka queue. If the Query Engine goes down before consuming these messages, they remain in the queue. Upon restart, the engine resumes consumption, ensuring that no client queries are lost during downtime.

Thus, this architecture provides end-to-end fault tolerance in both state recovery and query handling, as illustrated in Figure 3.4.

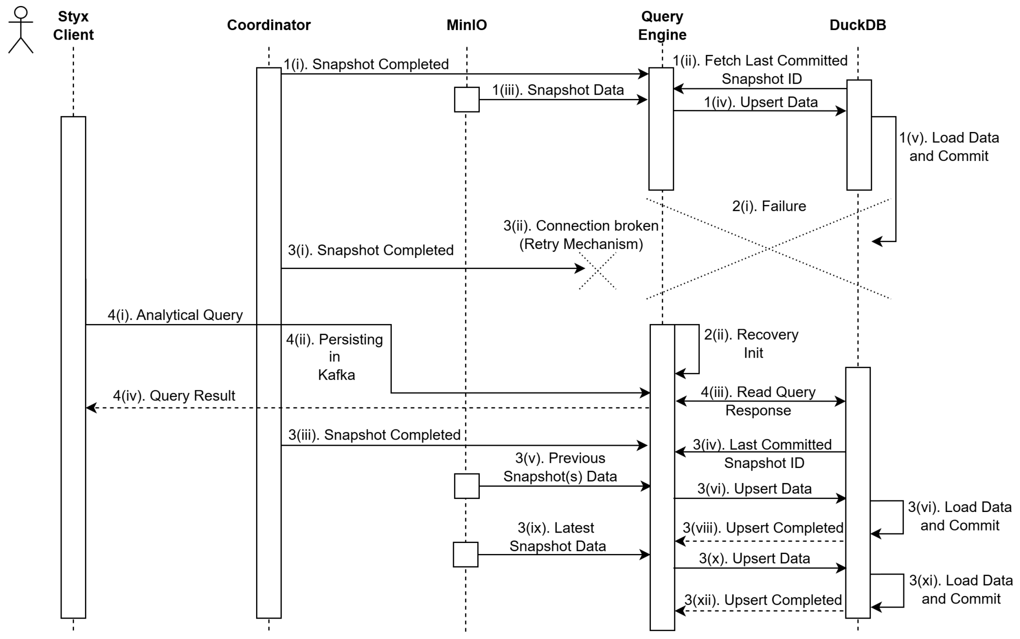

Styx Client
Coordinator
MinIO
Query Engine
DuckDB
1(i). Snapshot Completed
1(ii). Fetch Last Committed Snapshot ID
1(iii). Snapshot Data
1(iv). Upsert Data
1(v). Load Data and Commit
2(i). Failure
3(ii). Connection broken (Retry Mechanism)
3(i). Snapshot Completed
4(i). Analytical Query
4(ii). Persisting in Kafka
2(ii). Recovery Init
4(iv). Query Result
4(iii). Read Query Response
3(iii). Snapshot Completed
3(iv). Last Committed Snapshot ID
3(v). Previous Snapshot(s) Data
3(vi). Upsert Data
3(vi). Load Data and Commit
3(ix). Latest Snapshot Data
3(viii). Upsert Completed
3(x). Upsert Data
3(xi). Load Data and Commit
3(xii). Upsert Completed

FIGURE 3.4: Fault Tolerance in Query Engine

3.5 H-Styx

As described in this chapter, the Query Engine is designed to remain logically and physically decoupled from the core Styx architecture. It operates on dedicated hardware with resource isolation, ensuring that all analytical processing and snapshot ingestion occur independently of the transactional (OLTP) data path. This separation maintains isolation of the two workloads.

This Query Engine architecture that uses MinIO snapshots as its sole input, bypassing the need for traditional ETL pipelines, directly addresses RQ1. The snapshots represent the global distributed state of the streaming system at well-defined points in time. By directly consuming and incrementally applying these snapshots, the Query Engine enables efficient global state querying with minimal operational overhead and no disruption to ongoing transactional processing.

The integration of the Query Engine is entirely optional and controlled via a feature flag, allowing system operators to deploy Styx either as a pure transactional system or as a hybrid system that supports both transactional and analytical workloads. This configurability makes H-Styx suitable for a broad range of use cases while preserving system modularity and scalability.

Chapter 4

Experiments

In this chapter, we describe the experiments that were carried out to gain insight into RQ2 and RQ3. Both experiments were conducted on a Linux system (Ubuntu 22.04.5) with 128 CPU cores. The clients ran locally, whereas Styx was deployed on Docker with CPU limits.

4.1 Impact of Query Engine on Styx Transactions

In this experiment, we measure the effect of the newly added Query Engine component on the original architecture under comparable workloads. We perform non-inferiority testing between two systems: System A, which is Styx without the Query Engine, and System B, which is H-Styx.

4.1.1 Benchmark

A variant of the YCSB-T benchmark [18] was used. This benchmark was selected because it was originally used to evaluate Styx against other SFaaS systems [40]. The base dataset for this benchmark contains users and their associated credit amounts. Transactions performed on the data involve a single operation, `transfer`, which subtracts an amount from one user (the debtor) and adds it to another user (the creditor). The debtor is chosen randomly, while the creditor is selected according to a Zipfian distribution [44]. Transactions are sent continuously to Styx at a fixed input rate.

For H-Styx, we added an analytical workload based on standard SQL aggregation queries. Five queries were written, as listed in Appendix A. These queries are executed sequentially in a loop at runtime at a specified input query rate.

4.1.2 Metrics

We collected transactional metrics from Styx and H-Styx runs, which formed the basis for comparing the two systems. The metrics considered include latency, throughput, consistency, and the number of missed messages.

Mean latency is defined as:

$$Mean\_Latency = \frac{\Sigma(output\_response\_time - input\_request\_time)}{number\_of\_input\_requests}$$

In addition to the mean latency, the 95th percentile and 99th percentile latencies are also measured.

Throughput is defined as

$$Throughput = \frac{total\_output\_responses}{total\_time}$$

The output messages are filtered to remove any requests sent during the warm-up time.

TABLE 4.1: Non-inferiority margins

Metric	Test	Margin
Mean Latency	One-sided t-test	0.1 ms
95th Percentile Latency	Mann Whitney-U	0.1 ms
99th Percentile Latency	Mann Whitney-U	1 ms
Throughput	One-sided t-test	1 tps

Consistency is recorded as a true/false metric, evaluated by computing the total amount present in the system across all users. Since only transfer operations are performed, the total sum should remain identical before and after the execution of all transactions.

Finally, the number of missed messages is recorded by comparing input requests with output responses, ensuring that all transactions are completed successfully and that no messages are lost within the event-driven architecture.

The Styx coordinator and each worker are limited to one CPU core each, while the Query Engine in H-Styx uses an additional dedicated CPU core. These limits are enforced using Docker's resource restriction features to ensure consistent and equivalent hardware usage in both experimental runs.

4.1.3 Parameters

We configured a transactional `StyxClient` to generate 1000 transactions per second. For H-Styx, we introduced an additional client thread to submit queries at a rate of 20 queries per second. The Styx epoch size was set to 1000, number of worker nodes to 4, and each experiment ran for 60 seconds.

We varied two parameters: the dataset size and the snapshot interval. Experiments were conducted using datasets of 10,000 keys and 100,000 keys. Additionally, for the 10,000 key dataset, we tested with snapshot intervals of 10 seconds and 30 seconds. The former evaluates the impact of scaling, while the latter measures performance differences under increased snapshot frequency, resulting in more frequent delta loads.

4.1.4 Non-inferiority Testing

The goal of this experiment was to determine whether the addition of the Query Engine had a negative impact on the transactional performance of Styx. For this purpose, we performed non-inferiority tests [52], using the selected metrics to evaluate whether H-Styx showed a significant degradation in transaction processing. We define the statistical hypotheses as follows:

$$H_0 = \mu_{H-Styx} - \mu_{Styx} \geq \Delta$$

$$H_1 = \mu_{H-Styx} - \mu_{Styx} < \Delta$$

where μ represents any of the metrics and Δ represents the allowable margin for declaring non-inferiority. We defined a typical p-value threshold (α) of 0.05 [13]. The null hypothesis assumes that H-Styx is worse than Styx by at least the non-inferiority margin, while the alternative hypothesis states that H-Styx is no worse than Styx by more than the margin.

Based on pilot readings and design expectations, parametric or non-parametric tests were applied, and an acceptable margin is defined for each metric. This is shown in Table 4.1. For consistency and missed messages, the readings were reported directly since no variability was observed. All readings were found to be stable with a low standard deviation; hence, a sample size of 10 was chosen for the final results.

4.2 Comparison with an HTAP System

The second experiment measures how the transactional and analytical capabilities of H-Styx compare to an HTAP system. For this, we use the HATtrick benchmark discussed in Section 2.3.2.

4.2.1 Benchmark

The HATtrick benchmark [34] is a comprehensive framework for measuring the performance isolation and data freshness of HTAP systems. It first identifies the saturation points of the system in terms of transactional throughput and analytical throughput independently, and then evaluates hybrid performance under varying loads. HATtrick builds on the Star Schema Benchmark [36], which is a simplified, denormalized version of TPC-H [28]. SSB defines 13 analytical queries involving joins over multiple tables and aggregations as listed in Appendix B. These queries are executed by an analytical client sequentially in a loop, starting from a randomly chosen query.

HATtrick creates two transactions on the SSB schema.

- **New Order:** It is an insert transaction that creates a new order consisting of multiple line order entries, each of which has a part key. It iterates over each line order, fetches customer data, supplier data, date data, and part data, and calculates extended price, revenue, and total order price. Multiple rows are inserted into `LINE_ORDER` table, one for each part of the order.
- **Payment:** It is an update and insert transaction. It updates `CUSTOMER` table, updates `SUPPLIER` table and inserts new entry into `HISTORY` table.

A transaction client continuously sends these transactions with the same probability. The maximum throughput values are derived after gradually increasing the number of clients, thereby increasing input request rate, until it saturates.

4.2.2 Metrics

The HATtrick benchmark evaluates a system based on two metrics. The source code can be found here. We consider two additional metrics.

Throughput Frontier

HATtrick defines a throughput frontier as a two-dimensional graph representing the hybrid performance of a system under varying loads. In this graph, transactional throughput is plotted along the X-axis and analytical throughput along the Y-axis. The saturation throughputs under isolated workloads, denoted by X^T (transactional) and X^A (analytical), establish the bounds of this graph.

The input rates that yield these saturation throughputs are τ_{max} and α_{max} respectively. These input rates are divided into six segments each, resulting in 36 combinations of hybrid input rates. For each of these combinations, we record the throughputs. The throughput measurements are plotted on the graph, and the frontier is defined by the extremities of these points.

In the evaluation, both the magnitude and the shape of the throughput frontier are considered. The graph is bounded by a bounding box of maximum throughputs, and a proportional line shows a linear dependency between the transactional and analytical performance. A throughput frontier above the proportional line and closer to the bounding box indicates very good performance isolation.

Freshness

Freshness measures the lag between transactional and analytical data. In HATtrick, it is calculated using periodic freshness queries that utilize the monotonically increasing IDs of the *New Order* transaction, a pure insert operation. Freshness is defined as the time difference between the execution of an analytical query and the completion time of the first transaction that was committed, but not yet visible in the query results. Each freshness query retrieves the last committed transaction ID, while transaction completions are logged with timestamps. This makes it possible to identify the first missing transaction in the query output. These measurements are then aggregated using mean or percentile statistics. Since freshness can vary significantly with input load, it is measured individually at each point. Freshness queries are run periodically as part of the hybrid load.

Latency

Latency is measured as the time difference between sending a transactional or analytical request and receiving its response. Latency metrics are collected for all workload combinations alongside throughput and freshness data.

Transaction Failures

Because Styx guarantees serializable transactions, we also apply a strict serializability constraint to our baseline. As a result, some transactions fail due to concurrency conflicts. These failures are counted and retries are attempted up to three times before a transaction is definitively reported as failed.

4.2.3 Baseline

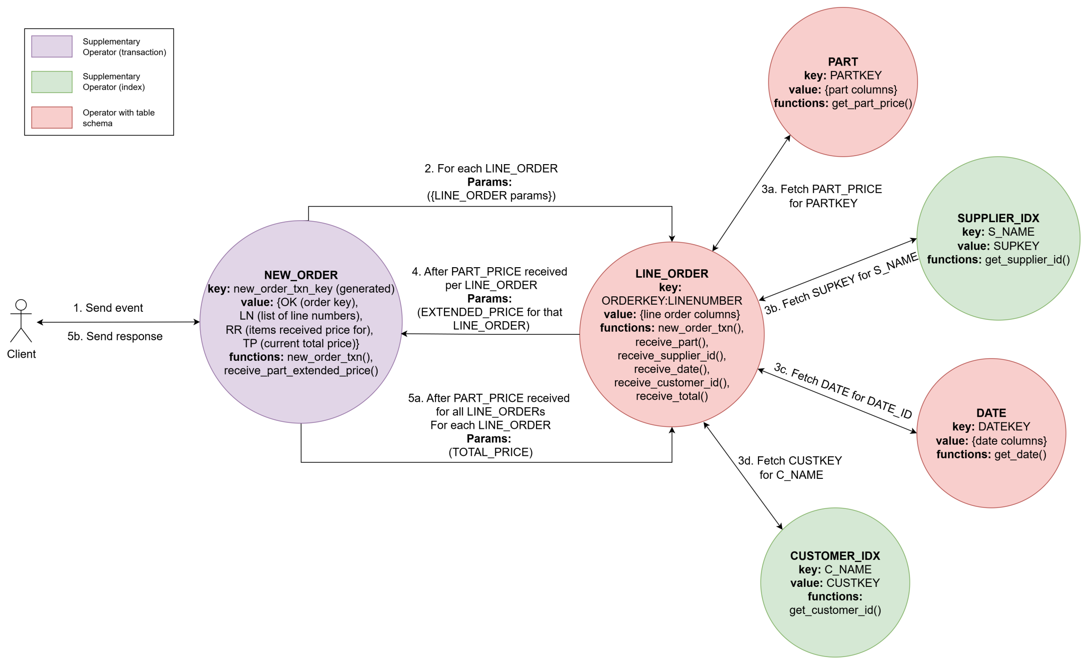
We selected PostgreSQL 14.5 with Streaming Replication (Postgres-SR) as the baseline HTAP system, following the best practices from [45]. PostgreSQL was configured with serializable transactions enabled, and asynchronous streaming replication to permit non-fresh analytical queries. System parameters, including cache size, WAL size, and streaming delay, were tuned for high-volume loads. This setup parallels Styx in its use of serializability and asynchronous replication, providing a fair comparison. The HATtrick benchmark code supports data generation according to the SSB schema and uses stored procedures to drive transactional loads on Postgres. Generated CSV data was loaded into a Docker-based Postgres deployment with streaming replication. Transactions were sent to the primary node, while analytical queries ran on the read replica. We additionally modified the benchmark to add features and collect additional metrics mentioned above.

4.2.4 HATtrick for H-Styx

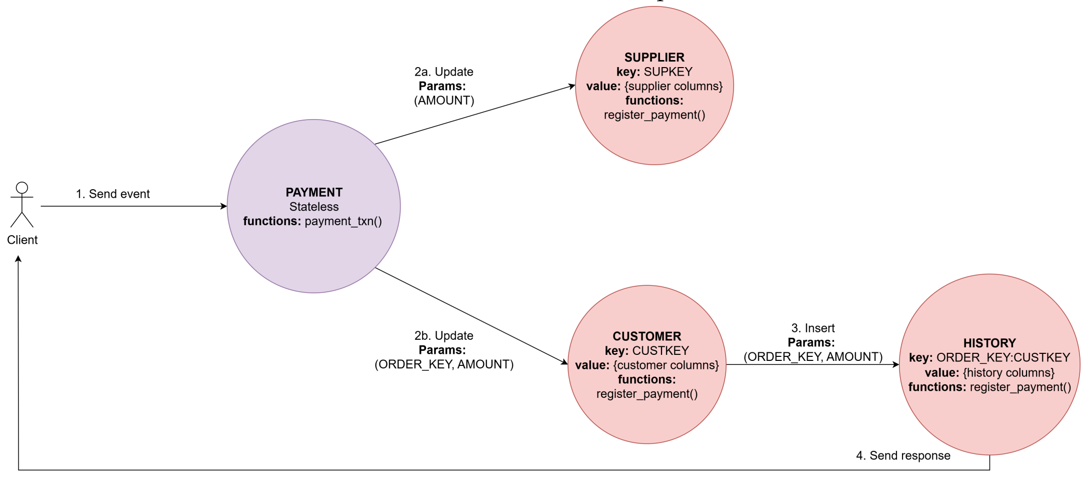
Transactions as Stateful Functions

Implementing HATtrick within Styx required converting the transaction logic into stateful functions to serve the transactional workload. Each of the tables was directly mapped to an operator. Further, the index columns used for each of the transactions were converted into index operators. For example, for *New Order* transaction, the customer name is provided within the input, and we need to retrieve the corresponding customer ID to insert into the `LINE_ORDER` table. This is done by creating an index operator where the key is the customer name and the value is the customer ID. Finally, transaction entrypoint functions are required, and for this purpose two operators were created for each of the transactions. The call graphs for both transactions are illustrated in Figure 4.1.

For the analytical client, the SQL queries were used directly and sent to the Query Engine via `StyxClient`.


Supplementary Operator (transaction)
Supplementary Operator (index)
Operator with table schema
PART
key: PARTKEY
value: {part columns}
functions: get_part_price()
2. For each LINE_ORDER
Params:
({LINE_ORDER params})
3a. Fetch PART_PRICE for PARTKEY
SUPPLIER_IDX
key: S_NAME
value: SUPKEY
functions: get_supplier_id()
NEW_ORDER
key: new_order_txn_key (generated)
value: {OK (order key), LN (list of line numbers), RR (items received price for), TP (current total price)}
functions: new_order_txn(), receive_part_extended_price()
4. After PART_PRICE received per LINE_ORDER
Params:
(EXTENDED_PRICE for that LINE_ORDER)
LINE_ORDER
key:
ORDERKEY:LINENUMBER
value: {line order columns}
functions: new_order_txn(), receive_part(), receive_supplier_id(), receive_date(), receive_customer_id(), receive_total()
3b. Fetch SUPKEY for S_NAME
1. Send event
5b. Send response
Client
3c. Fetch DATE for DATE_ID
DATE
key: DATEKEY
value: {date columns}
functions: get_date()
5a. After PART_PRICE received for all LINE_ORDERs
For each LINE_ORDER
Params:
(TOTAL_PRICE)
3d. Fetch CUSTKEY for C_NAME
CUSTOMER_IDX
key: C_NAME
value: CUSTKEY
functions:
get_customer_id()

(A) New Order Call Graph



(B) Payment Call Graph

FIGURE 4.1: Transactional Call Graphs for HATtrick

Finding Saturation Points

For the baseline system, saturation throughput is determined by gradually increasing the number of client threads that continuously send requests and wait for response after completion. In contrast, Styx has early commits and asynchronous replies. Therefore, we use a single-threaded client with increasing input rates (measured in TPS and QPS) to identify the throughput bounding box. These input requests are recorded to a file. At the end of each hybrid run, a consumer task polls Kafka for response messages from Styx's output topics. By joining these responses with the logged input requests, we reconstruct the complete results for all issued requests. Metrics are then computed based on this matched data. Throughput frontier points are derived using the same procedure.

4.2.5 Hardware Limits

We monitored the initial HATtrick runs to document hardware utilization across the baseline and H-Styx deployments. Based on these observations, we allocated 5 CPUs for transactional workloads and 7 CPUs for analytical workloads. This corresponds to a 5–7 CPU split between the primary and read replica in the Postgres baseline. In the H-Styx deployment, we allocated 1 CPU for the coordinator and 1 CPU per transactional worker, leading to a total of 5, and 7 CPUs for the Query Engine.

4.2.6 Varying Parameters

The HATtrick benchmark was executed using scale factors 1 and 10 of the SSB dataset. Scale factor 1 corresponds to approximately 600 MB of data, while scale factor 10 is roughly 6 GB. These variations were used to evaluate H-Styx's performance for larger datasets. Each hybrid configuration run was 60 seconds long, with 10 seconds of warm-up time.

Chapter 5

Results

In this chapter, we document the results of the experiments described in the previous chapter.

5.1 Impact of Query Engine on Styx Transactions

In the first experiment, we compared the performance of Styx and H-Styx over 10 independent runs. The results are summarized in Tables 5.1, 5.2, and 5.3. At a high level, both systems demonstrated nearly identical performance under normal workloads. All measured metrics confirm non-inferiority, and thereby the primary hypothesis that the introduction of the Query Engine in H-Styx does not negatively affect transactional performance under regular load. The architecture of H-Styx appears to successfully decouple the analytical and transactional execution paths.

5.1.1 Transactional Metrics

Across 10 independent runs for the configuration of 10,000-key dataset and 30-second snapshot interval, both Styx and H-Styx showed almost identical transactional behavior. Mean transactional throughput remained stable around 982 transactions per second, with marginal deviations well within the defined margin. Latency metrics showed close alignment across both systems, with no observable degradation in tail latency in H-Styx despite the additional load from concurrent query processing under moderate input rates.

Non-inferiority testing on mean latency, 95th percentile latency, and throughput showed that all results were within the non-inferiority margins defined in Table 4.1, with p-values supporting the rejection of the null hypothesis in each case.

The data remained consistent at the end of each run of transaction processing indicating no transactional anomalies across all runs of both Styx and H-Styx. Similarly, no missed messages were recorded in any sample, indicating that the event-driven messaging of Styx remains reliable even with the additional processing layers introduced in H-Styx.

5.1.2 Impact of Dataset Size

When scaling the dataset from 10,000 to 100,000 keys, both systems showed similar trends. Transaction latency and throughput remained stable, highlighting the isolation of analytical processing in H-Styx across different data sizes.

5.1.3 Snapshot Interval Variation

When varying the snapshot interval from 30 seconds to 10 seconds on the 10,000-key dataset, H-Styx maintained its transactional performance, with only a marginal increase in mean latency, which may be attributed to more frequent snapshot delta loads. Throughput remained

TABLE 5.1: YCSB-T for Styx and H-Styx for 10k rows

Metric	Styx	H-Styx	Δ	p-value	Non-Inferiority
Mean Latency	4.629ms	4.631ms	0.1ms	3e-3	true
95p Latency	7.0ms	7.0ms	0.1ms	0	true
99p Latency	9.0ms	9.0ms	1ms	0	true
Mean Throughput	982.588tps	982.282tps	1tps	1e-4	true

TABLE 5.2: YCSB-T for Styx and H-Styx for 100k rows

Metric	Styx	H-Styx	Δ	p-value	Non-Inferiority
Mean Latency	4.658ms	4.662ms	0.1ms	7.8e-4	true
95p Latency	7.0ms	7.0ms	0.1ms	0	true
99p Latency	9.1ms	9.2ms	1ms	0	true
Mean Throughput	981.990tps	982.017tps	1tps	1e-14	true

unaffected, and consistency was preserved throughout. This demonstrates that H-Styx is resilient to higher snapshot frequency and can sustain consistent transaction processing even with more frequent state captures, which can lead to fresher analytical queries.

As shown in Table 5.3, mean latency in H-Styx rose slightly by 0.014ms, a negligible increase. Throughput remained stable, and tail latencies were statistically indistinguishable from those observed with the 30-second interval. These findings suggest that the snapshot mechanism in H-Styx is lightweight and efficiently integrated into the runtime, enabling fresher analytical data without compromising transaction responsiveness.

TABLE 5.3: YCSB-T for Styx and H-Styx for 10s snapshot interval

Metric	Styx	H-Styx	Δ	p-value	Non-Inferiority
Mean Latency	4.624ms	4.638ms	0.1ms	4e-5	true
95p Latency	7.0ms	7.0ms	0.1ms	0	true
99p Latency	9.1ms	9.0ms	1ms	0	true
Mean Throughput	982.380tps	982.478tps	1tps	0	true

5.2 Comparison with an HTAP System

In the second experiment, we compared the performance of H-Styx to an HTAP system: Postgres-SR. A comprehensive analysis of metrics was done for both systems to gain insight about H-Styx's hybrid performance.

5.2.1 Performance Isolation

We compared the throughput frontiers of the two systems for scaling factor 1 (Figure 5.1) and scaling factor 10 (Figure 5.2). For SF1, H-Styx exhibited a small but steady decline in analytical performance from early on, despite achieving much higher analytical throughput than Postgres-SR. Postgres-SR maintained good performance isolation until it reached saturation, after which its analytical performance degraded sharply. This suggests that Postgres-SR isolates analytical and transactional workloads better under moderate load but suffers significant degradation near saturation, while H-Styx sustains consistent analytical performance even at high transactional throughput.

A similar trend appeared for SF10. Postgres-SR again provided stronger isolation until near saturation, where performance dropped sharply, as in SF1. H-Styx showed a slightly greater decline in analytical throughput compared to SF1 but followed the same general pattern.

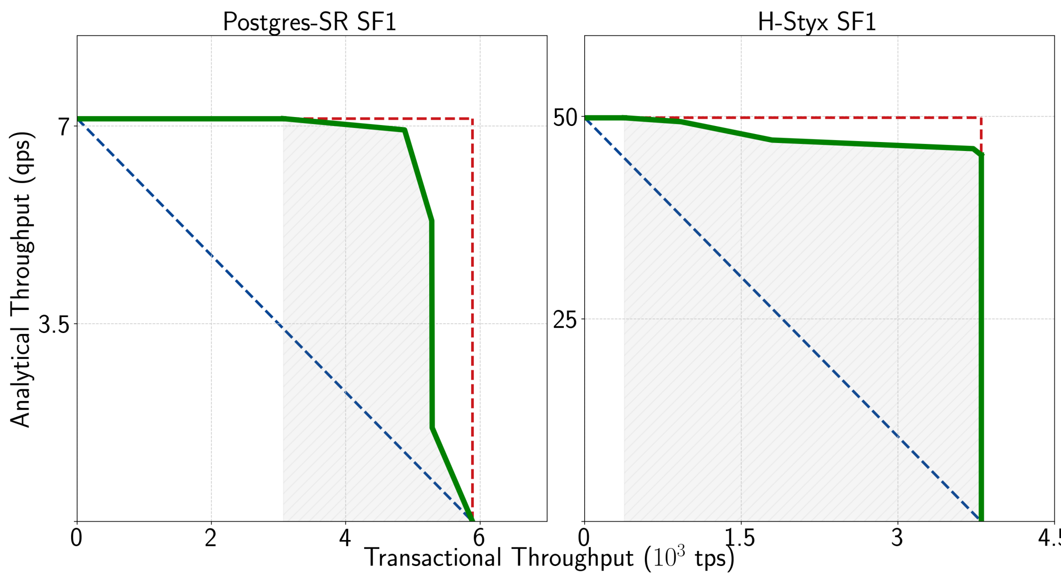


FIGURE 5.1: Throughput Frontier for SF1

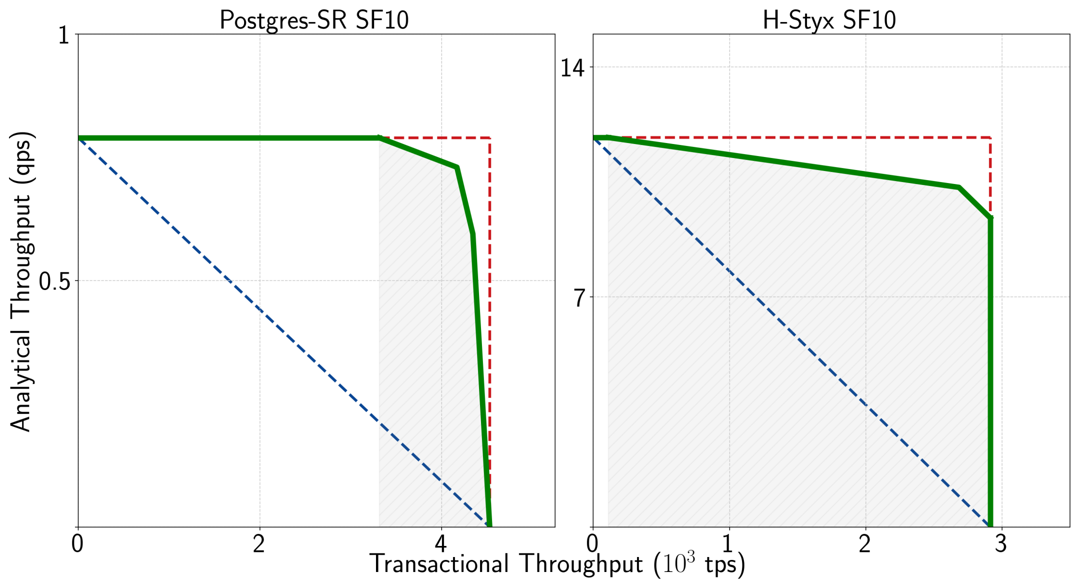


FIGURE 5.2: Throughput Frontier for SF10

5.2.2 HTAP Performance

We superimposed the throughput frontiers for both scaling factors in Figure 5.3. For SF1, Postgres-SR reached a maximum transactional throughput of approximately 6000 tps, compared to 3800 tps for H-Styx. However, H-Styx achieved an analytical throughput of 50 qps, significantly higher than Postgres-SR's 7 qps. For SF10, Postgres-SR reached around 4500 tps in transactional throughput, while H-Styx achieved 2900 tps. In terms of analytical throughput, H-Styx maintained about 12 qps, compared to Postgres-SR's approximately 1 qps.

Across both scales, H-Styx consistently achieved significantly higher analytical throughput, while Postgres-SR provides superior transactional throughput. These results highlight the systems' peak performance profiles: H-Styx is better suited for analytical queries, whereas Postgres-SR favors transactional workloads. Nevertheless, under moderate loads, H-Styx completely envelopes the Postgres-SR frontier, indicating superior HTAP performance according to the HATtrick evaluation criteria.

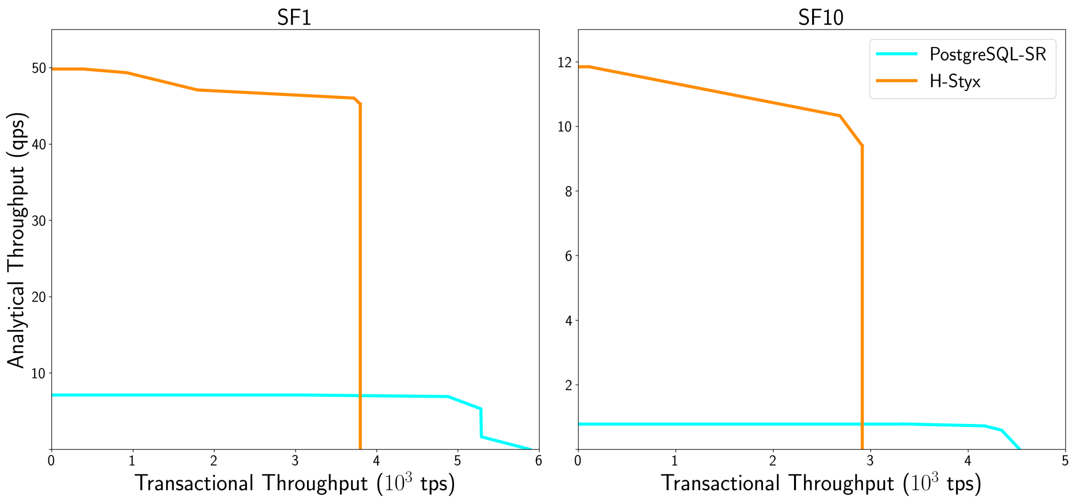


FIGURE 5.3: Unified Throughput Frontiers for SF1 and SF10

5.2.3 Freshness

We compared the freshness of both systems across varying transactional throughputs to assess how freshness is affected as transaction volume, and consequently delta load on the read replica or Query Engine, increases. The results shown in Figures 5.4 and 5.5 demonstrate that H-Styx consistently delivered better freshness than Postgres-SR for the mean, 95th percentile, and maximum freshness metrics.

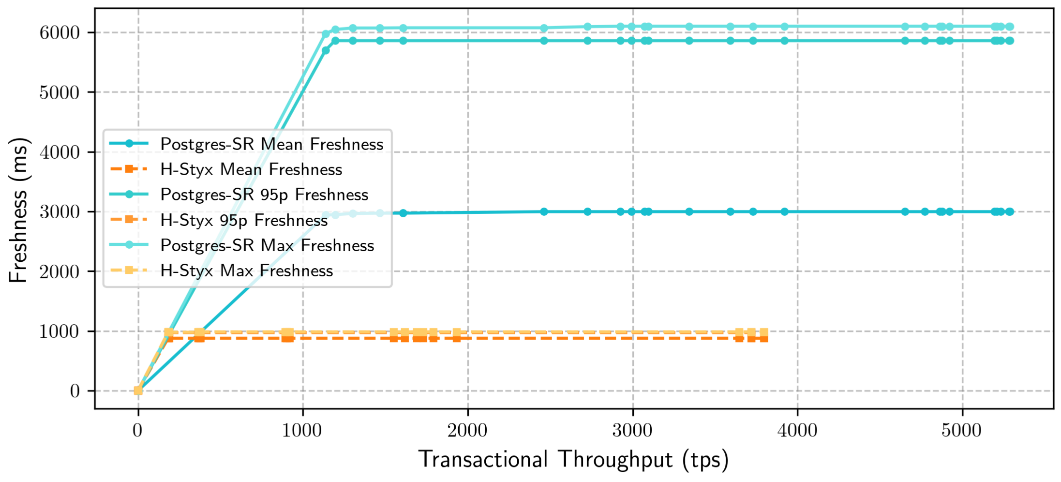


FIGURE 5.4: Transactional Throughput vs Freshness for SF1

Additionally, we evaluated freshness in H-Styx under varying analytical throughput to examine its behavior relative to query volume in Figure 5.6. This analysis was limited to H-Styx because Postgres-SR's analytical throughput was significantly lower. The results indicate that freshness remains stable across increasing query loads for both data sizes, demonstrating good scalability.

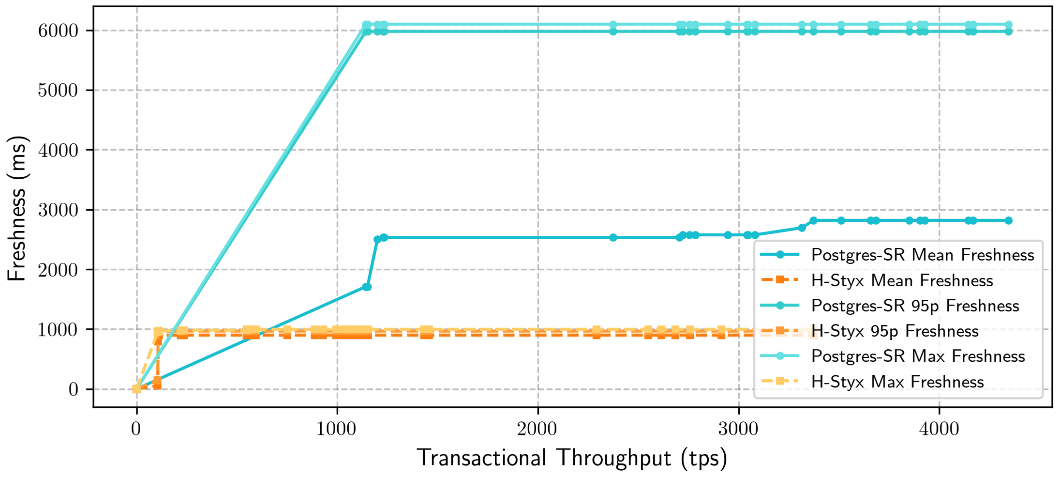


FIGURE 5.5: Transactional Throughput vs Freshness for SF10

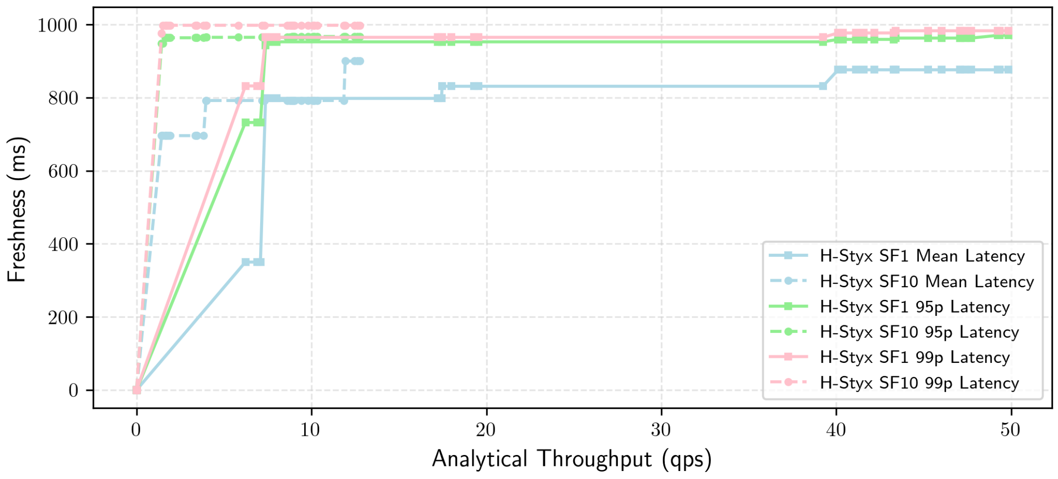


FIGURE 5.6: Analytical Throughput vs Freshness for H-Styx

5.2.4 Transactional Latency

The transactional latency comparison revealed a significant difference between Postgres-SR and H-Styx. Postgres-SR maintained consistently low latency of a few milliseconds even as load increased. For SF1, H-Styx sustained low latency up to around 1000 transactions per second (tps), after which latency gradually rose until 2000 tps and then sharply increased near 3500 tps. There was a notable gap between mean and worst-case latencies, with many transactions experiencing latencies close to 20 seconds once throughput exceeds 3000 tps.

This pattern was similar for SF10, though with some differences. While H-Styx showed higher latency at lower throughputs for SF10 compared to SF1, it performed significantly better near saturation. At nearly 3500 tps, the mean latency for SF10 remained under 12 seconds, compared to almost 20 seconds for SF1. Worst-case latency at saturation was also much lower for SF10 – around 18 seconds at 3500 tps versus over 40 seconds for SF1 at 3800 tps. These results indicate that H-Styx delivers better transactional performance on larger datasets with lower contention.

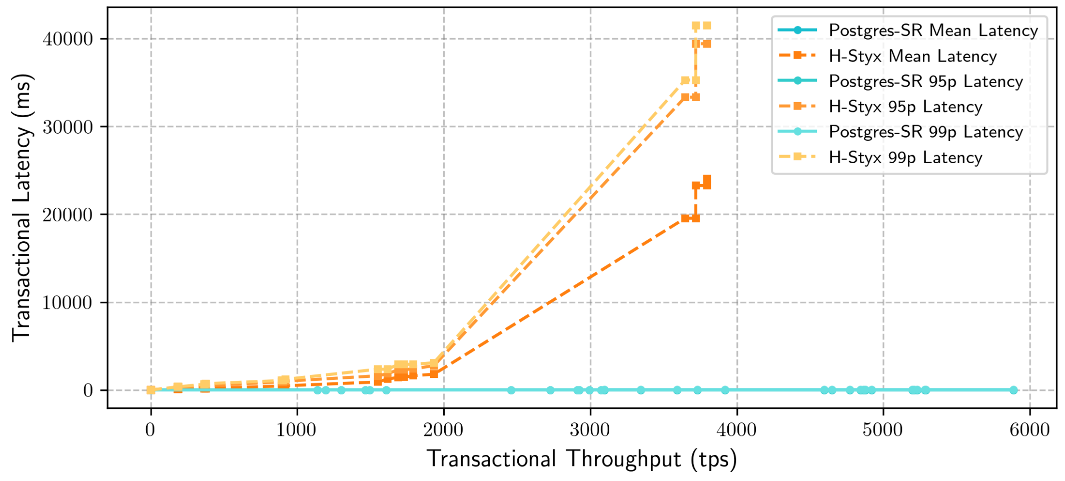


FIGURE 5.7: Transactional Throughput vs. Latency for SF1

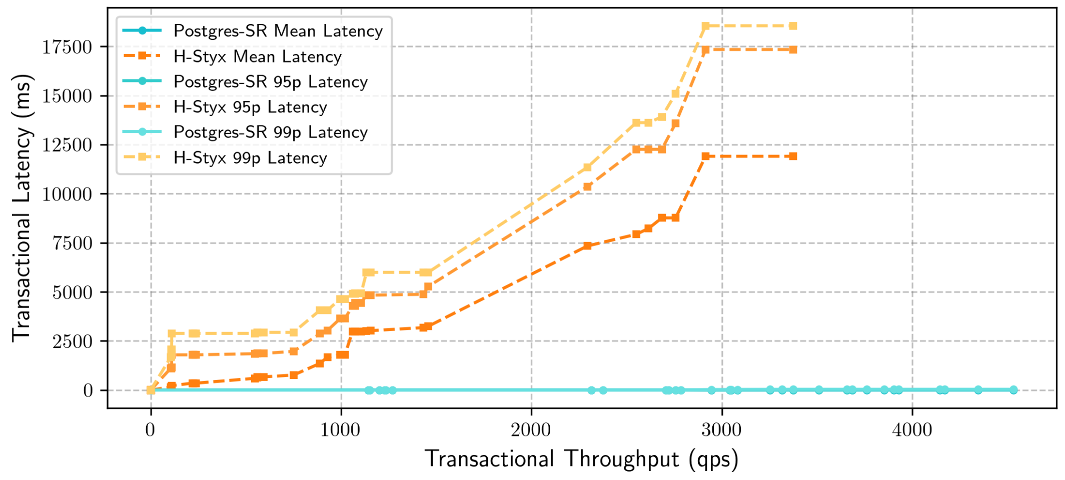


FIGURE 5.8: Transactional Throughput vs. Latency for SF10

5.2.5 Analytical Latency

The scaling factor significantly impacted analytical latency, as illustrated in Figures 5.9 and 5.10. Postgres-SR demonstrated lower worst-case analytical latency for SF1 queries but exhibited much higher latency under SF10, with worst-case latencies exceeding 60 seconds. This aligned with its low analytical throughput of barely 1 qps at SF10.

In comparison, H-Styx showed similar latency for comparable throughput under SF1, though with notably worse 95th and 99th percentile latencies. However, H-Styx achieved comparatively lower latency under SF10. For fairness, the H-Styx readings were filtered to analytical throughput measures equivalent to those of Postgres-SR. The complete latency plot for H-Styx is shown in Figure 5.11.

For SF1, H-Styx maintained stable latency across mean, 95th, and 99th percentiles up to about 40 qps, after which latency rose sharply to around 60 seconds on average, with worst-case values reaching 90 seconds. Under SF10, a similar pattern emerged: latency remained around 20 seconds up to 8 qps, then increased to 90 seconds on average, with worst-case latency reaching as high as 120 seconds.

These results indicate that although H-Styx supports significantly higher analytical throughput, its asynchronous design results in higher latencies that must be considered when evaluating it for HTAP workloads.

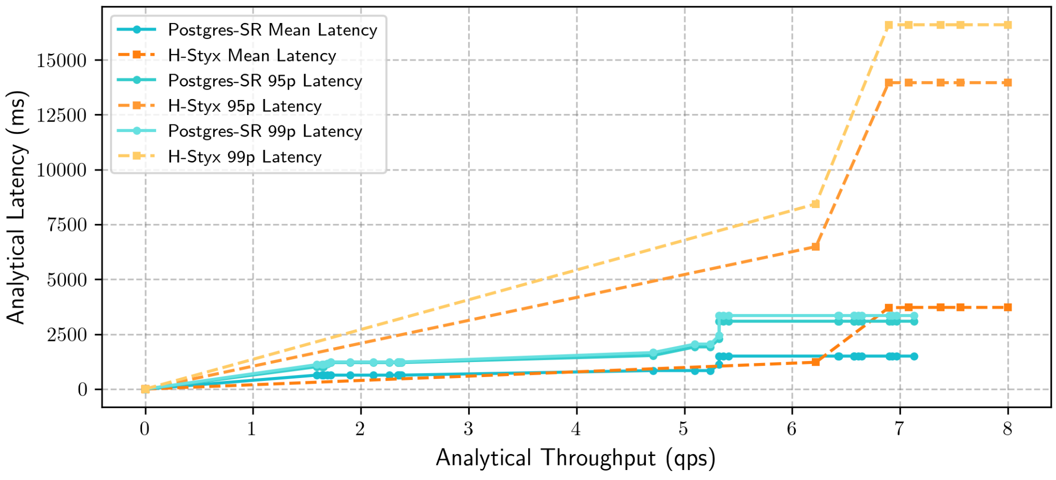


FIGURE 5.9: Analytical Throughput vs. Latency for SF1

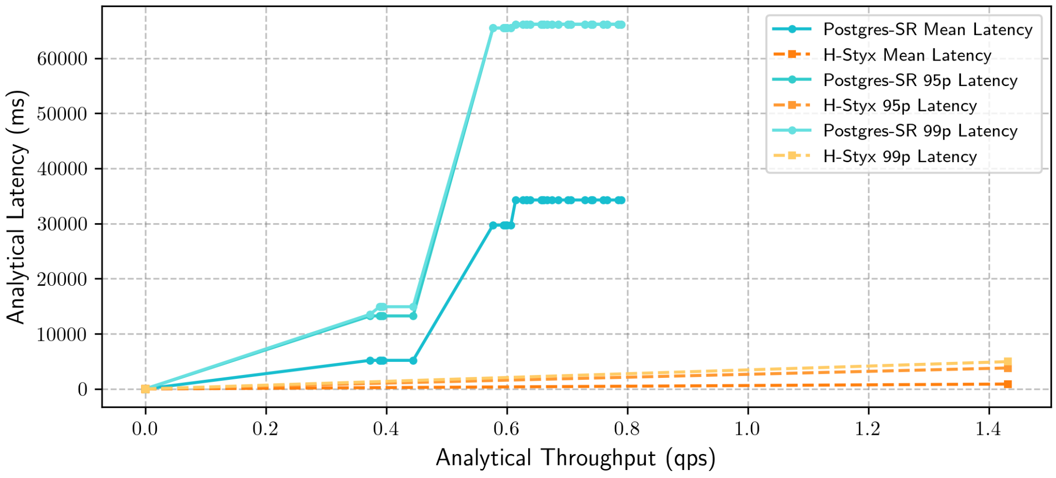


FIGURE 5.10: Analytical Throughput vs. Latency for SF10

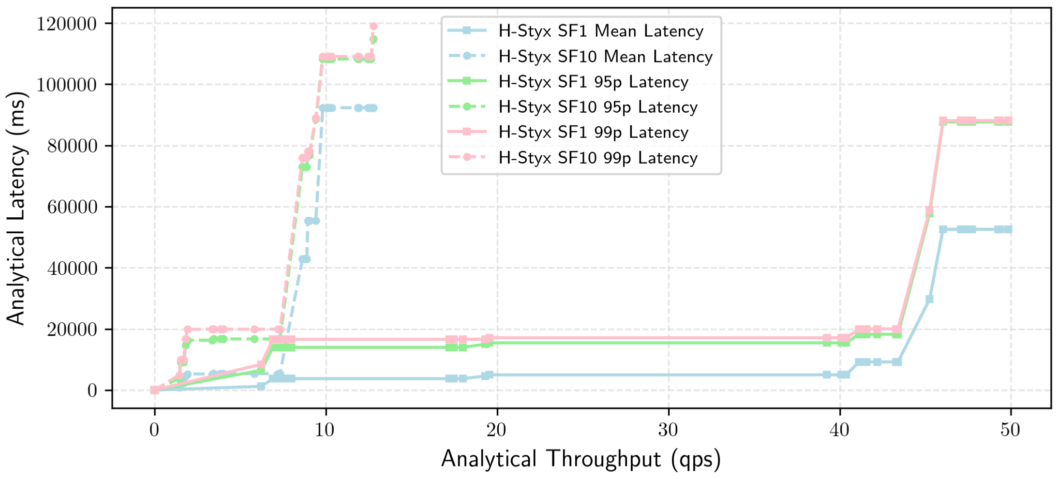


FIGURE 5.11: Analytical Throughput vs. Latency for H-Styx

5.2.6 Transaction Failures in Postgres-SR

Finally, we measured transaction failures in Postgres-SR caused by serialization errors. No failures were observed at lower throughputs; however, as throughput increased, the number of failed transactions rose due to higher contention and locking conflicts. Around 3000 tps, failures began to appear for both scaling factors. SF1 exhibited a maximum failure rate of 2.4%, while SF10 reached over 3%. At peak transactional throughput, these failure rates corresponded to approximately 144 failed transactions per second for SF1 and 120 failed transactions per second for SF10.

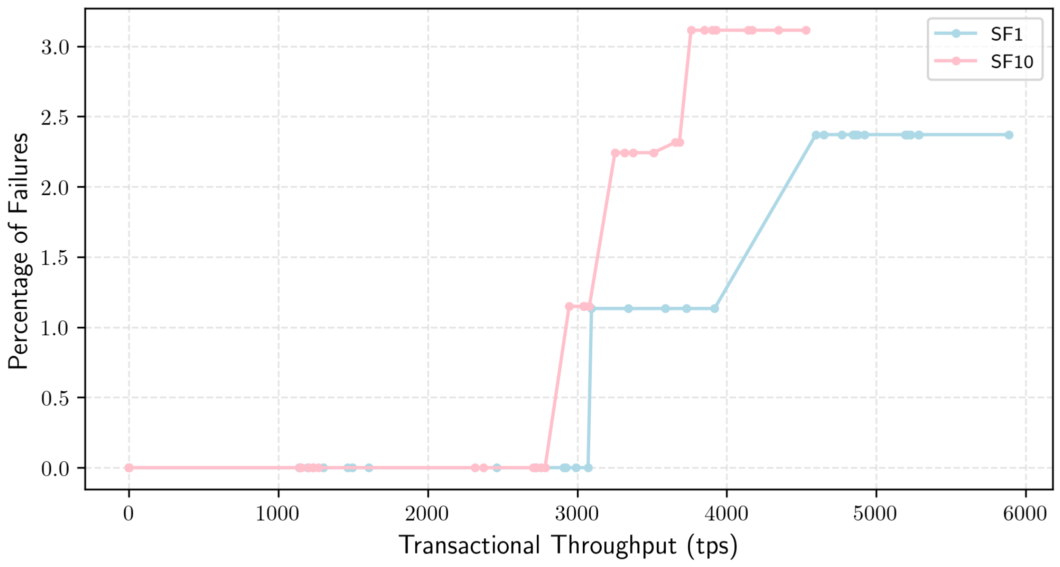


FIGURE 5.12: Transactional Throughput vs Failures for Postgres-SR

Chapter 6

Discussion

In this chapter, we reflect upon the results of the experiments conducted and their implications for our developed solution.

6.1 Analysis of Results

This work set out to investigate whether periodic blob-store snapshots could enable querying of the global distributed state of a streaming system without requiring traditional ETL and while maintaining minimal interference with transactional processing (RQ1). The design of H-Styx achieves this by integrating a lightweight Query Engine into the Styx architecture, operating on isolated hardware and using periodic snapshots to expose analytical views. As discussed in Chapter 3, this approach preserves Styx's transactional guarantees, since the Query Engine is decoupled from the transaction processing path. Experiment 1 validated this design by demonstrating that switching from Styx to H-Styx required only a configuration flag, with no significant changes to deployment or architecture, confirming that periodic blob-store snapshots can be leveraged effectively with minimal impact on transactional operations.

In addressing RQ2, concerning the impact of introducing this snapshot-based Query Engine on transactional throughput, latency, and consistency, the evaluation showed positive results. Experiment 1 employed non-inferiority testing across a range of parameters, revealing negligible changes in transactional throughput and latency after integrating the Query Engine. H-Styx maintained stable performance even as dataset size increased, suggesting that the architectural enhancements, particularly snapshotting and query handling, scale linearly and do not introduce bottlenecks in state access or processing.

Varying the snapshot interval produced similar results. H-Styx's performance remained comparable despite more frequent delta loads. This variation increased the rate of snapshot ingestion, potentially introducing higher I/O and computational overhead, yet H-Styx handled the increased workload without degradation. These findings strengthen the case for H-Styx as a scalable extension of Styx capable of supporting HTAP workloads without compromising OLTP performance as data volume grows. Experiment 2 further confirmed this by demonstrating strong performance isolation under moderate hybrid workloads. Even at saturation, no observable drop in transactional throughput occurred. This validates that the hybrid design of H-Styx preserves transactional performance nearly identical to that of Styx, quantitatively confirming its ability to serve hybrid workloads without compromising consistency or throughput.

In relation to RQ3, which compares H-Styx with a baseline HTAP system (Postgres with Streaming Replication) in terms of workload isolation, data freshness, and performance, the evaluation highlighted several key advantages of H-Styx. First, its decoupled architecture enables analytical queries to proceed largely independently of the transactional workload, supporting much higher analytical throughput and lower latency compared to Postgres-SR, both under analytical-only and hybrid loads. H-Styx does exhibit early decay in analytical performance, especially at SF10, but this can be attributed to the asynchronous snapshot mechanism and DuckDB's limited support for multiprocessing and thread-safety, which forces the Query Engine to context-switch between delta loading and query processing. As delta loads grow, these context switches can delay analytical queries, causing performance degradation. Nevertheless, because of DuckDB's columnar architecture, query execution remains substantially faster than the row-based Postgres-SR.

H-Styx shows stronger transactional correctness compared to Postgres-SR, since the latter shows transaction aborts and serializability anomalies under high load which amount to a large number of failures, while H-Styx maintains serializable transaction execution without failures even under hybrid workloads. However, H-Styx does have higher transactional latency than Postgres-SR due to its asynchronous, distributed transaction model involving multiple network calls, compared to Postgres's single-node design with minimal coordination overhead. Styx scales well at larger data volumes, with SF10 experiments demonstrating better transactional performance than SF1, highlighting the benefits of its distributed architecture.

In terms of data freshness, H-Styx shows better freshness compared to Postgres-SR, but this advantage may be a consequence of higher analytical latency: because the Query Engine is asynchronous, queries are processed by DuckDB slightly later, allowing fresher snapshot data to be incorporated before the query completes. The analytical latency plot for SF1 show that this improved freshness comes at the cost of longer waiting times. Overall, for moderate hybrid loads, latency and freshness are both acceptable and indicative of H-Styx being a successful hybrid system. Freshness remains relatively stable with increasing transactional throughput until saturation, and even drops to zero in some cases, reflecting highly up-to-date snapshots under those conditions.

In summary, these results confirm that H-Styx delivers strong hybrid workload support with minimal transactional interference, high analytical throughput, excellent workload isolation, and scalable performance across data sizes, while also being fault tolerant. These qualities position H-Styx as a promising architecture for modern streaming systems requiring global state queries without the overhead of traditional ETL, while preserving robust transactional and analytical guarantees.

6.2 Limitations

There are certain limitations in this study. Primarily, DuckDB operates in-process with a single connection for all operations. Although it supports multi-threaded execution internally via parallel cursors, it is not thread-safe with read-write. This constraint added complexity to the development of the Query Engine and makes distributed querying challenging. While DuckDB scales well in terms of storage and analytical query performance, its limited parallelism requires manual coordination at the application level to handle concurrent query workloads.

Additionally, the scope of this project and its evaluation focused on the analytical component, while the transactional subsystem of Styx was left unchanged. Transactional performance could potentially be improved further by fine-tuning Styx parameters such as the epoch size, number of workers, and by optimizing the design of stateful functions.

Another limitation lies in benchmarking: the evaluation could not cover a broader range of realistic HTAP use cases due to the lack of comprehensive HTAP benchmarks. While SSB is a well-established analytical benchmark, its transactional extensions (as used via HATtrick) are limited to only two basic transactions, and lack a unified metric. As a result, the analysis of freshness and hybrid throughput required more manual inspection and could be subject to some variability. Given the added complexity and variability introduced by serverless environments compared to traditional HTAP systems, more sophisticated and representative benchmarks are needed to fully capture real-world hybrid workloads.

Chapter 7

Conclusion

This thesis addressed the critical challenge of enabling efficient, real-time analytical queries over distributed state in Stateful Functions-as-a-Service systems. By designing and implementing H-Styx, an architecture that uses traditional HTAP architectures using periodic snapshots from blob store and an integrated delta store, this work demonstrated how the snapshotting mechanism of Styx can be extended to support near-real-time OLAP-style queries while preserving transactional guarantees.

The evaluation confirmed that H-Styx achieves strong hybrid workload performance, maintaining transactional throughput and latency comparable to Styx while delivering high analytical throughput and scalable query performance. Its decoupled, fault-tolerant design ensures robust workload isolation, and its periodic snapshot architecture allows fresh analytical views of the distributed state without the overhead of traditional ETL. Comparisons with a baseline HTAP system showed that H-Styx provides superior analytical performance while retaining the scalability and elasticity of a modern serverless architecture.

Overall, this work demonstrates the practicality and effectiveness of unifying transactional and analytical capabilities in an SFaaS environment, bridging an important gap in the current state-of-the-art and laying the foundation for more comprehensive, hybrid stateful data processing systems.

7.1 Future Scope

This work leaves substantial scope for further extension. Incorporating more advanced HTAP features, such as automatic schema inference, could be valuable. Although schema inference was intentionally excluded here due to the complexity of arbitrary keys and heterogeneous state structures, a sophisticated inference mechanism for simpler datasets could significantly enhance usability.

Currently, communication between the Query Engine and the Styx Coordinator is one-way; extending this to support bidirectional messaging would enable the Query Engine to signal events such as completion of an initial load or delta load. This could make it possible to re-enable Styx's native snapshot merging functionality, which is currently disabled to avoid inconsistent delta application.

Additionally, the evaluation in this thesis primarily focused on synthetic hybrid workloads like YCSB-T and SSB. While this provided useful insights into the architectural behavior of H-Styx, a more comprehensive validation would require benchmarking a broader set of realistic, domain-specific use cases, with scenarios involving multi-stage workflows, skewed access patterns, and a mixture of short- and long-running queries, posing unique challenges for both transactional and analytical components.

The hybrid performance of H-Styx also leaves room for improvement through finer-grained tuning of Styx's transactional runtime parameters -— such as epoch durations, snapshot intervals, worker-core bindings, and load-balancing strategies. This fine-tuning is non-trivial to optimize, especially in distributed and serverless environments, where the underlying execution is highly dynamic. Unlike monolithic systems such as our Postgres baseline, which operate on tightly integrated single-node architectures, modern cloud-native systems rely on distributed and disaggregated infrastructure. As a result, performance trends can differ significantly across configurations. This observation highlights the need for a dedicated benchmarking framework tailored to SFaaS platforms that is capable of evaluating transactional consistency, snapshot freshness, analytical throughput, and workload interference across varying deployment environments. Designing such a benchmark suite could itself form the basis for a valuable follow-up research project.

It is also worth noting that while this architecture is loosely coupled, which is a desirable trait for performance isolation, it does not support true live-state querying, but rather near-real-time queries via periodic snapshots. For use cases demanding strict real-time state observability, a more deeply integrated solution can be explored.

Finally, while the Query Engine was implemented specifically within the Styx infrastructure, its architectural principles are broadly applicable to other distributed systems employing periodic snapshots. Generalizing and modularizing the implementation with proper benchmarking could encourage adoption across other stateful streaming or serverless computing frameworks.

Appendix A

Analytical Queries for YCSB-T

The following are the analytical queries that are run for the analytical part of Experiment 1 on YCSB-T data.

```
SELECT COUNT(value) AS total_rows FROM ycsb;

SELECT SUM(value) AS total_sum FROM ycsb;

SELECT (value / 1000) AS bucket, COUNT(*) AS count
        FROM ycsb
        GROUP BY bucket
        ORDER BY bucket;
SELECT
      PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY value) AS p50,
      PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY value) AS p90,
      PERCENTILE_CONT(0.99) WITHIN GROUP (ORDER BY value) AS p99
    FROM ycsb;

SELECT STDDEV_POP(value) AS std_dev, VAR_POP(value) AS variance
   FROM ycsb;
```

Appendix B

Analytical Queries for HATtrick

The following are the analytical queries that are run for the analytical run of HATtrick.

```
SELECT SUM(lo.EXTENDEDPRICE*lo.DISCOUNT) AS REVENUE
FROM LINE_ORDER lo, DATE d
WHERE lo.ORDERDATE = d.DATEKEY
AND d.YEAR = 1993
AND lo.DISCOUNT BETWEEN 1 AND 3
AND lo.QUANTITY<25;

SELECT SUM(lo.EXTENDEDPRICE*lo.DISCOUNT) AS REVENUE
FROM LINE_ORDER lo, DATE d
WHERE lo.ORDERDATE = d.DATEKEY
AND d.YEARMONTHNUM = 199401
AND lo.DISCOUNT BETWEEN 4 AND 6
AND lo.QUANTITY BETWEEN 26 AND 35;

SELECT SUM(lo.EXTENDEDPRICE*lo.DISCOUNT) AS REVENUE
FROM LINE_ORDER lo, DATE d
WHERE lo.ORDERDATE = d.DATEKEY
AND d.WEEKNUMINYEAR = 6
AND d.YEAR = 1994
AND lo.DISCOUNT BETWEEN 5 AND 7
AND lo.QUANTITY BETWEEN 26 AND 35;

SELECT SUM(lo.REVENUE), d.YEAR, p.BRAND1
FROM LINE_ORDER lo, DATE d, PART p, SUPPLIER s
WHERE lo.ORDERDATE = d.DATEKEY
AND lo.PARTKEY = p.PARTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND p.CATEGORY = 'MFGR#12'
AND s.REGION = 'AMERICA'
GROUP BY d.YEAR, p.BRAND1
ORDER BY d.YEAR, p.BRAND1;

SELECT SUM(lo.REVENUE), d.YEAR, p.BRAND1
FROM LINE_ORDER lo, DATE d, PART p, SUPPLIER s
WHERE lo.ORDERDATE = d.DATEKEY
AND lo.PARTKEY = p.PARTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND p.BRAND1 BETWEEN 'MFGR#2221' AND 'MFGR#2228'
AND s.REGION = 'ASIA'
GROUP BY d.YEAR, p.BRAND1
ORDER BY d.YEAR, p.BRAND1;
```

```
SELECT SUM(lo.REVENUE), d.YEAR, p.BRAND1
FROM LINE_ORDER lo, DATE d, PART p, SUPPLIER s
WHERE lo.ORDERDATE = d.DATEKEY
AND lo.PARTKEY = p.PARTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND p.BRAND1 = 'MFGR#2221'
AND s.REGION = 'EUROPE'
GROUP BY d.YEAR, p.BRAND1
ORDER BY d.YEAR, p.BRAND1;

SELECT c.NATION, s.NATION, d.YEAR, SUM(lo.REVENUE) AS REVENUE
FROM CUSTOMER c, LINE_ORDER lo, SUPPLIER s, DATE d
WHERE lo.CUSTKEY = c.CUSTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND lo.ORDERDATE = d.DATEKEY
AND c.REGION = 'ASIA' AND s.REGION = 'ASIA'
AND d.YEAR >= 1992 AND d.YEAR <= 1997
GROUP BY c.NATION, s.NATION, d.YEAR
ORDER BY d.YEAR ASC, REVENUE DESC;

SELECT c.CITY, s.CITY, d.YEAR, SUM(lo.REVENUE) AS REVENUE
FROM CUSTOMER c, LINE_ORDER lo, SUPPLIER s, DATE d
WHERE lo.CUSTKEY = c.CUSTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND lo.ORDERDATE = d.DATEKEY
AND c.NATION = 'UNITED STATES'
AND s.NATION = 'UNITED STATES'
AND d.YEAR >= 1992 AND d.YEAR <= 1997
GROUP BY c.CITY, s.CITY, d.YEAR
ORDER BY d.YEAR ASC, REVENUE DESC;

SELECT c.CITY, s.CITY, d.YEAR, SUM(lo.REVENUE) AS REVENUE
FROM CUSTOMER c, LINE_ORDER lo, SUPPLIER s, DATE d
WHERE lo.CUSTKEY = c.CUSTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND lo.ORDERDATE = d.DATEKEY
AND (c.CITY = 'UNITED KI1' OR c.CITY = 'UNITED KI5')
AND (s.CITY = 'UNITED KI1' OR s.CITY = 'UNITED KI5')
AND d.YEAR >= 1992 AND d.YEAR <= 1997
GROUP BY c.CITY, s.CITY, d.YEAR
ORDER BY d.YEAR ASC, REVENUE DESC;

SELECT c.CITY, s.CITY, d.YEAR, SUM(lo.REVENUE) AS REVENUE
FROM CUSTOMER c, LINE_ORDER lo, SUPPLIER s, DATE d
WHERE lo.CUSTKEY = c.CUSTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND lo.ORDERDATE = d.DATEKEY
AND (c.CITY = 'UNITED KI1' OR c.CITY = 'UNITED KI5')
AND (s.CITY = 'UNITED KI1' OR s.CITY = 'UNITED KI5')
AND d.YEARMONTH = 'Dec1994'
GROUP BY c.CITY, s.CITY, d.YEAR
ORDER BY d.YEAR ASC, REVENUE DESC;
```

```sql
SELECT d.YEAR, c.NATION, SUM(lo.REVENUE-lo.SUPPLYCOST) AS PROFIT
FROM DATE d, CUSTOMER c, SUPPLIER s, PART p, LINE_ORDER lo
WHERE lo.CUSTKEY = c.CUSTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND lo.PARTKEY = p.PARTKEY
AND lo.ORDERDATE = d.DATEKEY
AND c.REGION = 'AMERICA'
AND s.REGION = 'AMERICA'
AND (p.MFGR = 'MFGR#1' OR p.MFGR = 'MFGR#2')
GROUP BY d.YEAR, c.NATION
ORDER BY d.YEAR, c.NATION;

SELECT d.YEAR, c.NATION, p.CATEGORY, SUM(lo.REVENUE-lo.
    SUPPLYCOST) AS PROFIT
FROM DATE d, CUSTOMER c, SUPPLIER s, PART p, LINE_ORDER lo
WHERE lo.CUSTKEY = c.CUSTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND lo.PARTKEY = p.PARTKEY
AND lo.ORDERDATE = d.DATEKEY
AND c.REGION = 'AMERICA'
AND s.REGION = 'AMERICA'
AND (d.YEAR = 1997 OR d.YEAR = 1998)
AND (p.MFGR = 'MFGR#1' OR p.MFGR = 'MFGR#2')
GROUP BY d.YEAR, c.NATION, p.CATEGORY
ORDER BY d.YEAR, c.NATION, p.CATEGORY;

SELECT d.YEAR, s.CITY, p.BRAND1, SUM(lo.REVENUE-lo.SUPPLYCOST)
    AS PROFIT
FROM DATE d, CUSTOMER c, SUPPLIER s, PART p, LINE_ORDER lo
WHERE lo.CUSTKEY = c.CUSTKEY
AND lo.SUPPKEY = s.SUPPKEY
AND lo.PARTKEY = p.PARTKEY
AND lo.ORDERDATE = d.DATEKEY
AND c.REGION = 'AMERICA'
AND s.NATION = 'UNITED STATES'
AND (d.YEAR = 1997 OR d.YEAR = 1998)
AND p.CATEGORY = 'MFGR#14'
GROUP BY d.YEAR, s.CITY, p.BRAND1
ORDER BY d.YEAR, s.CITY, p.BRAND1;
```

Bibliography

[1] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. “Stateful functions as a service in action”. In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1890–1893.

[2] Yousef J Al-Houmaily and George Samaras. “Two-phase commit”. In: *Encyclopedia of Database Systems*. Springer, 2018, pp. 4259–4265.

[3] Moiz Arif, Kevin Assogba, and M Mustafa Rafique. “Canary: fault-tolerant faas for stateful time-sensitive applications”. In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2022, pp. 1–16.

[4] Michael Armbrust et al. “Delta lake: high-performance ACID table storage over cloud object stores”. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3411–3424.

[5] Irina Astrova et al. “Serverless, FaaS and why organizations need them”. In: *Intelligent Decision Technologies* 15.4 (2021), pp. 825–838.

[6] *AWS Lambda Documentation*. https://docs.aws.amazon.com/lambda/. [Accessed 24-06-2025].

[7] *AWS Step Functions Documentation*. https://docs.aws.amazon.com/step-functions/. [Accessed 24-06-2025].

[8] Ronald Barber et al. “Wildfire: HTAP for big data”. In: *Encyclopedia of Big Data Technologies*. Springer, 2018, pp. 1–7.

[9] Daniel Barcelona-Pons et al. “On the faas track: Building stateful distributed applications with serverless architectures”. In: *Proceedings of the 20th international middleware conference*. 2019, pp. 41–54.

[10] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading, 1987.

[11] Hebert Cabane and Kleinner Farias. “On the impact of event-driven architecture on performance: An exploratory study”. In: *Future Generation Computer Systems* 153 (2024), pp. 52–69.

[12] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

[13] Bruno Mario Cesana. “What p-value must be used as the Statistical Significance Threshold? P< 0.005, P< 0.01, P< 0.05 or no value at all”. In: *Biomedical Journal of Scientific & Technical Research* 6.3 (2018), pp. 5310–5318.

[14] Sanket Chintapalli et al. “Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 1789–1792. DOI: 10.1109/IPDPSW.2016.138.

[15] Fábio Coelho et al. “Htapbench: Hybrid transactional and analytical processing benchmark”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 293–304.

[16] Richard Cole et al. “The mixed workload CH-benCHmark”. In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. 2011, pp. 1–6.

[17] Yan Cui, Yu Chen, and Yuanchun Shi. "Scaling OLTP applications on commodity multi-core platforms". In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE. 2010, pp. 134–143.

[18] Akon Dey et al. "YCSB+ T: Benchmarking web-scale transactional databases". In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE. 2014, pp. 223–230.

[19] Haowen Dong et al. "Cloud-native databases: A survey". In: *IEEE Transactions on Knowledge and Data Engineering* (2024).

[20] *DuckDB Documentation - Indexing*. `https://duckdb.org/docs/stable/guides/performance/indexing.html`. [Accessed 24-06-2025].

[21] Nnamdi Ekwe-Ekwe and Lucas Amos. "The State of FaaS: An analysis of public Functions-as-a-Service providers". In: *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. IEEE. 2024, pp. 430–438.

[22] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database system implementation*. Vol. 672. Prentice Hall Upper Saddle River, 2000.

[23] Nishant Garg. *Apache kafka*. Packt Publishing, 2013.

[24] *Index - Polars user guide — docs.pola.rs*. `https://docs.pola.rs/`. [Accessed 17-06-2025].

[25] Zhipeng Jia and Emmett Witchel. "Boki: Stateful serverless computing with shared logs". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 691–707.

[26] Guoxin Kang, Simin Chen, and Hongxiao Li. "Benchmarking htap databases for performance isolation and real-time analytics". In: *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 3.2 (2023), p. 100122.

[27] Petr Kurapov and Areg Melik-Adamyan. "Analytical Queries: A Comprehensive Survey". In: *arXiv preprint arXiv:2311.15730* (2023).

[28] Scott T Leutenegger and Daniel Dias. "A modeling study of the TPC-C benchmark". In: *ACM Sigmod Record* 22.2 (1993), pp. 22–31.

[29] 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*. 2022, pp. 2483–2488.

[30] Ramakrishna Manchana. "Balancing Agility and Operational Overhead: Monolith Decomposition Strategies for Microservices and Microapps with Event-Driven Architectures". In: *North American Journal of Engineering Research* 2.2 (2021).

[31] Toni Mattis et al. "Columnar objects: Improving the performance of analytical applications". In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* 2015, pp. 197–210.

[32] Wes McKinney et al. "pandas: a foundational Python library for data analysis and statistics". In: *Python for high performance and scientific computing* 14.9 (2011), pp. 1–9.

[33] Brenda M Michelson. "Event-driven architecture overview". In: *Patricia Seybold Group* 2.12 (2006), pp. 10–1571.

[34] Elena Milkai et al. "How good is my HTAP system?" In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 1810–1824.

[35] Adhish Nanda, Swati Gupta, and Meenu Vijrania. "A comprehensive survey of OLAP: recent trends". In: *2019 3rd International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. IEEE. 2019, pp. 425–430.

[36] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. "The star schema benchmark (SSB)". In: *Pat* 200.0 (2007), p. 50.

[37] Richard Patsch and Karl Michael Göschka. "Make Applications FaaS-ready: Challenges and Guidelines". In: *Proceeding of the 2024 6th International Conference on Information Technology and Computer Communications*. 2024, pp. 57–64.

[38] Massimo Pezzini et al. "Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation". In: *Gartner (2014, January 28) Available at https://www. gartner. com/doc/2657815/hybrid-transactionanalyticalprocessing-foster-opportunities* (2014), pp. 4–20.

[39] Bjorn Pijnacker and Jesper van der Zwaag. "Opportunities and Challenges in the Adoption of Function-as-a-Service Serverless Computing". In: *20th SC@ RUG 2022-2023* (), p. 65.

[40] Kyriakos Psarakis et al. "Styx: Transactional Stateful Functions on Streaming Dataflows". In: *arXiv preprint arXiv:2312.06893* (2023).

[41] *Python &x2014; Apache Arrow v20.0.0 — arrow.apache.org*. `https://arrow.apache.org/docs/python/index.html`. [Accessed 17-06-2025].

[42] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, 1981–1984. ISBN: 9781450356435. DOI: `10.1145/3299869.3320212`. URL: `https://doi.org/10.1145/3299869.3320212`.

[43] Aunn Raza et al. "Adaptive HTAP through elastic resource scheduling". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2043–2054.

[44] Alexander I Saichev, Yannick Malevergne, and Didier Sornette. *Theory of Zipf's law and beyond*. Vol. 632. Springer Science & Business Media, 2009.

[45] Hans-Jürgen Schönig. *PostgreSQL Replication*. Packt Publishing Ltd, 2015.

[46] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. "Performance characterization of HTAP workloads". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE. 2021, pp. 1829–1834.

[47] Haoze Song et al. "A survey on hybrid transactional and analytical processing". In: *The VLDB Journal* 33.5 (June 2024), 1485–1515. ISSN: 0949-877X. DOI: `10.1007/s00778-024-00858-9`. URL: `http://dx.doi.org/10.1007/s00778-024-00858-9`.

[48] Vikram Sreekanti et al. "Cloudburst: Stateful functions-as-a-service". In: *arXiv preprint arXiv:2001.04592* (2020).

[49] Mark Szalay, Peter Matray, and Laszlo Toka. "Real-time faas: Towards a latency bounded serverless cloud". In: *IEEE Transactions on Cloud Computing* 11.2 (2022), pp. 1636–1650.

[50] *User Guide &x2014; pandas 2.3.0 documentation — pandas.pydata.org*. `https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html`. [Accessed 17-06-2025].

[51] Deepak Vohra. "Apache parquet". In: *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Springer, 2016, pp. 325–335.

[52] J Walker. "Non-inferiority statistics and equivalence studies". In: *BJA education* 19.8 (2019), pp. 267–271.

[53] *Why DuckDB — duckdb.org*. `https://duckdb.org/why_duckdb`. [Accessed 17-06-2025].

[54] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. "Transactional causal consistency for serverless computing". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 83–97.

[55] Chao Zhang, Guoliang Li, and Tao Lv. "HyBench: A new benchmark for HTAP databases". In: *Proceedings of the VLDB Endowment* 17.5 (2024), pp. 939–951.

[56] Chao Zhang et al. “HTAP databases: A survey”. In: *IEEE Transactions on Knowledge and Data Engineering* (2024).

[57] Chao Zhang et al. “HTAP Databases: A Survey”. In: *IEEE Transactions on Knowledge and Data Engineering* 36.11 (Nov. 2024), 6410–6429. ISSN: 2326-3865. DOI: `10.1109/tkde.2024.3389693`. URL: `http://dx.doi.org/10.1109/TKDE.2024.3389693`.

[58] Haoran Zhang et al. “Fault-tolerant and transactional stateful serverless workflows”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 1187–1204.