Towards Off-the-Shelf Real-Time Transactional Analytics On Cloud-Native Database Systems

By

Elena Milkai

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: May 5, 2025

The dissertation is approved by the following members of the Final Oral Committee:

Xiangyao Yu, Assistant Professor, Computer Sciences

Jignesh M. Patel, Professor, Computer Sciences

Paraschos Koutris, Associate Professor, Computer Sciences

Dimitris Papailiopoulos, Associate Professor, Electrical and Computer Engineering

Acknowledgments

This dissertation wouldn't have been possible without the support, encouragement, and inspiration of many people, to whom I'm deeply grateful.

First and foremost, I want to express my deepest gratitude to my advisor, Xiangyao Yu, for his unwavering support throughout my Ph.D. journey. I feel incredibly fortunate to have had him as my advisor. I still remember taking the very first class he taught—it became a turning point for me. At a moment when I was uncertain about continuing in the program, his course reminded me why I loved research in the first place. Xiangyao's enthusiasm, insightful feedback, and constant encouragement shaped every stage of this dissertation—from the earliest ideas to the final drafts. But more than that, he has influenced how I think, how I tackle problems, and how I continue to grow both as a researcher and as a person. I'm especially grateful for his guidance not only during the Ph.D., but also as I take my next steps beyond it.

I'm equally grateful to my co-advisor, Jignesh M. Patel, for his mentorship and support. It's been a real privilege to work with him and to learn from his deep experience. Jignesh has a way of challenging your thinking—always pushing for clarity, ambition, and impact. His perspective added something truly special to this journey. Working with both of my advisors created an environment that was not only intellectually stimulating but also deeply motivating. I'm especially grateful for Jignesh's thoughtful advice and encouragement—both throughout the Ph.D. and as I prepared for life beyond it.

I'm also thankful to Paris Koutris for serving on my committee over the years and for his thoughtful input during research presentations. His questions and suggestions helped me strengthen my work. I'd also like to thank Dimitris Papailiopoulos for being part of my Ph.D. committee.

Special thanks go to Lukas Maas and Vivek Narasayya, my mentors during my internship at

Microsoft Research in Redmond. I'm truly grateful for the opportunity to work on impactful, real-world database problems, and for your thoughtful guidance and feedback on my Ph.D. research. I'm also thankful to Venki Ramarathnam and Christian König—our wide-ranging conversations made the internship even more exciting and meaningful.

I've been lucky to collaborate with several amazing researchers at UW–Madison, including Zhihan Guo, Yannis Chronis, and Kevin Gaffney. Each brought unique insights that helped shape my work. I'm especially thankful to Yannis Chronis for always being ready with helpful advice—on both research and life in academia. I also want to thank my talented labmates—Yifei Yang, Aarati Kakaraparthy, Abigale Kim, Wenjie Hu, Ling Zhang, Xiangpeng Hao, Hokeun Cha, Bobbi Winema Yogatama, Kevin Kristensen, and Martin Prammer. Not only are they some of the best database researchers I know, but I've also learned so much from them over the years. They constantly set a high bar, which pushed me to keep improving and growing. I'm really lucky to have shared this journey with them.

Next, I want to thank my friends and fellow students at UW–Madison, with whom I shared not only research discussions but also truly unforgettable moments during my time in Madison. Some are now far away, some still close by—but all of them are in my heart: Eleni, Eleanna, Vasiliki, Vasilis K., Argiris, Thanasis, Takis, Konstantinos, Vasilis P., Giannis, Rojin, Evangelia, Yannis, and Thanos. Special thanks to my "brother" Nikos Zarifis for always being there for me through both good and difficult times, and to my "sister" Dimitra Giantsidi, with whom I can party the hardest and still brainstorm exciting research ideas—sharing great times in both worlds.

Last but not least, I am deeply grateful to my parents—Christos and Mariola—my brother Michalis, and my extended family (just a few of them—otherwise, I'd need a whole chapter): my cousins Andreas and Marios, my aunt Mimoza, my grandmothers Eleftheria and Polyxeni, and my goddaughter Olivia. Their constant love and support have been a tremendous source of strength and motivation throughout this journey. Without them, none of this would have been possible.

Contents

Acknowledgments **i**

Contents **iii**

Abstract **vi**

1 Introduction **1**

1.1 Motivation 1

1.2 A Systematic Evaluation Framework for HTAP Systems 3

1.3 Off-the-Shelf Real-Time Transactional Analytics 5

1.4 Contributions and Highlights 6

1.5 Overview 7

2 From OLTP to Real-Time Analytics **9**

2.1 Historical Evolution of OLTP Engines 9

2.2 The Rise of OLAP Engines and Data Warehouses 11

2.3 ETL Bottlenecks and the Demand for Fresh Data 12

2.4 Streaming & CDC Innovations 13

2.5 Emergence of HTAP Architectures 15

3 HATtrick: A Systematic Methodology to Evaluate HTAP Systems **18**

3.1 Motivation 19

3.1.1 Design challenges 19

3.1.2 Design classification 20

3.1.3 Current HTAP benchmarks . . . 21
3.2 Performance-centric definition of HTAP systems . . . 22
3.2.1 Throughput frontier . . . 23
3.2.2 Interpretation of the throughput frontier . . . 24
3.2.3 Calculation of throughput frontier . . . 28
3.3 Freshness of HTAP Systems . . . 28
3.3.1 Theoretical definition of Freshness . . . 29
3.3.2 Measuring Freshness Score . . . 30
3.4 Design of HATtrick Benchmark . . . 31
3.4.1 The Schema and Data . . . 32
3.4.2 Workload . . . 33
3.4.3 Benchmark Procedure . . . 35
3.5 Experimental Evaluation . . . 36
3.5.1 Experimental Configuration . . . 36
3.5.2 PostgreSQL . . . 38
3.5.3 PostgreSQL Streaming Replication . . . 41
3.5.4 System-X . . . 45
3.5.5 TiDB . . . 47
3.5.6 Comparison across systems . . . 50
3.5.7 Discussion . . . 51
3.6 Related Work . . . 52
3.7 Conclusion . . . 53

4 HERMES: An Off-the-Shelf Real-Time Transactional Analytics System 54
4.1 Design Goals . . . 55
4.2 HERMES Overview . . . 56
4.2.1 System Architecture . . . 57
4.2.2 HERMES Integration . . . 58
4.2.3 HERMES Design Details . . . 60
4.3 Transactional Analytics with HERMES . . . 69

4.3.1 Design Challenges . . . 69
4.3.2 HERMES' Isolation Levels Solutions . . . 69
4.3.3 Transactional Analytics Workload (TAW) . . . 73
4.4 HERMES Potential Extensions . . . 75
4.4.1 Cache Offloading to HERMES . . . 75
4.4.2 HERMES in a Distributed Setup . . . 76
4.4.3 HERMES Advancing Middle Layers . . . 77
4.5 Experimental Evaluation . . . 77
4.5.1 Experimental Setup . . . 77
4.5.2 HERMES Evaluation . . . 81
4.5.3 HATtrick Evaluation Across Systems . . . 85
4.5.4 TAW Evaluation Across Systems . . . 87
4.6 Related Work . . . 91
4.7 Conclusion . . . 93

5 Conclusions **94**
5.1 Summary . . . 94
5.1.1 A Systematic Evaluation Framework for HTAP Systems . . . 95
5.1.2 Off-the-Shelf Real-Time Transactional Analytics . . . 95
5.2 Future Work . . . 96
5.3 Vision for HERMES . . . 96
5.4 Concluding Remarks . . . 98

Bibliography **101**

Abstract

Hybrid Transactional and Analytical Processing (HTAP) systems aim to unify transactional and analytical workloads within a single platform, enabling real-time insights over fresh data. Current HTAP solutions face two key limitations: the lack of a systematic methodology to evaluate real-time analytics capabilities, and the absence of a non-intrusive architecture that allows organizations to enable real-time analytics using their existing Transaction Processing (TP) and Analytical Processing (AP) engines without costly migrations.

This dissertation addresses these challenges through two main contributions. First, we introduce HATtrick, an intuitive and systematic benchmark designed to evaluate HTAP systems across two orthogonal dimensions: *throughput frontier*, which captures absolute performance and the system's ability to handle concurrent transactional and analytical workloads without interference, and *freshness*, which measures how up-to-date analytical query results are with respect to the most recent transactions. We also propose a visualization method that makes these metrics easy to interpret, helping users understand trade-offs and draw meaningful conclusions across systems. Our evaluation demonstrates that while modern HTAP systems have improved, substantial opportunities for optimization remain.

Second, we propose HERMES, a novel off-the-shelf HTAP architecture that enables real-time transactional analytics using an organization's existing TP and AP engines—without requiring engine modifications or expensive migrations to a new HTAP system. HERMES introduces a lightweight middle layer between the engines and storage, which dynamically merges live transaction logs with analytical reads to ensure query freshness. The design also preserves performance isolation, supports end-to-end transactional consistency, and enables fine-grained control over isolation levels for transactional analytics. We implemented a prototype using MySQL and DuckDB in the cloud and show that HERMES achieves up to 3× higher throughput on transactional analytics

workloads compared to native HTAP systems.

Together, these contributions provide both rigorous tools for evaluating HTAP systems and a practical architecture for enabling real-time analytics in production environments. We hope this work encourages the HTAP community to refine benchmarks, build plug-and-play solutions, and define clear design principles to make real-time analytics accessible to a broader range of organizations.

Chapter 1

Introduction

1.1 Motivation

Real-time analytics continuously ingest, transform, and analyze data as it is generated, enabling timely insights and decisions based on the most current information [137, 41]. Unlike batch analytics, which process data in delayed intervals, real-time systems demand low latency and high freshness to support time-sensitive applications. Use cases include mid-flight fraud detection, real-time recommendations, sub-second cybersecurity alerts, and up-to-date supply chain monitoring [71, 64, 131].

Yet no single class of database system fully satisfies the demands of real-time analytics. Online Transactional Processing (OLTP) engines are optimized for high-throughput, low-latency transactions with strong ACID guarantees—atomicity, consistency, isolation, and durability—but they struggle with scan-heavy analytical queries, often degrading transactional performance [61]. Online Analytical Processing (OLAP) systems and data warehouses are better suited for large-scale, ad hoc analytics over historical data, but rely on periodic ETL (extract–transform–load) pipelines that introduce latency ranging from minutes to hours [39]. These delays result in stale analytical views that fail to reflect current operational state. Even modern ELT pipelines, which delay transformation until after ingestion, still incur overhead from moving, indexing, and preparing data before it can be analyzed [70, 75].

To reduce the days-to-hours latency of traditional batch ETL, modern data architectures increasingly rely on streaming platforms [78, 1, 22, 21, 60] and Change Data Capture (CDC)

tools [47, 108, 13, 68] to capture and process every row-level change in near real time. In these systems, CDC extracts committed transactions from OLTP logs and passes them to streaming engines for real-time transformation and application into data warehouses or lakes. However, because each change traverses multiple independent commit points—the source database, the streaming layer, and the analytical store—there is no inherent guarantee of atomicity across the entire pipeline. As a result, analytical queries can observe incomplete or inconsistent states, where only a subset of a transaction's changes are visible, violating referential integrity or business invariants [75]. Achieving full end-to-end transactional consistency therefore requires not only exactly-once delivery and ordered processing but also atomic commit coordination across systems (e.g., via two-phase commit protocols or ACID-compliant table formats), ensuring that analytical queries reflect only fully committed source transactions.

Amid these limitations, Hybrid Transactional and Analytical Processing (HTAP) systems have emerged as the most promising architecture for real-time analytics [109, 57]. HTAP systems co-locate transactional and analytical processing on a shared, up-to-date dataset, eliminating the need for separate pipelines and enabling immediate analytical visibility into live operational data while preserving full end-to-end transactional consistency. Whether built using a single engine [107, 55, 130, 72, 67] or decoupled engines [66, 93, 145, 91, 58], these architectures aim to deliver strong consistency, low latency, and high throughput—without compromising flexibility or incurring costly data duplication and movement. Despite these advances, two fundamental challenges remain unaddressed in the current HTAP landscape.

First, despite substantial progress in HTAP system design [130, 55, 72, 101, 93, 66, 58, 145, 31, 56, 91, 82, 132, 119, 118, 23, 27, 100], the field still lacks a principled framework for evaluating real-time analytics capabilities. Traditional benchmarks such as TPC-C [15], TPC-H [14], and SSB [110] assess transactional and analytical workloads in isolation, failing to capture how both perform under concurrent execution [109]. More recent efforts—CH-Benchmark [116], HTAPBench [44], and Swarm64 [113]—attempt to address hybrid workloads but often treat transactional and analytical tasks as independent, overlooking the tightly coupled execution patterns and shared-state contention that define true HTAP systems. Furthermore, these benchmarks lack standardized, interpretable metrics—such as data freshness and workload interleaving—leading to fragmented insights and limited comparability. In the absence of a unified evaluation methodology, system

designers struggle to understand architectural trade-offs, and practitioners lack reliable guidance for selecting HTAP-ready platforms.

Second, no single architectural design has emerged as a universal solution for HTAP systems. Single-engine designs [107, 55, 130, 72, 67] offer low latency but often struggle with scalability and performance isolation under increasing load. In contrast, decoupled architectures [66, 93, 145, 91, 58] improve scalability by separating transactional and analytical components. However, their dependence on specific Transactional Processing (TP) and Analytical Processing (AP) engines complicates integration with the existing engines of an organization, often requiring costly and disruptive migrations that are difficult to justify in production environments. Moreover, most existing HTAP systems lack effiecient support for *true HTAP transactions* [109], which interleave transactional logic and analytical queries within a single ACID transaction. These workloads require analytical queries to reflect the latest data, including uncommitted updates from the same transaction. Efficient support for such execution models not only streamlines application development [42], but also enables advanced real-time use cases [73, 149].

This dissertation takes initial steps into addressing these challenges by posing two central research questions: (1) Which metrics and evaluation methodologies best capture the core requirements of HTAP workloads, enabling meaningful comparisons across diverse system architectures? and (2) How can we architect an HTAP system that delivers real-time analytics and efficient support for true HTAP transactions, while preserving data freshness, ensuring high performance, and scaling elastically—without requiring costly migrations or imposing significant engineering overhead on existing infrastructures? By addressing these questions, we develop both a principled evaluation framework and a novel HTAP architecture that aligns naturally with the infrastructure standards of modern organizations. Together, these contributions aim to advance the state of the art in real-time hybrid processing.

1.2 A Systematic Evaluation Framework for HTAP Systems

Building on the limitations identified in Section 1.1—particularly the lack of standardized metrics and evaluation methodologies for HTAP systems—we introduce `HATtrick` [98], a systematic framework for characterizing the performance and behavior of Hybrid Transactional and Analyti-

cal Processing (HTAP) architectures. HATtrick consists of two key components: a performance-centric method for assessing how well a system handles hybrid workloads, and a technique for quantifying *freshness*—that is, how up-to-date the data is in analytical queries.

In practice, real-world workloads rarely fall neatly into purely transactional or purely analytical categories. For instance, TPC-E [12] incorporates both. We formalize this continuum as the *HTAP spectrum*, which spans from fully transactional to fully analytical workloads. A robust HTAP system should maintain high performance across the entire spectrum without favoring one workload type over the other—a property known as *performance isolation*. To capture this behavior, we introduce the concept of the *throughput frontier*—a two-dimensional visualization that summarizes system performance across different transactional and analytical workload mixes. This representation provides an intuitive and informative overview of how each HTAP system responds under hybrid loads, helping to identify performance trade-offs and bottlenecks.

Another critical dimension in HTAP systems is *freshness*, which measures the delay between transactional updates and their visibility to analytical queries. HATtrick includes a practical, empirical method for measuring freshness in deployed systems, offering insights into their ability to support real-time or near-real-time analytics.

Together, the throughput frontier and freshness metrics offer a comprehensive view of an HTAP system's capabilities. We implement these concepts in the HATtrick benchmark, which evaluates systems using a suite of parametrically generated workloads that span the HTAP spectrum. For each workload configuration—or "operating point"—HATtrick records both performance and freshness, enabling systematic and comparative evaluation.

We apply the HATtrick benchmark to a range of HTAP-capable database systems. The resulting throughput frontiers reveal how effectively each system balances hybrid workloads and allocates resources between transactional and analytical components. Freshness measurements highlight how promptly recent transactional updates are made visible to analytical queries. Our results suggest that while current systems show promise, there remains substantial room for improvement in both performance and data recency.

1.3 Off-the-Shelf Real-Time Transactional Analytics

Having developed a systematic evaluation framework in Section 1.2, we now turn to our second research objective from Section 1.1: designing an Hybrid Transactional and Analytical Processing (HTAP) architecture that enables real-time analytics using an organization's existing Transactional Processing (TP) and Analytical Processing (AP) engines—without requiring costly data migrations or significant engineering effort, while still achieving strong freshness and performance.

We introduce *off-the-shelf real-time analytics*, a novel architecture [99] that enables real-time analytical queries, supports pluggable TP and AP engines, and efficiently executes what we term *Transactional Analytics*—our term for systems that support true HTAP transactions.

An off-the-shelf real-time analytics system is constructed using *existing TP and AP engines* with no or minimal modifications to them. The key insight is to introduce a new system layer between the database engines and the storage, which merges the transactional logs with the analytical reads for analytical queries. Unlike existing HTAP databases that conduct this merging functionality within the database engines, we demonstrate the feasibility of performing this outside the TP/AP engines in a non-intrusive manner. This approach avoids the need for compulsory migration, allowing organizations to continue using their existing TP/AP engines. It also achieves fresh queries and delivers performance that is competitive with current HTAP systems.

A core goal of this architecture is to enable efficient execution of *Transactional Analytics*. For high performance, the analytical components run on the AP engine, while the transactional components run on the TP engine. In these systems, achieving transactional analytics at the requested isolation level involves minimal modifications to the internals of the TP/AP engines. The solution relies on coordination between the off-the-shelf system and TP/AP engines for achieving various isolation levels.

To validate our architecture, we developed HERMES, a prototype cloud-based real-time analytics system. HERMES acts as a middle layer between computation and storage, intercepting storage requests from TP engines (e.g., logging to AWS EBS) and AP engines (e.g., reading from AWS S3). It merges log updates with analytical reads in real time and coordinates with the TP engine to enforce the appropriate isolation level on behalf of the AP engine.

We evaluate HERMES using MySQL [106] as the TP engine and FlexPushdownDB [146] and

DuckDB [117] as the AP engines. Our results show that HERMES introduces minimal overhead while preserving compatibility with existing systems. We compare its performance with MySQL and TiDB [66] on standard HTAP workloads and demonstrate a competitive performance-cost trade-off. To further evaluate transactional analytics capabilities, we introduce the *Transactional Analytics Workload (TAW)*, which extends existing HTAP benchmarks with true HTAP transaction patterns. Our results show that HERMES outperforms existing solutions (e.g., MySQL and TiDB) by up to $3\times$, demonstrating the feasibility of off-the-shelf real-time transactional analytics.

1.4 Contributions and Highlights

We list the main contributions of this dissertation.

A Systematic Evaluation Framework for HTAP Systems

- We introduce the concept of the *throughput frontier*, a novel performance metric that captures how well an HTAP system maintains performance isolation across varying mixtures of transactional and analytical workloads.
- We propose an empirical method for quantifying *freshness*, capturing how quickly analytical queries reflect recent transactional updates.
- We present HATtrick, a systematic benchmark designed to evaluate HTAP systems. It generates a range of parametrized hybrid workloads and extracts both throughput frontier and freshness metrics.
- We propose an intuitive visualization technique for representing the throughput frontier and freshness metrics extracted by HATtrick. This allows users to compare multiple HTAP systems and easily interpret trade-offs between performance and freshness.
- We use HATtrick in a range of HTAP-capable systems and demonstrate its utility as a comparative evaluation tool. Our results highlight trade-offs across different designs and reveal significant opportunities for improving both performance and data freshness.

Off-the-Shelf Real-Time Transactional Analytics

- We introduce the concept of *off-the-shelf real-time analytics*, that allows fresh analytics over existing TP and AP engines without compulsory migration to a dedicated HTAP database.
- We define and implement *transactional analytics*, a core capability for modern HTAP systems that supports the execution of mixed transactional and analytical operations within a single ACID transaction.
- We develop HERMES, a system layer that enables off-the-shelf real-time transactional analytics. We use MySQL [106] as the TP engine and FlexPushdownDB [146] and DuckDB [117] as the AP engines.
- We evaluate HERMES and compare it against state of the art HTAP systems. The results indicates that HERMES exhibits comparable performance to MySQL and TiDB on HATtrick [98], while outperforming both by $3\times$ on a transactional analytics workload we introduce, called TAW.

1.5 Overview

We briefly describe the contents of the chapters of this dissertation.

- **Background.** In Chapter 2, we trace the historical evolution from early systems that focused solely on transactions—namely, Online Transaction Processing (OLTP) database systems—to today's systems that prioritize real-time analytics, highlighting how application needs have shifted over the decades of database system development.
- **A Systematic Evaluation Framework for HTAP Systems.** In Chapter 3, we define two key metrics essential for characterizing HTAP systems: the *throughput frontier* and *freshness*. We detail how each metric is computed and visualized, and we introduce the HATtrick benchmark—a parametrized workload suite designed to evaluate HTAP performance across a range of hybrid workloads. Finally, we use HATtrick to compare multiple HTAP-capable database engines, outlining our evaluation methodology and presenting insights drawn from the resulting comparisons.

- **Off-the-Shelf Real-Time Transactional Analytics.** Chapter 4 introduces the *off-the-shelf* architecture—an HTAP system design that leverages existing TP and AP engines to deliver fresh analytical queries and efficient support for *transactional analytics*. We demonstrate this concept through the design and implementation of HERMES, a middleware layer that merges transactional log records with analytical reads without requiring modifications to the underlying engines. The chapter concludes with a detailed performance evaluation of HERMES.

- **Conclusions and Future Work.** Chapter 5 summarizes the key contributions of the dissertation and reflects on the lessons learned throughout the research process. It also outlines potential future directions and extensions of the work presented.

Chapter 2

From OLTP to Real-Time Analytics

This chapter provides a historical overview and analysis of the evolution of database systems, from transaction-focused architectures to today's real-time analytical solutions. We first examine the development of Online Transaction Processing (OLTP) engines in Section 2.1, followed by the emergence of Online Analytical Processing (OLAP) engines and data warehouses in Section 2.2. Next, we discuss the evolution of ETL (extract, transform, load) processes and the growing need for fresher data in Section 2.3. We then cover the advent of streaming platforms and Change Data Capture (CDC) tools in Section 2.4. Finally, we present the current state-of-the-art approach for real-time analytics—Hybrid Transactional and Analytical Processing (HTAP) systems—in Section 2.5.

2.1 Historical Evolution of OLTP Engines

Online Transaction Processing (OLTP) technology has gone through several major stages since it began. In the 1970s, IBM's System R [37] was introduced as a groundbreaking research project that defined many of the basic ideas behind today's relational databases. It introduced the Structured Query Language (SQL) and helped formalize the four key properties of transactions: atomicity, consistency, isolation, and durability (ACID) [61]. System R also demonstrated practical techniques like cost-based query planning and two-phase locking, which allowed multiple users to safely access and update shared data at the same time [54]. These ideas laid the foundation for later systems such as PostgreSQL [114, 138], MySQL [106], and Microsoft SQL Server [97], all of which followed System R's approach to transactions and SQL.

In the 1980s and 1990s, relational databases became solid, enterprise-ready products. IBM DB2 [67], which grew out of System R, focused on speed, reliability, and easy recovery, making it popular in sectors like banking and government [126]. Oracle Database added features such as multi-version read consistency, clustering, and support for transactions across multiple servers, helping companies keep their data safe even if part of their network went down [76]. During this time, Microsoft SQL Server [97] also became well-known, first as a joint project with Sybase and later as its own product tightly integrated with Windows, offering strong SQL performance, full ACID support, and user-friendly management tools. Together, these systems showed that relational databases could handle serious business needs with both speed and reliability.

Starting in the late 1990s, open-source databases made these enterprise features available to everyone. PostgreSQL [114, 138], which evolved from the POSTGRES project at UC Berkeley, brought in Multi-Version Concurrency Control (MVCC), support for custom data types, and strong standards compliance, making it a flexible choice for many uses. MySQL [106] became popular for its ease of use and performance in web applications. When MySQL added the InnoDB storage engine [105], it gained full ACID transactions and MVCC, allowing it to run more demanding transaction workloads. These projects helped spread database technology into startups, small businesses, and the wider software community.

In the 2010s and beyond, new SQL-based systems built for the cloud emerged to meet demands for scale and global reach. Google Cloud Spanner [45] combined SQL with automatic data splitting across many servers and a synchronized global clock (TrueTime) to keep data consistent around the world. Inspired by Spanner, CockroachDB [81] used the Raft consensus protocol to replicate data safely and offered full serializable transactions by default. Amazon Aurora [20] redesigned MySQL and PostgreSQL for the cloud by separating storage from compute, copying data across multiple availability zones, and providing fast failover with minimal downtime.

These stages show how OLTP systems have built on System R's original ideas, adapting to new hardware, deployment models, and scaling challenges—all while keeping the transaction guarantees and reliability that business applications need. While OLTP databases handle high-volume transactions efficiently, they lack built-in support for complex aggregations and ad-hoc analysis. This gap drove the creation of specialized OLAP engines and data warehouses, as discussed next.

2.2 The Rise of OLAP Engines and Data Warehouses

The early 1990s marked the emergence of Online Analytical Processing (OLAP) as a framework for multidimensional data analysis. OLAP systems were typically categorized as either multidimensional OLAP (MOLAP), which stored data in multidimensional arrays for rapid aggregations, or Relational OLAP (ROLAP), which mapped analytical operations to SQL over relational schemas [39]. Early commercial products such as Oracle Express and Arbor Essbase [52] demonstrated the viability of cube-based analytics [51]. Microsoft's introduction of SQL Server Analysis Services [96] in 1998 further propelled OLAP into widespread enterprise adoption by integrating it directly into the SQL Server ecosystem.

In the 2000s, data warehouses became the foundation for business intelligence. Major relational databases added support for materialized views and query rewrite, enabling precomputed aggregations and efficient group-by processing. At the same time, massively parallel processing (MPP) architectures such as Teradata and Netezza distributed data across compute nodes to accelerate performance on terabyte-scale queries [135]. These "data warehouse appliances" provided near-linear scalability and became widely deployed in enterprise analytics.

By the late 2000s, column-oriented storage and vectorized execution began reshaping OLAP performance. Systems like MonetDB/X100 [35] and C-Store [136] demonstrated that storing each column contiguously and processing data in vectorized batches could greatly improve cache efficiency and throughput. Commercial successors, such as Vertica [84] and Sybase IQ [92], adopted these ideas alongside compression and late materialization [16], becoming foundational OLAP platforms for large-scale analytics.

The 2010s brought an explosion of open-source OLAP engines and big data SQL frameworks. Apache Hive [36], Facebook's Presto [125], and Cloudera Impala [121] offered MPP-style SQL over Hadoop and cloud storage, while Apache Spark [148] provided fast in-memory analytics.

Today, OLAP has fully transitioned to the cloud. Platforms like Amazon Redshift [62], Google BigQuery [95], and Snowflake [46] offer serverless, elastic MPP query engines with decoupled compute and storage. This architecture enables multiple processing clusters to independently scale against a shared data layer. These systems combine the best of previous decades—columnar storage, vectorization, MPP parallelism, materialized views—while adding cloud-native elasticity

and high concurrency, setting the foundation for real-time and hybrid analytics architectures.

Despite advances in OLAP performance, these systems still rely on periodic data ingestion through ETL (Extract–Transform–Load) pipelines. The next section examines the evolution of ETL solutions and their impact on data freshness.

2.3 ETL Bottlenecks and the Demand for Fresh Data

Traditional ETL (Extract–Transform–Load) pipelines have long served as the backbone of data warehousing, but they inherently introduce significant latency. Enterprise tools like Informatica PowerCenter [103] and open-source platforms such as Talend [134] have been widely used to orchestrate complex batch integration—extracting data from operational databases during off-peak hours (e.g., nightly), transforming it on intermediate servers, and loading it into a separate warehouse. While this batch-oriented approach met historical needs for periodic reporting, it left analytics operating on stale snapshots. In practice, data in OLAP warehouses was often out of date—ranging from one day to one week old due to the delays inherent in ETL and the separation between transactional and analytical systems. As organizations shifted toward digital, always-available operations, this latency became untenable, creating an urgent need for fresher data and faster, more responsive insights.

One approach to reducing ETL latency was the adoption of micro-batching techniques. Micro-batch processing accelerates traditional batch ETL by collecting and processing smaller batches of data at frequent intervals—sometimes as short as seconds or minutes—rather than waiting for an entire day's accumulation. This approach served as a stepping stone between slow, coarse-grained batches and true continuous streaming: it improved data freshness while reusing familiar batch-oriented tools and infrastructure. For example, frameworks like Apache Spark Streaming [123] introduced micro-batch execution models, enabling near-real-time ETL by slicing incoming data into small, time-based chunks (e.g., every few seconds). By running ETL pipelines more continuously—hourly or even sub-minute—organizations could significantly reduce data ingestion delays compared to traditional overnight processing. However, even with micro-batches, notable delays persisted, and operational complexity increased as systems had to manage higher-frequency job scheduling and maintain data consistency across multiple stages.

Cloud-based ETL services further alleviated traditional bottlenecks by offering elastic scaling and fully managed infrastructure for data pipelines. Platforms such as AWS Glue [124] and Google Cloud Dataflow [59] made it easier to build, schedule, and operate ETL workflows in a serverless environment, handling large data volumes with minimal operational overhead. AWS Glue, for example, is a fully managed service that automatically scales jobs from gigabytes to petabytes and allows users to design pipelines without manually provisioning infrastructure, leveraging Apache Spark [148] under the hood. Similarly, Google Cloud Dataflow [59] provides a unified model for batch and streaming pipelines, enabling scalable ETL pipelines and real-time stream analytics on serverless infrastructure. These cloud-native ETL tools significantly reduced the turnaround time for loading new data into warehouses or data lakes by supporting continuous or on-demand ingestion and transformation. Nevertheless, even managed cloud ETL pipelines often rely on micro-batching or triggered jobs, which may still introduce latencies of minutes or more. The growing need for truly up-to-the-minute data ultimately pushed the industry toward streaming data pipelines and real-time change capture.

Modern requirements for real-time analytics and operational intelligence have thus pushed beyond traditional ETL. Organizations realized that waiting even hours for overnight ETL could hinder decision-making in fast-paced environments. The limitations of batch processing, even accelerated by micro-batching, set the stage for streaming data pipelines and change data capture tools.

2.4 Streaming & CDC Innovations

To achieve fresher data with lower latency, the data engineering community embraced streaming [78, 1, 22, 21, 60] and change data capture (CDC) [47, 108, 13, 68] technologies. Rather than relying on periodic bulk transfers, streaming pipelines move data continuously as events occur. Apache Kafka [1], a foundational technology in this shift, acts as a high-throughput, fault-tolerant event bus that decouples data producers from consumers. Kafka enables real-time distribution of changes from databases, sensors, logs, and other sources to multiple subscribers with minimal latency. By replacing batch ETL with continuous event flow, organizations reduced data staleness and delivered fresher insights. Kafka's architecture, which scales horizontally to han-

dle trillions of messages per day with millisecond latencies, made real-time integration at scale practical. Streaming pipelines thus eliminated batch windows and opened the door to continuous, up-to-the-moment analytics.

Expanding streaming capabilities, a new generation of stream processing frameworks and SQL-oriented engines emerged to transform and analyze data in motion. Apache Flink [3] exemplifies this shift: a distributed stream processor designed for stateful computations with high throughput, low latency, and exactly-once consistency guarantees. Flink can ingest unbounded data streams and perform complex operations such as windowing, joins, and aggregations. Critically, it exposes high-level APIs, including a relational SQL layer, enabling continuous queries over live streams much like traditional SQL on static tables. By bridging relational paradigms and real-time processing, Flink and similar engines make streaming analytics accessible to a broader range of developers.

Another key innovation for reducing data latency is Change Data Capture (CDC), which extracts incremental changes (inserts, updates, deletes) directly from database logs in real time. Instead of bulk dumps, CDC tools continuously propagate small deltas as they occur. Debezium [47], for instance, monitors write-ahead logs and streams row-level changes into systems like Kafka. This enables downstream consumers or stream processors to react immediately, without waiting for batch jobs. In the enterprise space, Oracle GoldenGate [108] has long provided log-based CDC to replicate committed transactions across heterogeneous systems with minimal latency. By bypassing traditional ETL, CDC pipelines drastically cut data lag and shift transformation and loading closer to real-time.

In tandem with CDC, integrated streaming data pipelines platforms were developed to simplify real-time data integration. Tools like StreamSets Data Collector [69] provide a graphical pipeline engine that can ingest from databases (with CDC origins), apply transformations, and load into sinks, all in a continuous flow. StreamSets is designed for "smart data pipelines" encompassing streaming, CDC, and batch data without hand coding. For example, a pipeline might use a StreamSets CDC origin to capture changes from an Oracle redo log and immediately route those events through transformations to a target data lake or NoSQL store. Such platforms combine the reliability of CDC with the flexibility of stream processing, allowing data engineers to build real-time ETL flows visually. The result is that fresh data can be delivered to analytics platforms or data lakes within seconds of a transaction occurring, a stark improvement from hours in the

batch ETL paradigm.

While streaming and CDC innovations have greatly advanced data freshness, they also introduce new challenges in system complexity and consistency. Achieving exactly-once semantics in distributed streams, handling out-of-order events, and coordinating across multiple components remain non-trivial tasks. Streaming and CDC pipelines generally maintain a clear separation between transactional processing and analytics, connected by a real-time data pipeline. Although this design favors scalability and modularity, it typically cannot guarantee full end-to-end transactional consistency, as downstream systems reflect changes asynchronously [75]. Ensuring strict consistency would require complex safeguards, which are not easy to be implemented in practice (e.g., via two-phase commit protocols or ACID-compliant table formats). As a result, analytical queries over CDC outputs may observe inconsistent states, violating referential integrity or business rules.

In contrast, Hybrid Transactional/Analytical Processing (HTAP) systems aim to unify transaction and analytical workloads over a single, up-to-date dataset, preserving strong consistency while delivering fresh analytical insights. As we explore next, HTAP architectures represent a fundamental shift toward fully integrated, low-latency, real-time analytical systems.

2.5 Emergence of HTAP Architectures

To meet the dual demands of transaction processing and real-time analytics, database systems evolved toward Hybrid Transactional/Analytical Processing (HTAP) architectures. HTAP systems unify online transactional processing (OLTP) and online analytical processing (OLAP) within a single platform, operating directly on fresh data without the delays introduced by traditional ETL (extract, transform, load) pipelines [109, 57, 89]. Gartner [111] introduced the term HTAP in 2014 to describe systems that integrate transactional and analytical workloads within a single platform, often using in-memory technologies to achieve low-latency performance. By removing the need for separate operational and analytical systems, HTAP architectures allow queries to reflect the current state of data in real time. This integration simplifies system design, eliminates data duplication, and enables faster, more informed decision-making. However, delivering HTAP remains challenging due to the fundamentally different performance requirements of OLTP and

OLAP workloads.

The evolution of HTAP systems has followed several paths, depending on how closely transactional and analytical workloads are integrated [109, 57, 89]. One strategy is the single-engine design, where both OLTP and OLAP run on a unified engine and data store [82, 130, 67, 72, 50]. SAP HANA is a landmark example: it combines row-based storage (for fast transactions) and columnar storage (for fast analytics) within an in-memory architecture. New transactions first land in a row-store delta and are periodically merged into the compressed column-store, allowing high transactional concurrency and low-latency analytical queries. Oracle Database In-Memory [82] adopts a similar model: OLTP workloads operate on disk-based row tables, while an in-memory column store accelerates analytical queries. The Oracle optimizer transparently directs queries to the appropriate format. These single-engine HTAP systems exploit memory and dual storage formats to deliver fresh, consistent data for both transactions and analytics without needing a separate pipeline. Academic research further advanced single-engine HTAP designs. The HyPer system demonstrated that high OLTP and OLAP performance can coexist on the same data through careful engineering. HyPer is an in-memory relational DBMS that uses multi-version concurrency control (MVCC) and hardware-assisted virtualization to isolate analytical queries from live transactions. By compiling queries and transactions into machine code, it achieves OLTP throughput comparable to dedicated systems and OLAP performance matching specialized engines—all without stalling updates.

Another HTAP approach uses a multi-engine architecture, separating transactional and analytical processing across distinct components but tightly coordinating them through replication [66, 93, 58]. In these designs, OLTP and OLAP engines—potentially with different storage formats or hardware—work together as a single logical database. TiDB [66], an open-source distributed SQL system, exemplifies this model: its TiKV nodes handle transactions with row storage, while TiFlash nodes maintain real-time columnar replicas for analytics. Updates propagate asynchronously using Raft-based replication, ensuring TiFlash maintains causal consistency with TiKV. TiDB directs OLTP queries to TiKV and OLAP queries to TiFlash, allowing analytical workloads to scale independently without disrupting transactions, at the cost of a small freshness lag. SAP HANA clusters [58] offer a similar architecture, with some nodes optimized for OLTP and others maintaining in-memory columnar replicas for OLAP, all within a single tightly integrated

DBMS. Compared to external ETL pipelines, these shared-cluster designs provide fresher data for analytics with lower latency while keeping the OLTP and OLAP components synchronized internally.

Finally, decoupled HTAP architectures adopt a looser integration strategy, evolving traditional data pipelines into more seamless services [145, 31, 91]. In these systems, transactional and analytical subsystems operate independently—often built on different technologies—while asynchronous replication (typically CDC-based) keeps the analytical store nearly up-to-date. The main goal is to preserve OLTP performance while enabling near-real-time analytics. A prominent example is Google F1 Lightning [145], which adds HTAP capabilities atop existing OLTP databases without modification. Lightning streams changes from systems like F1 [127] into a read-optimized columnar store, and a federated query engine transparently combines live OLTP data with Lightning replicas. More recently, systems like Apache Hudi [5] and Delta Lake have adopted similar designs. Hudi ingests change data capture (CDC) streams into cloud storage tables (e.g., S3 [2] or HDFS [128]), allowing SQL engines (e.g., Presto [125], Athena [19], or SparkSQL [26]) to query near-real-time snapshots with minimal lag. Delta Lake [24] similarly applies operational changes through micro-batch or streaming ingestion, maintaining ACID-compliant tables on object storage. Both frameworks enable analytics on fresh, rapidly changing data by managing distributed storage with transactional consistency, achieving near-real-time insights while isolating analytical workloads from OLTP systems.

In summary, each architectural approach presents its own challenges—ranging from maintaining isolation to synchronizing data formats and balancing resource demands—but the payoff is significant: HTAP enables real-time operational intelligence by allowing analytics directly on fresh transactional data. This largely removes the traditional bottlenecks of ETL, with the database itself serving as the point of integration. Today, a wide range of HTAP systems exists across all categories, with designs within each group often converging on similar architectural patterns. Yet no single HTAP architecture has emerged as universally superior. A lively debate continues in both academia and industry over which design best balances freshness, consistency, scalability, and system complexity.

Chapter 3

HATtrick: A Systematic Methodology to Evaluate HTAP Systems

In this chapter, we present the details of the systematic methodology we propose for evaluating HTAP systems—and, more broadly, any system with real-time capabilities. We begin in Section 3.1 by discussing the core goals that an HTAP system should achieve and motivate the need for a new systematic evaluation framework by highlighting the limitations of existing HTAP benchmarks.

In Section 3.2, we introduce our first proposed metric, the *throughput frontier*, which captures how a system performs across the entire HTAP spectrum. This metric measures how well a system shares resources between the transactional and analytical portions of the hybrid workload, and how much one portion impacts the other during concurrent execution. We also propose a visualization method for the throughput frontier that can be adopted by any benchmark.

We continue in Section 3.3 by introducing our second metric, *freshness*, which measures how up-to-date analytical queries are with respect to the latest transactional updates. We discuss the challenges of accurately measuring freshness in real-world systems and explain how we address these challenges in practice. Then, in Section 3.4, we present the design details of our benchmark, HATtrick, which incorporates both the throughput frontier and freshness metrics. We describe the benchmark's schema, data generation, and workload characteristics.

In Section 4.5, we use HATtrick to experimentally compare multiple systems with HTAP capabilities under different configurations. We also walk through the exact evaluation process that users should follow when comparing HTAP systems with HATtrick and how to interpret the

results. Finally, in Section 4.6 we review related work in the field, and in Section 4.7 we conclude the chapter.

3.1 Motivation

In this section, we first present a classification of HTAP systems based on their *performance isolation* and *freshness* properties. Then, we describe existing work on benchmarking HTAP systems. We then motivate the need for our proposed HATtrick benchmark.

3.1.1 Design challenges

Generally speaking, an HTAP system should achieve the following two goals: (i) *performance isolation* — the transactional and analytical workloads should not interfere with each other and (ii) *freshness* — analytical queries should observe the latest transactions' updates.

An H**TA**P database contains two workloads, an OL**T**P workload and an OL**A**P workload, against the same physical database. For simplicity, we refer to these two workloads as the T and the A workloads. T workloads typically include a mix of read and write transactions, each of which operates on a small subset of the database and uses indexes to accelerate search. In contrast, A workloads are mostly read-only and often involve scans, joins, and aggregates of large subsets of the database

An HTAP system achieves ideal performance isolation when each of the T and A workloads achieves the performance as if it was executed independently. This is a desirable behavior since it allows the two workloads to run without one blocking or affecting the performance of the other. The practical challenge, therefore, is to design a system that can share the resources between the two workloads in a way that minimizes the interference between them.

Moreover, an HTAP system should allow every A query to read the latest modifications of the T workload. These modifications produce *fresh* data. We say that an HTAP system achieves perfect *freshness* when there is no delay from the time the T workload commits its changes to the time the A workload is able to process the same data. The challenge is to provide freshness without negatively impacting the performance of the T or the A workloads.

In the next section, we describe how different HTAP designs use different solutions to achieve

performance isolation and freshness.

3.1.2 Design classification

HTAP systems today follow many different designs. We classify them based on their architectures into three categories: (i) *shared design*, (ii) *isolated design*, and (iii) *hybrid design*. Then, we provide some representative examples in each category.

Shared design. Systems that belong to this category execute the T and A workloads in a single engine. They maintain a single copy of data and share resources between the two workloads (e.g., memory bandwidth, CPU cores, and shared caches). Examples of systems that belong to this category include all the traditional relational databases such as PostgreSQL [138, 114], DB2 [67], and Oracle [107] but also specialized in-memory databases such as SAP HANA [130, 55], Hyper [72, 101], L-Store [122], and DB2 BLU [118]. Systems that follow the shared design use various ways to provide isolation between the two workloads. Creating *snapshots* of the main database is one way to create "data replicas" and reduce the interference between reads and writes. So, they use the copy-on-write (CoW) or multiversion concurrency control (MVCC) mechanisms. Each of the systems that we mention above use their snapshot isolation mechanism to achieve fresh analytics. For example, in MVCC every analytical query that arrives needs to traverse lengthy version chains [142] and find the right snapshot.

Isolated design. Systems that belong to this category usually provide compute isolation and dedicated resources to each workload. This is achieved by using different NUMA nodes for each workload or even different machines. Also, two different copies of the data are maintained, which have different representations. For example, row-store format is used in the T engine and column-store format is used for the A engine, which supports efficient data compression for processing high volumes of data in-memory. Examples of systems that belong to this category are BatchBD [93], TiDB [66], SAP HANA SOE's [58], F1 Lightning [145], Wildfire [31], Db2 event store [56], Greenplum [91], PostgreSQL Streaming Replication [112], and the fractured mirrors [49]. An advantage of the systems that follow the isolated design is the mitigation of the interference between the two workloads since there is no sharing of resources. For achieving fresh analytics, the systems above traditionally follow an ETL process. Recent solutions aim to more frequently update the A replica of the data and achieve higher freshness.

Hybrid design. This category combines characteristics from the two previously mentioned designs. Systems that belong to this category usually are in-memory databases which execute the two workloads in a single machine with shared resources but maintain two copies of data with different representations. Examples of systems that belong to this category are Microsoft SQL Server with Hekaton [86, 50], Oracle dual-format DB [82], and SingleStore [132]. Maintaining two copies of the data is a way for these systems to aim for performance isolation. To provide fresh analytics, every analytical query before execution has to fetch the changes from the transactional log or the tail of the T copy.

3.1.3 Current HTAP benchmarks

Existing popular HTAP benchmarks include CH-Benchmark[116], HTAPBench [44], and Swarm64 [113]. We identify important limitations in the current HTAP benchmarks. For each limitation discussed below we briefly discuss the strategy we will follow in HATtrick, the benchmark proposed in this work.

Unable to measure performance isolation. The current hybrid benchmarks cannot identify whether a tested system is achieving performance isolation between the T and the A workloads. HTAPBench and Swarm64 view one of the workloads as the primary, usually the T, and the other as a turbulence of the primary. Their goal is to execute the secondary workload without affecting the target throughput of the primary workload. In HATtrick we view the T and A workloads as equal. Our primary goal is to discover how good the current HTAP systems are at achieving performance isolation when both workloads are equal. The *throughput frontier* metric, which we will discuss in detail in Section 3.2.1, shows how close is a system at achieving performance isolation, how performance scales, and the interference of the two workloads.

Unable to measure freshness. The second limitation of existing benchmarks is that they cannot measure the freshness of an HTAP system. CH-Benchmark is the only benchmark that identifies freshness as an important factor in system performance. They show how the performance is affected by different freshness configurations in the old version of the Hyper [72] database. However, they do not provide any methodology for measuring the freshness of a system. In HATtrick, we provide a method to measure freshness applicable to all HTAP design categories discussed in Section 3.1.2. Our method is simple and can be adopted by any HTAP benchmark

with minimal changes, we provide more details in Section 3.3.

Unable to identify design category. In Section 3.1.2 we categorize HTAP systems based on their architectures. These categories have been also discussed in other research works [109, 119, 57, 65] and they are important to understand and improve an HTAP system. None of the current hybrid benchmarks is able to discover the category of a tested system. HATtrick can extract this information and communicate it to the user in a friendly way. Our evaluation in Section 4.5 will show how HATtrick discovers the correct category for each system.

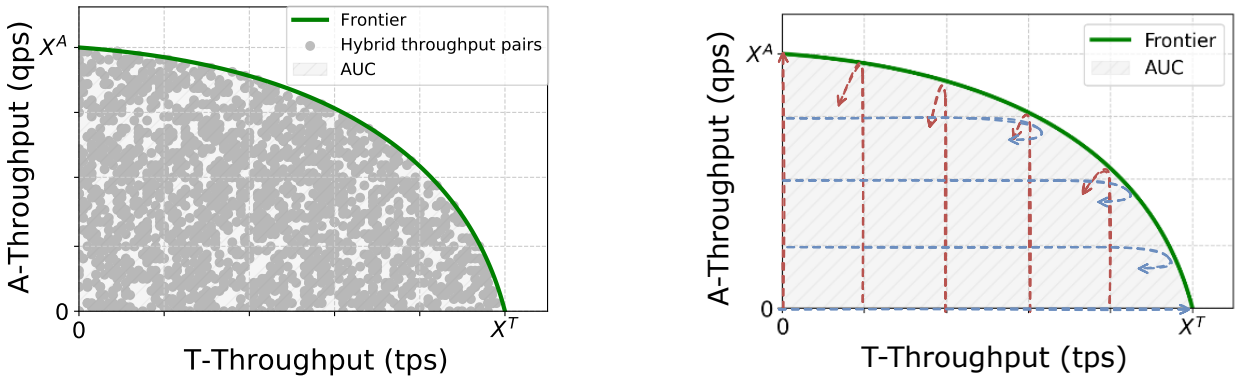
Complicated schemas. The existing benchmarks are all created by combining the schemas of TPC-C [15] and TPC-H [14] benchmarks. The TPC schemas are complex which makes their implementation not straightforward to the users. In the world of OLAP, this has led to the creation of the SSB [110] benchmark which is based on TPC-H but significantly simplified. SSB is widely used due to its simplicity. For our proposed benchmark we extend the SSB schema to support a new T workload which is an adapted version of TPC-C. We discuss in detail the design of HATtrick in Section 3.4. We believe HATtrick can be useful in the same way that SSB has been useful.

Hard to compare multiple HTAP systems. Existing benchmarks do not provide a systematic way to compare multiple HTAP systems, they mostly focus on benchmarking one system. We focus on combining all the information needed to compare different HTAP systems into a small set of metrics. We also provide a visualization of the metrics to make the comparison process more intuitive.

Due to the above limitations, we believe that there is still space for further research in benchmarking HTAP systems and this work is a step towards filling this gap with the proposed HATtrick benchmark.

3.2 Performance-centric definition of HTAP systems

Although many HTAP systems that follow different designs exist in both academia and industry, it is not clear how their performance should be measured and compared with each other. In this section, we introduce the concept of a *throughput frontier* and define the performance characteristics that capture the key properties of an HTAP system.



(a) Sampling method.

(b) Saturation method.

Figure 3.1: An illustration of throughput frontier and different methods of creation.

3.2.1 Throughput frontier

The performance of an OLTP or OLAP system is typically characterized by plotting *throughput* versus the number of clients. However, characterizing HTAP performance is more complex.

We consider a hypothetical HTAP system that serves a mix of T- and A-clients, each of which issues a constant stream of requests. We model the performance of the system using a function S. The input to S is a 2-tuple $(\tau, \alpha) \in \mathbb{N}^2$, where τ and α are the number of T- and A-clients, respectively. The performance of S is a 2-tuple $(x_t, x_a) \in \mathbb{R}^2_{\geq 0}$ where x_t and x_a are the T- and A-throughputs, respectively. We refer to the 2-tuple (x_t, x_a) as the *hybrid throughput* of S.

Fortunately, we can make the simplifying assumption that S is bounded. We argue that the most interesting set of points for HTAP performance characterization are those in the bound. Intuitively, these points represent the maximum *hybrid throughput* that can be achieved by the system across all configurations of clients. Of course, real HTAP systems cannot be perfectly modeled as described above. However, as our experiments demonstrate, it is possible to estimate a reasonably smooth curve that denotes a system's maximum achievable hybrid throughput. For the remainder of this chapter, we refer to this curve as the *throughput frontier*.

Visualizing the throughput frontier is straightforward — it can be represented by mapping the hybrid throughputs to 2D space. Figure 3.1a shows an example of a throughput frontier created by randomly sampling a large number of different workload mixes ((τ, α) pairs) and computing the corresponding hybrid throughputs. The x-axis represents the T throughput measured in completed

successful transactions per second (tps). The y-axis represents the analytical throughput measured in completed queries per second (qps). We denote the maximum transactional and analytical throughput as X^T and X^A, respectively. The throughput frontier is always bounded by X^T in the x-axis and by X^A in the y-axis.

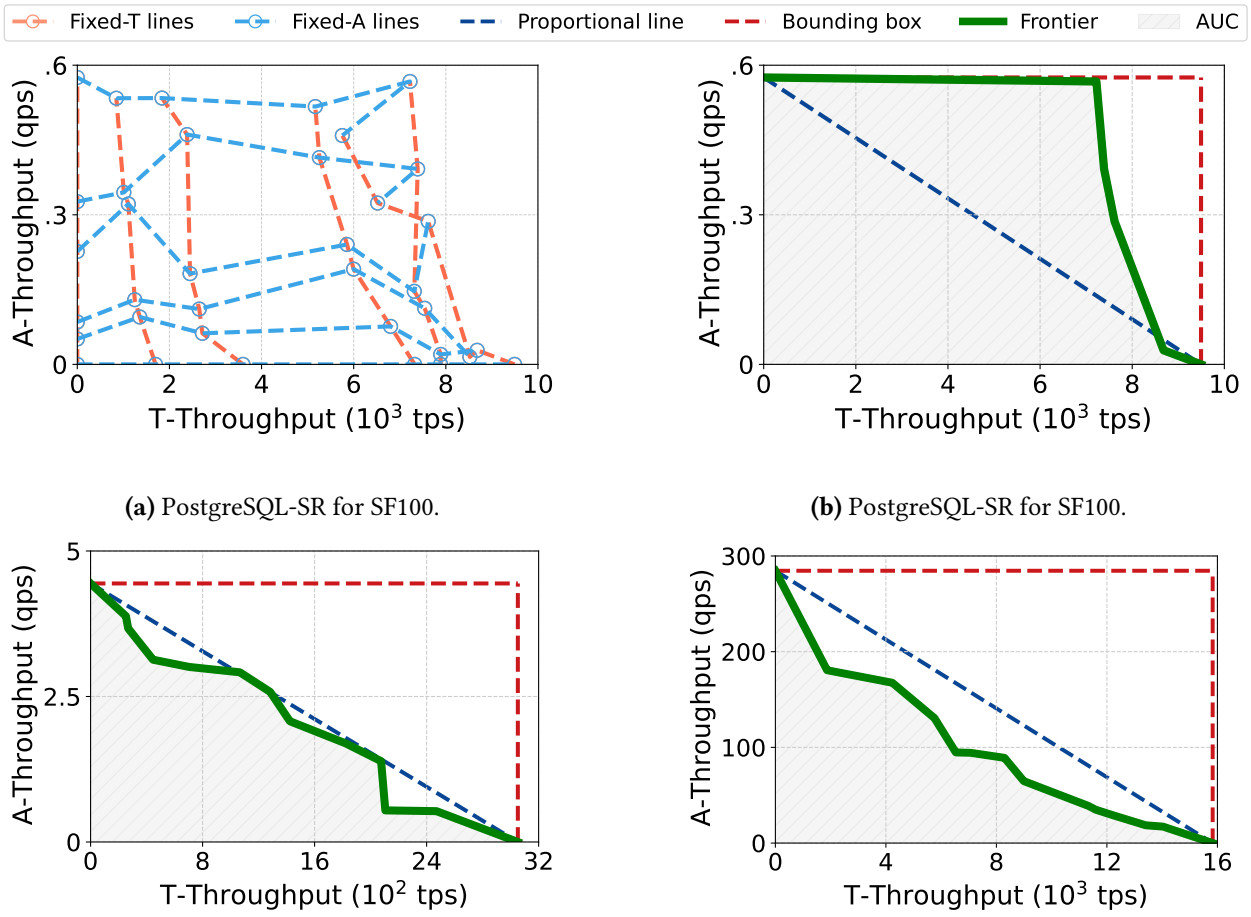
This sampling approach to create the throughput frontier can be prohibitively time-consuming. A more systematic way of computing the throughput frontier is illustrated in Figure 3.1b, called the saturation method. Instead of randomly sampling different workload mixes, we fix either the T or A clients while varying the number of the other type of clients until the performance stops improving. The vertical and horizontal lines shown in the figure correspond to series of measurements where the number of T (or A) clients are fixed and the number of A (or T) clients is varied. We call them *fixed-T* and *fixed-A* lines respectively. We call the graph formed from the fixed-T and fixed-A lines the *grid graph (g)*.

Figure 3.2a shows a real example of a grid graph created for PostgreSQL streaming replication (PostgreSQL-SR) with a 100 GB dataset; more details of the workload and experiment will be presented in Sections 3.4 and 4.5. The real grid graphs do not include pure vertical or horizontal lines. As it shows in Figure 3.2a, the real fixed-T and fixed-A lines are sloped and the distances between the individual lines varies. The shape of the fixed-T and fixed-A lines can explain the way the T and A components of a workload affect each other when they run concurrently. We provide more details in the interpretation of the fixed-T and fixed-A lines in Section 3.2.2. Moreover, in Section 3.2.3, we will discuss the way Figure 3.2a was created by introducing an efficient algorithm.

3.2.2 Interpretation of the throughput frontier

In this section, we discuss the information that can be extracted from the throughput frontier and how this information can be used to interpret the performance of an HTAP system. In general, the throughput frontier quantifies the absolute T- and A-throughput, and their relationship. It is useful for diagnosing performance issues.

To fully understand the performance of an HTAP system, we must consider both the *magnitude* and the *shape* of its throughput frontier. The magnitude of the throughput frontier (i.e., the distance between each point on the frontier and the origin) represents the absolute performance of the system across the entire HTAP workload spectrum. The throughput frontier magnitude is most



(a) PostgreSQL-SR for SF100.

(b) PostgreSQL-SR for SF100.

(c) TiDB for SF10.

(d) System-X for SF1.

Figure 3.2: Examples of our performance-centric definition: (a) Grid graph and (b, c, d) Throughput frontier.

useful when comparing multiple HTAP systems. If the throughput frontier region for some system *A* completely envelops that of another system *B*, we can say that system *A* offers higher HTAP performance than system *B* on the given workload. In contrast, it is also possible for neither throughput frontier region to fully contain the other. In this case, we recommend a deeper analysis, which takes into account additional factors such as the expected workload mix, to determine which system is more desirable. The remainder of this section is dedicated to analysis of throughput frontier shape.

To enhance this discussion, we will use examples of throughput frontiers derived from experiments on real systems. Figure 3.2b, Figure 3.2c, and Figure 3.2d show the throughput frontiers of PostgreSQL-SR, TiDB, and a commercial database which we anonymize as System-X. PostgreSQL-SR and System-X use the serializable isolation level while TiDB guarantees snapshot isolated

reads. A scaling factor of 100 for the HATtrick benchmark was used for PostgreSQL-SR, 10 for TiDB and 1 for System-X. The total raw data size is roughly 80 GB for PostgreSQL-SR, 10 GB for TiDB, and 1 GB for System-X, in all cases the data fits in main memory. More configuration details will be presented in Section 4.5.

We now introduce two annotations to the throughput graph to better understand the shape of the throughput frontier: the *proportional line* and the *bounding box*. The proportional line (p_f), illustrated by the blue dashed line in Figure 3.2b and subsequent figures, is the line drawn from the two extreme points of the throughput frontier. It represents a relationship of *linear dependence* between T- and A-throughput. The bounding box (b_f), illustrated by the red dashed rectangle in Figure 3.2b and subsequent figures, is the rectangle formed by the extreme points of the throughput frontier (i.e., $0 \leq x \leq X^T$ and $0 \leq y \leq X^A$). The bounding box represents *independence* between T- and A-throughput.

In subsequent paragraphs, we explain how the proportional line and the bounding box aid in the analysis of the throughput frontier. We consider three general throughput frontier patterns. The first is a throughput frontier that is *close to* the proportional line. The second is a throughput frontier that is *well above* the proportional line and close to the bounding box. The third is a throughput frontier that is *well below* the proportional line and close to the axes. While it is conceptually useful to think of these patterns as separate cases, note that a real system may exhibit a throughput frontier with any combination of patterns. Here, we separately consider each pattern only to build intuition about the throughput frontier.

Close to the proportional line. As described earlier, the proportional line represents a linear relationship between T- and A-throughput. The proportional line is named as such to emphasize the tradeoff between T- and A-throughput: in an HTAP system whose throughput frontier remains close to the proportional line, any increase in T-throughput is accompanied by a *proportional* decrease in A-throughput, and vice versa. HTAP systems that exhibit this behavior are attractive for their predictable performance. An example of a system and workload configuration that produces a frontier with this pattern is TiDB with SF10, as shown in Figure 3.2c.

Above the proportional line, close to the bounding box. As described earlier, the bounding box represents independence between T- and A-throughput. In an HTAP system whose throughput frontier is well above the proportional line and close to the bounding box, it may be possible to

increase T-throughput with minimal impact on A-throughput, and vice versa. HTAP systems that exhibit this behavior are attractive for their performance isolation. An example of a system and workload configuration that produces a frontier with this pattern is PostgreSQL-SR with SF100, as shown in Figure 3.2b. Note that, by definition, the throughput frontier of every HTAP system will always be within the bounding box.

Below the proportional line, close to the axes. Qualitatively, the degree to which a throughput frontier is below the proportional line and close to the axes represents the amount of negative interference between the T- and A-portions of the workload. A throughput frontier that is well below the proportional line is an indicator of poor HTAP performance and may indicate contention for resources in the system. Identification of this pattern may be useful in diagnosing performance issues. An example of a system and workload configuration that produces a frontier with this pattern is System-X with SF1, as shown in Figure 3.2d. Importantly, the size of the database in this configuration is comparatively small, which results in increased contention for data items. We find that HTAP systems generally exhibit throughput frontiers below the proportional line for small database sizes. These results will be discussed in more detail in Section 4.5.

Grid Graph

In addition to the throughput frontier, the grid graph provides complementary information regarding *workload preference*, through the slope of the fixed-T and fixed-A lines. Ideally, if there is no workload interference, the grid would be comprised of pure vertical and horizontal lines. This is rarely the case in real systems, the lines tend to be slanted due to the interference between the T and A workloads. The closer a fixed-T or fixed-A line is to be perpendicular to the axes the less the corresponding workload is affected by the increase of the other workload. Figure 3.2a, shows the grid graph of PostgreSQL-SR which corresponds to the throughput frontier of Figure 3.2b. The fixed-T lines of the figure are closer to vertical, are clearly placed and tend to have the same length which reaches the X^A. The fixed-A lines are not smooth since they have fluctuations in the absolute numbers of the T-throughput but they tend to have the same length which reaches the X^T. This means that the interference of the T and A workloads is minimized in PostgreSQL-SR in this specific configuration and that PostgreSQL-SR is not favoring a workload over the other.

We also get *workload preference* information from the throughput frontier but the *grid graph*

provides more resolution at operation areas below the frontier that might be of interest in practice.

3.2.3 Calculation of throughput frontier

In Section 3.2.1, we introduced the saturation method for calculating the throughput frontier (Figure 3.1b). Here, we describe in detail how it works including the creation of the fixed-T and fixed-A lines.

First, we find the number of transactional clients (τ_{max}) that maximize the transactional throughput X^T. To find (τ_{max}), the HTAP system executes the transactional workload with an increasing number of clients, until the transactional throughput does not further increase. The algorithm repeats the same steps to find the number of analytical clients (α_{max}) that maximize the analytical throughput X^A. Note that for any other different workload mix, the DBMS cannot achieve a transactional or analytical throughput higher than X^T or X^A, respectively.

The next step is to collect the data points that create the fixed-T and fixed-A lines. Each line requires a series of measurements, in which the number of *T* (or *A*) clients is fixed and the number of *A* (or *T*) clients is varied. In our evaluation we create six fixed-T and six fixed-A lines, by equally diving the ranges [0, τ_{max}] and [0, α_{max}]. For each line, we collect six points. We found that this configuration provides a good coverage of the space, but the number of points per line collected as well as the spacing of the lines can be tuned to provide better coverage. After all data is collected, we calculate the throughput frontier. The throughput frontier is made up from the highest point of each fixed-T and fixed-A lines.

3.3 Freshness of HTAP Systems

In addition to the performance-centric definition, we need to highlight the importance of an HTAP system to provide fresh analytics. In this section, we introduce the concept of *freshness score* which is used to describe the recency of the data read by an analytical query. We also describe our method that can be used to measure the *freshness scores* of queries in real database systems.

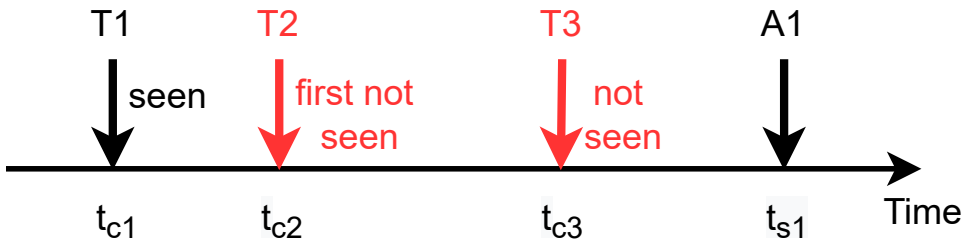


Figure 3.3: Illustration of freshness for analytical queries.

3.3.1 Theoretical definition of Freshness

We consider again a hypothetical HTAP system that serves a mix of T- and A-clients and each of them issues a constant stream of requests. In this definition, we assume both the clients and the HTAP database have access to the same global clock. A transaction is considered committed when the updates of the transaction are applied to the database and are visible to the other transactions. Each analytical query starts and finishes at a particular time based on the global clock and reads a specific snapshot of the operational data. An up-to-date version of the operational data includes all the updates made by transactions that committed before the start of the analytical query. In contrast, a stale version misses some of such updates. We say an HTAP system provides fresh analytics if every analytical query is executed on an up-to-date version of the operational data. Otherwise, we regard it as a system providing stale analytics.

We define *freshness score* of an analytical query A_q as a quantitative measure $f_{A_q} = max(0, t^s_{A_q} - t^{fns}_{A_q})$. $t^{fns}_{A_q}$ is the commit time of the **f**irst transaction **n**ot **s**een by A_q and $t^s_{A_q}$ is the start time of the A_q. Both measures are based on the global clock. Given the definition, the smaller the measure is, the fresher the system will be. The freshness score of A_q is zero when the query can see the updates from transactions committed before the start of the query, which means the snapshot is up-to-date. When the snapshot is outdated, to calculate the freshness score we need to find the time after which the snapshot became stale. This time is equal to the commit time of the first transaction whose updates are not present in the version of the data in which A_q runs. Then, the freshness score of A_q is equal to the difference between the start time of the query and the first unseen transaction measured in time units, e.g., seconds. Figure 3.3 shows an example of transactions T1, T2, T3 and an analytical query A1. Each t_{ci} corresponds to the commit time of the transaction i and the t_{s1} corresponds to the start time of the analytical query A1. We assume

A1 sees all the changes made by transaction T1 but does not see changes by T2 or T3. Therefore, T2 is the first-not-seen transaction and the freshness score of A1 is $f_{A_q} = t_{s1} - t_{c2}$.

Since the A-clients issue multiple requests, each A_q will have a different freshness score. Thus, we define the *freshness score* of an HTAP system as the aggregation of the freshness scores of all analytical queries, denoted as f_{agg}. agg can be any aggregation function such as the average or 95% percentile. Freshness $f_{avg} = 0$ means that the HTAP system can always provide the most recent version of the operational data to all the analytical queries. Freshness $f_{avg} = p$ seconds means that on average the snapshot used by the analytical queries is out-dated by p seconds.

3.3.2 Measuring Freshness Score

The theoretical definition of freshness defined in the previous section can be challenging to measure in a practical system. In particular, we identify the following two challenges:

Challenge 1: No global clock. The theoretical definition of freshness score requires a global clock that is accessible from both clients and the database. A practical system, however, does not have an accurately synchronized clock across different nodes, making it difficult to measure commit time or query start time.

Challenge 2: Hard to identify first not seen transaction. The definition of freshness score requires identifying the first transaction that is not seen by each analytical query. This task is particularly difficult since by definition, the analytical query cannot identify such a transaction. Extra bookkeeping information needs to be kept to identify a not seen transaction.

We introduce the following new algorithm to approximate the theoretical freshness score of a query and resolve the two challenges above. The algorithm has minimal impact to the workload in terms of modifications, and can be applied to general HTAP benchmarks.

To resolve the first challenge, we decide to conduct all time measurement on the client side. In particular, the commit time of each transaction is the time when the transaction result is returned to a client. The start time of an analytical query is the time when the query is sent to the database. This solution avoids clock synchronization across database nodes, and the freshness score is consistent with what the client observes.

To solve the second challenge, we need to first ensure that a client knows the results of which transactions each analytical query should observe, and second be able to tell which transactions the

analytical query actually observed based on the returned result. We introduce a set of lightweight tables $FRESHNESS_j$, where $j \in [1, \tau]$ and τ is the number of transactional clients. For each transactional client j we create one such table that acts as a synchronization point. We also update the transactions and analytical queries in the workload such that they update and read the corresponding table.

Each $FRESHNESS_j$ table contains only one integer field, which is the ID of the last transaction from transactional client j. Each transaction will execute extra logic to update the $FRESHNESS_j$ table with its ID. Note that a transactional client submits transactions to the database sequentially with increasing IDs. Therefore, at most one transaction will be updating each $FRESHNESS_j$ table at any given time and the ID in the table will monotonically increase. We deliberately design the $FRESHNESS_j$ tables to be separate (one for each client) instead of storing multiple rows in a single table in order to reduce contention from having different clients updating their IDs concurrently. Thus, the transactional latency is not affected by the table locking protocol of each database.

To identify which transactions are observed by an analytical query, we modify each query to read all $FRESHNESS_j$ tables and return the contents to the client. Specifically, we union the $FRESHNESS_j$ tables and cross-join the result with the original query. If a query is executed against a consistent snapshot, the returned IDs define the transactions observed by the query — transactions with larger IDs are not observed by the query. This way, we successfully identify the first-not-seen transaction and can calculate the freshness score.

Note that the algorithm described above works well when analytical queries are serializable or snapshot isolated, where reads of data tables and freshness tables are consistent within a query. If the queries are executed with lower isolation levels, one way to measure freshness is to embed the $FRESHNESS_j$ information into each tuple at the cost of higher overhead. All the systems we measure run with at least snapshot isolation and therefore we maintain $FRESHNESS_j$ as separate tables.

3.4 Design of HATtrick Benchmark

We will now move to the design of our hybrid benchmark called HATtrick which we will use to validate our performance centric definition and the freshness concept. HATtrick complements

the throughput frontier and incorporates our freshness measurement method. Thus, can be used to effectively evaluate HTAP systems.

The HATtrick benchmark contains an analytical component and a transactional component. The analytical component is based on the Star-Schema Benchmark (SSB) [110]. We extend the SSB schema to support a new transactional workload which is an adapted version of the TPC-C [15] benchmark. This new transactional workload is the transactional component of HATtrick. We choose the SSB benchmark as our base because it offers a simple schema and query set that is based on the Kimbal [74] definition of data warehouse.

The HATtrick benchmark has three execution steps: (a) the initial population of the database, (b) the warm-up period, and (c) the measurement period. This section discusses the schema, the workload, and the implementation details of the HATtrick benchmark.

3.4.1 The Schema and Data

Figure 3.4 shows the schema of the HATtrick benchmark, which keeps all the SSB entities and relationships almost unmodified. We update the CUSTOMER, SUPPLIER, and PART relations by adding one new attribute to each of them. Also, we introduce a new relation called HISTORY and a series of relations called $FRESHNESS_j$, where $j \in [1, \tau]$ and τ the number of transactional clients. The purpose of adding the new attributes and the HISTORY relation is to support the transactional workload component of HATtrick while, the $FRESHNESS_j$ relations are used in the freshness measurement process as described in Section 4.2. Each $FRESHNESS_j$ table contains only one integer field, the TXNNUM.

Specifically, we add the attribute PAYMENTCNT in the CUSTOMER relation which is an integer that keeps track of the total number of payments each customer makes. Also, we add the attribute YTD in the SUPPLIER relation which is a decimal that accumulates the year to date profits of each specific supplier. Both these attributes are going to be used and updated in transactions which are similar to the payment transaction in TPC-C benchmark.

The HISTORY relation consists of three attributes, the ORDERKEY from the LINEORDER relation, the CUSTKEY from the CUSTOMER relation, and the AMOUNT which is a new decimal attribute. An insertion in the HISTORY relation simulates the process of keeping historic information for a customer payment.

The last change is made in the PART relation where we added the PRICE attribute which is a decimal that stores the cost of each part. The PRICE attribute is used in every transaction that inserts new orders in the LINEORDER relation when the EXTENDEDPRICE and ORDERTOTALPRICE attributes are computed. We describe in Section 3.4.2 how exactly these additions are used in each transaction.

HATtrick benchmark follows the scaling of the SSB benchmark for the initial population of the database, Figure 3.4 shows more details. After the initial population, the sizes of the CUSTOMER, SUPPLIER, PART, and DATE relations remain unaffected by the T workload. However, the transactions of HATtrick change the sizes of the LINEORDER and HISTORY relations by adding

new tuples. The initial size of the HISTORY relation equals the number of the unique ORDERKEYs in the LINEORDER relation, that number is approximately the 25% of the size of LINEORDER relation. The size of each $FRESHNESS_j$ relation is fixed and equal to one.

3.4.2 Workload

There are two components in the HATtrick benchmark, the analytical and the transactional.

Transactions

The HATtrick benchmark defines three transactions modeled after the TPC-C benchmark. Specifically:

New order: This transaction enters a complete order with multiple lineorders through a single database transaction. The new order is inserted to the LINEORDER relation. Specifically, given a random customer name C_NAME, part key P_PARTKEY, supplier name S_NAME, and day of order D_DATE, the new order transaction scans the CUSTOMER, PART, SUPPLIER, and DATE relations to retrieve data. These data are used to create the new entries of the LINEORDER relation. For example, based on the P_PARTKEY, a P_PRICE is retrieved which is used to compute the attributes EXTENDEDPRICE and ORDERTOTALPRICE for that specific line-order. It is worth mentioning that, the dates are sampled from the fixed range of the DATE relation which is seven years of days from 1992 to 1998. Therefore, the new line-orders that are added through the new order transaction do not insert new dates but they keep sampling uniformly from the fixed range.

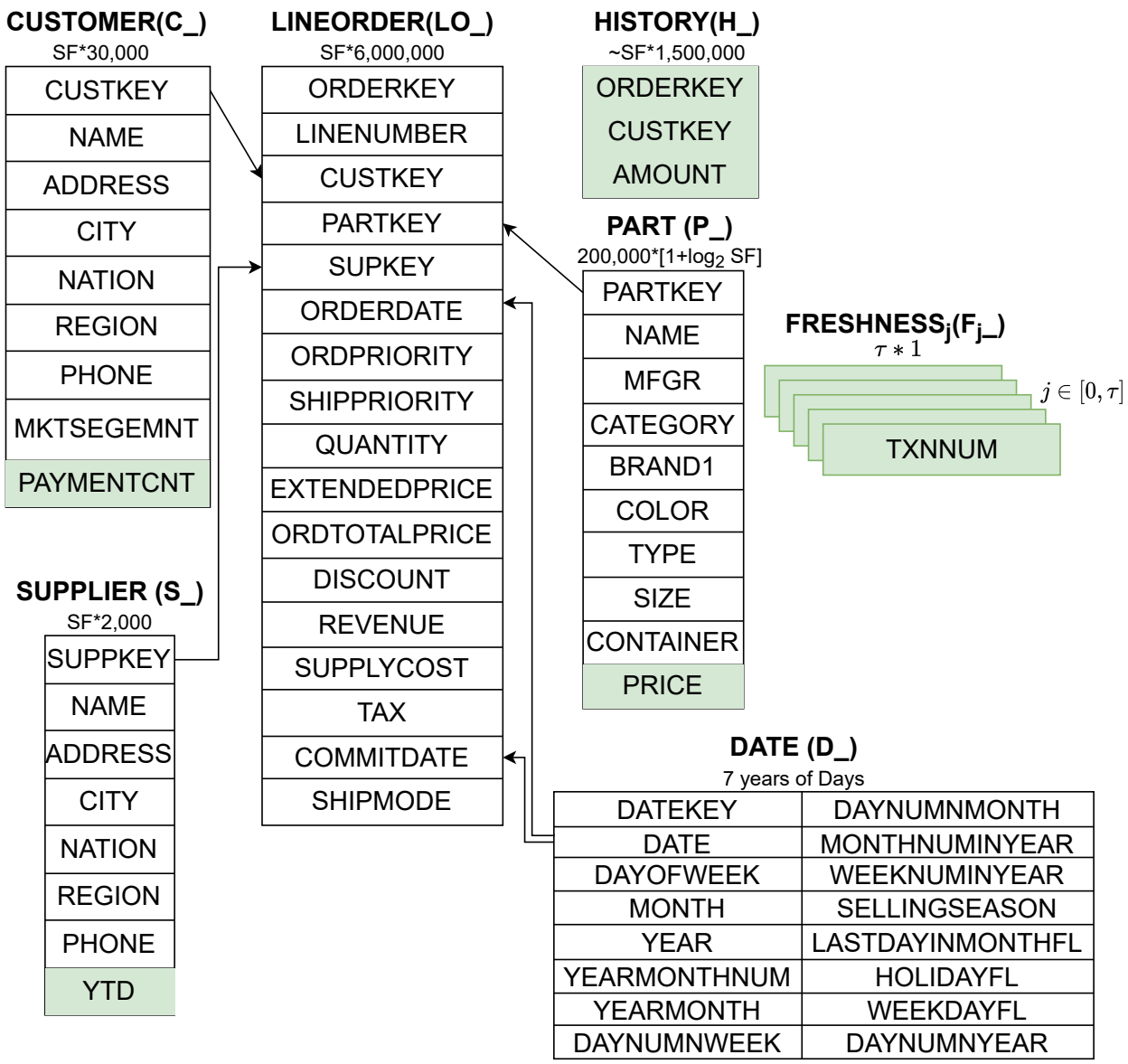


Figure 3.4: The schema of HATtrick benchmark based on modified SSB. New attributes in HATtrick are in shade.

Payment: This is an update transaction which simulates a customer's payment for an order that they already made. The payment transaction updates the customer's total number of paid orders and the year to date balance of the order's supplier which correspond to the C_PAYMENTCNT and S_YTD attributes respectively. The transaction commits after inserting the payment information to the HISTORY relation. The customer is selected by customer name C_NAME 60% of the time and by the customer key C_CUSTKEY the rest of the time.

Count Orders: This transaction is read-only and reports the total number of orders for a given customer. The customer is selected by C_NAME, so seeking on the secondary index of the CUSTOMER relation is required. The total number of customer's orders is retrieved from the LINEORDER relation. Each transaction generated by the T-client j, additionally to its original workload updates the $FRESHNESS_j$ relation with the transaction's ID.

Analytical Queries

The analytical component of the HATtrick benchmark includes all the 13 queries of the SSB benchmark modified to also return the data from the $FRESHNESS_j$ relations. During the measurement period, the transactions add new orders to the LINEORDER relation and thus, the analytical queries process more rows as the time passes. The new entries added through the New Order transaction follow the same specifications as defined in the SSB benchmark. No new dates are added in the DATE relation. Therefore, the predicates of the analytical queries process data created by the initial population of the database and the transactions.

3.4.3 Benchmark Procedure

During operation an HTAP system evaluates transactional requests and analytical requests simultaneously. Each client issues a transaction or a analytical query based on their type and waits for the result before issuing the next. The number of clients is not restricted but the ratio of T to A clients (T:A) is a benchmark parameter.

The T clients issue transactions with the following distribution: 48% New Order, 48% Payment and 4% Count Orders. Each client is independent of other clients. The A clients' queries are organized in batches. An A batch contains all the 13 queries ordered randomly. Once all the

queries in the batch have finished execution the *A* client continues with a new batch of the 13 queries and a new permutation of them. The *A* queries do not delay the transactions, which could happen if a client runs both types of queries. With this design, the tested database systems are free to delay the transactions or the analytical queries in order to improve performance.

3.5 Experimental Evaluation

In the evaluation, we experiment with different databases and configurations. Specifically we study how performance and freshness scores change for different database sizes, isolation levels, physical schemas, replication modes, and deployments (single node and distributed). We use PostgreSQL [138, 114], PostgreSQL Streaming Replication [112] (PostgreSQL-SR), an anonymized System-X, and TiDB [66] for this part of the evaluation. Before we present the experiments, we describe the experimental configuration and the setup of each database system that we use.

3.5.1 Experimental Configuration

System Configuration. The single-node and multiple-node experiments are performed on the same type of servers. Each server has a 2.35Ghz AMD EPYC™ 7452 processor with 32 physical cores, a 512GB RAM, and an SSD disk and runs the Ubuntu 18.04 LTS operating system.

Benchmark Configuration. We experiment with three scale factors of the HATtrick benchmark, SF1, SF10, and SF100, that correspond to raw data of sizes 570MB, 5.7GB, and 59GB respectively. For all scaling factors and database systems that we tested, the data always fits in memory. The clients that submit the transactional and the analytical requests run on the same machine in which the tested database system is installed. For multi-node setup, the clients of the benchmark run in one of the nodes (e.g., in PostgreSQL-SR, the clients run on the same machine with the primary node).

The duration of each benchmark run consists of a warm-up period and the measurement period. Each scaling factor has a different warmup and measurement period duration. For example, for SF100 the warm-up duration is 5min and the real measurement phase is 10min. For SF10 the warm-up is 3min and the measurement phase is 6min and for SF1 2min warm-up and 4min measurement phase. The duration of each period for each scaling factor was selected after conducting a small

experiment, where we discover the appropriate time periods for each phase so the performance is stable. The duration of the warmup and measurement periods remain the same across systems when experimenting with the same scaling factors. Before each benchmark run we reset the data to their initial state.

For each workload configuration (A:T client ratio) we repeat the execution of the benchmark three times and report the average results. For each workload configuration the benchmark reports the T throughput in successful transactions per second (tps) and A throughput in finished queries per second (qps). We also compute freshness score for each A:T client ratio. HATtrick benchmark extracts also the average response time of each transaction type and analytical query.

Evaluated Systems Configuration. The databases we use are PostgreSQL 14, System-X, and TiDB 5.2.0. Because of legal restrictions, we do not disclose the original name of System-X. In PostgreSQL and PostgreSQL-SR, we created all possible B+ tree indexes on the attributes used in the predicates of the transactional and analytical requests. We used this configuration to accelerate both workloads in PostgreSQL for all the experiments except for the one in which different physical schemas are tested. In System-X and TiDB, we created all needed B+ tree indexes for accelerating the transactional requests. Both System-X and TiDB provide an additional column based representation of the data to speed up the analytical requests. Stored procedures were used to execute the transactional requests and prepared statements to execute the analytical requests in PostgreSQL, PostgreSQL-SR, and System-X. Prepared statements were used to execute both the transactional and analytical requests in TiDB since stored procedures are not yet available. Finally, for all databases we disabled the option of intra-query parallelism since it leads to over-utilization of the resources when multiple analytical requests are executed in the database.

Reported Results. For each experimental configuration, we report three plots that correspond to the fixed-T lines, the fixed-A lines, and the throughput frontier, respectively. We generate the fixed-T and fixed-A plots as described in Section 3.2.3. We then compute the throughput frontier from the fixed-T and fixed-A data, also as described in Section 3.2.3. Each figure in this section was generated with this method. In addition to the throughput frontier we also compute the freshness scores as described in Section 4.2. We report the 99th-percentile of the freshness scores for the T:A client ratio points 20:80 (f_2), 50:50 (f_5) and 80:20 (f_8) measured in seconds.

3.5.2 PostgreSQL

In this section we run the HATtrick benchmark in PostgreSQL 14 and show results for different scale factors, isolation levels, and physical schemas. Our results show that there is a negative interference between the T and A workloads in all the scale factors, isolation levels, and schemas. This leads to a throughput frontier that is either below or close to the proportional line. Finally, PostgreSQL is able to provide a zero freshness score in all the experiments, which is expected based on its architecture.

System design. PostgreSQL is a relational database management system (RDBMS) designed primarily for transactional processing. However, like many other traditional databases, PostgreSQL can also serve hybrid workloads. PostgreSQL uses multiversion concurrency control (MVCC) in which readers never block writers, and vice versa. In this set of experiments, we use the serializable isolation level.

Varying scaling factors. Figures 3.5 shows the performance results of HATtrick benchmark for PostgreSQL in three different scaling factors. The fixed-T lines and the fixed-A lines for SF1 have a non smooth behavior. As the number of fixed T/A clients is increased, the lines become more slanted. This shows that the increase of the $T(A)$ clients across the fixed-A(fixed-T) lines affects negatively the $A(T)$ throughput. In general, the behavior of the fixed-T and fixed-A lines shows that as the number of the T and A clients increases, the two workloads are competing for compute resources and data. The small size of the database contributes more to this behavior [120, 115, 144] since many transactions update the same rows which due to locking leads to increased waiting times. The throughput frontier for SF1 is always below the proportional line, suggesting negative interference between the two workloads.

Moving to SF10, the fixed-T and fixed-A lines continue to have a slanted behavior. However, the frontier is now moving closer to the proportional line. This means that an increase in the T throughput is accompanied by a proportional decrease in A throughput, and vice versa. Thus, the resource sharing between the two workloads is more efficient than in SF1. Compared to SF1, we see a big drop in the maximum A throughput due to the increase of the database size which leads to bigger answer sizes. In terms of maximum T output there is also a slight reduction compared to SF1.

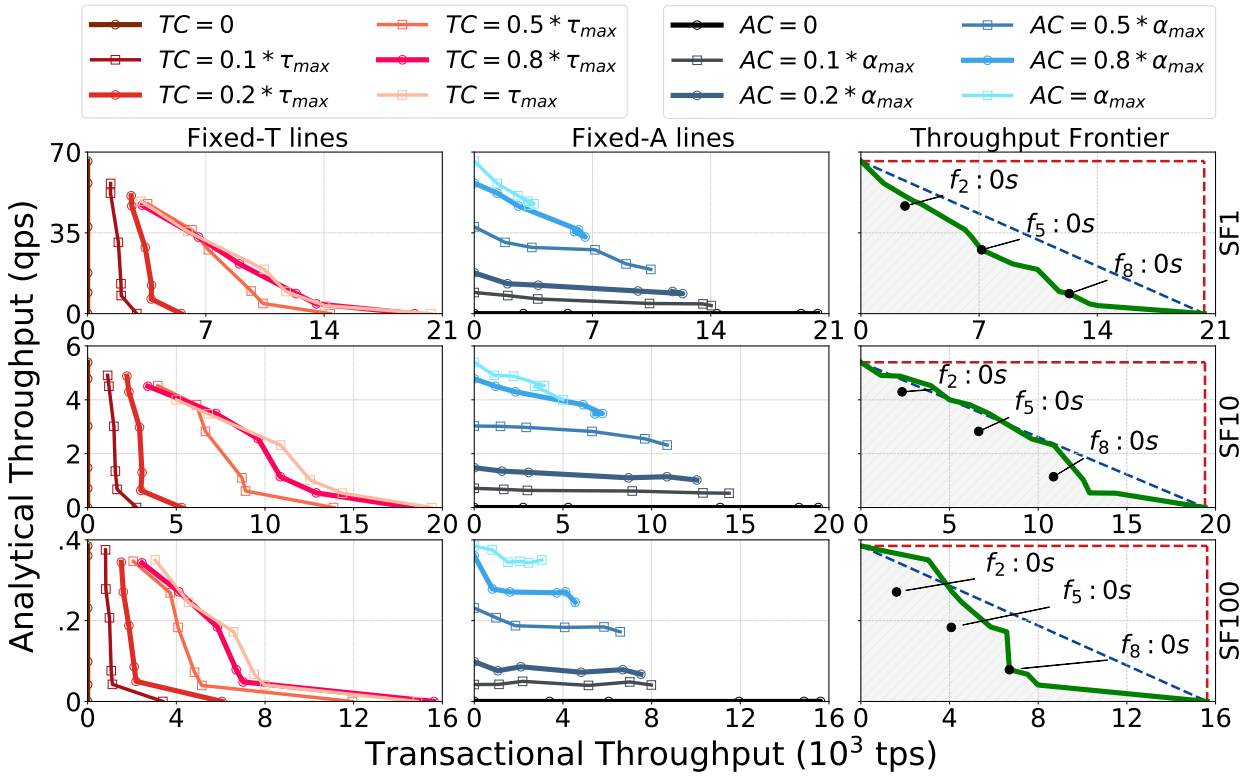
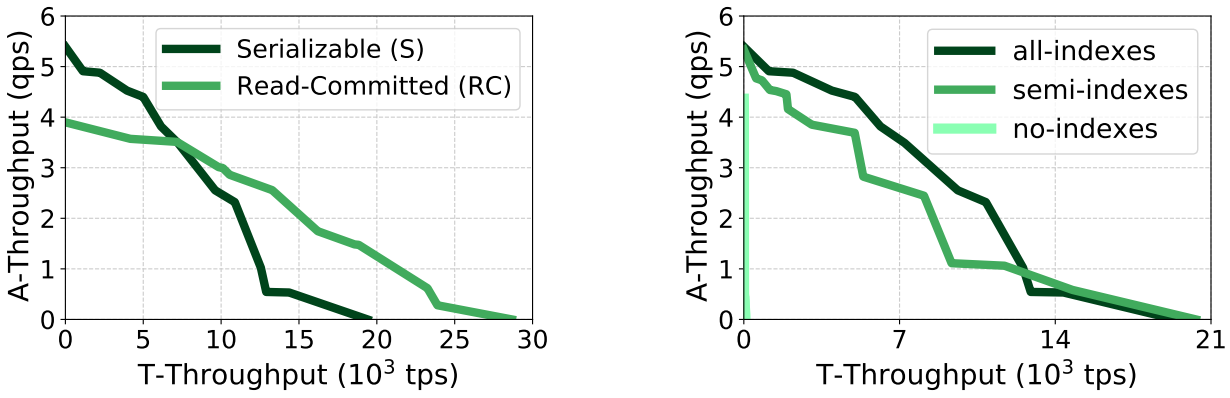


Figure 3.5: PostgreSQL for different scaling factors.

In SF100, the fixed-A lines are not significantly affected by the increase of the T clients. On the contrary, the fixed-T lines are the ones which are extremely affected by the increase of the A clients. Thus, the fixed-A lines tent to be parallel and to all have the same length while the fixed-T lines have a slanted behavior. As a result, the throughput frontier of SF100 is for the half part above or close to the proportional line and for the rest part below the proportional line. This indicates bad performance scaling and shows that the database system is not able to serve efficiently the two workloads in parallel because the T throughput is extremely affected by the increase of the A clients. Again, there is a drop in the maximum A throughput compared to SF10 which is related to the increase of the database size. Interestingly, we observe a significant decrease of the maximum T throughput compared to SF10. This is related to the big number of B+ tree indexes that we use to accelerate both the T and A parts of the workload. However, as the size of the database increases, the size of the indexes increases too. Therefore, more time is needed to traverse and update the indexes when the ”New Order” and ”Update” transactions are executed leading to degradation of the T throughput.



(a) Different isolation levels.

(b) Different physical schemas.

Figure 3.6: Within system experiments for PostgreSQL.

For all scale factors, the measured freshness score for all the ratio points is equal to zero. This is expected since PostgreSQL maintains one copy of the data and the updates of the transactions are made immediately available to the snapshot that the data analytical queries are using.

The advantage of PostgreSQL is that analytical requests can run concurrently with the transactional requests by using snapshot isolation. However, the two workloads still need to compete for resources, data structures, and data items, and this becomes worse when the number of the T and A clients are both high.

Varying Isolation Levels. We now use PostgreSQL and experiment with different isolation levels. We show how the throughput frontier captures the behavior differences between isolation levels.

Figure 3.6a shows the throughput frontiers of PostgreSQL in serializable and read committed isolation levels for SF10. The read committed isolation level achieves higher T and A throughput in almost all the parts of the throughput frontier. The throughput frontier of the serializable isolation level achieves a better maximum A throughput. This is because the query optimizer of PostgreSQL chooses different plans for the analytical queries in the different isolation levels. However, in all the other cases the serializable throughput frontier is always below the read committed throughput frontier. This is an expected result and this experiment demonstrates how throughput frontier reveals the behavior of a system in different isolation levels. Another important observation is the position of the two throughput frontiers relative to their proportional line — both throughput frontiers are close to their proportional lines.

Varying Physical Schemas. We now experiment with different physical schemas in PostgreSQL.

The throughput graphs helps us understand the advantages and disadvantages of each physical schema. Again, the results of the throughput frontiers coincide with what we expected to see.

Figure 3.6b shows the throughput graphs when the physical schema of the database changes; the experiment is performed with serializable isolation level and SF10. The three different physical schemas that are compared are the (1) no indexes, (2) with B+ tree indexes that accelerate only the T workload (semi indexes) and (3) with all possible B+ tree indexes that can accelerate the T and the A workload.

In terms of performance scaling, the physical schema with all the possible B+ tree indexes achieves the best results since the throughput frontier is almost always above the throughput frontiers of the other physical schemas. Next in ranking is the physical schema with the semi B+ tree indexes and the worst is the no indexes physical schema. After conducting an analysis we conclude that the different shapes in the throughput frontiers of the three physical schemas are due to the different query plans the optimizer creates for the analytical queries based on the available indexes.

In terms of maximum T throughput, the semi indexes physical schema achieves better performance compared to the all indexes schema. More indexes can affect the T throughput since they need to be updated in every change that transactions make. However, for the rest workload mixes the all indexes and the semi indexes schemas achieve similar T throughputs.

The use of indexes seem to help not only the T workload but also the A queries. PostgreSQL achieves the best results in both workloads when it uses the all indexes physical schema and this is demonstrated by the throughput frontier results.

In both experiments above (i.e., varying isolation levels and varying physical schemas), we show how the throughput frontier can be used for choosing among different database configurations. Our method combines all the needed information in one figure for multiple configurations, thus the users can understand the system's behavior easily and draw conclusions faster.

3.5.3 PostgreSQL Streaming Replication

In this section we use PostgreSQL 14 with streaming replication (PostgreSQL-SR) and we run the HATtrick benchmark for different scale factors and replication modes. The results show that as the scale factor increases the throughput frontier moves above the proportional line, indicating

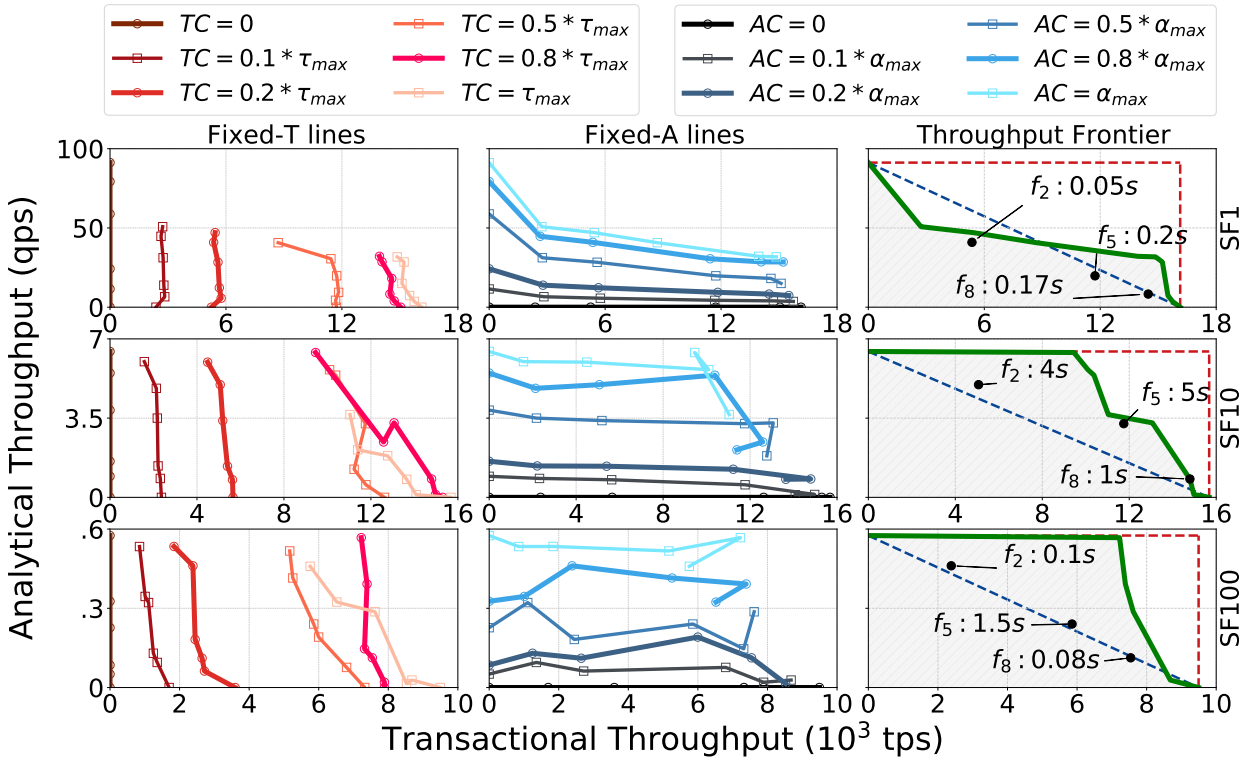


Figure 3.7: PostgreSQL-SR for different scaling factors.

good performance scaling and shows that the database system is able to concurrently serve the two workloads efficiently. However, in all the scale factors we experienced stale queries. Also, the experiments with different replication modes show a trade-off between performance and the freshness scores.

System design. Streaming replication is the most common PostgreSQL replication strategy in which a primary node replicates data to the standby server(s). It is based on streaming WAL records to the standby server(s) as they are generated without waiting for the WAL file to be filled. Thus, it allows the standby server(s) to stay more up-to-date than with the file-based log shipping. The primary node is usually used for transactional workloads while the standby node(s) is read only.

By default PostgreSQL streaming replication is asynchronous, which means that there is a small delay between committing a transaction in the primary and the changes becoming visible in the standby node(s). However, a user can set up different replication modes. One option is the strict synchronous replication in which a transaction in the primary commits only after the updates are

replayed in the standby node(s). This mode can be chosen by setting the synchronous_commit parameter of PostgreSQL-SR to remote_apply. We call this mode RA. In this mode, one can use PostgreSQL-SR to execute HTAP workloads and provide analytics with freshness score equal to zero.

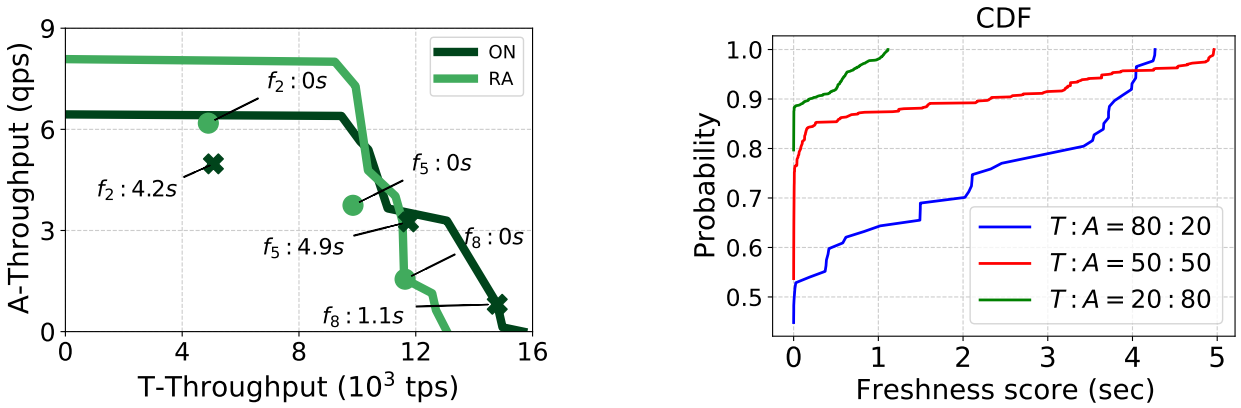
In our first part of the experiments we choose to relax the replication mode and set the synchronous_commit parameter to ON. We call this mode ON. In ON mode, a transaction in the primary node will commit only after the standby server(s) confirms that the transaction record was safely written to the disk of the standby server(s). The difference between the RA and ON mode is that in the ON mode the transmission of the updates happens synchronously but the actual replay of the updates is asynchronous. In RA mode both steps happen synchronously by the commit time of the transaction. Since the transaction updates in ON mode are replayed asynchronously in the standby server(s), we expect to see some stale queries.

Varying scaling factors. In this experiment, we set up PostgreSQL-SR in two identical nodes — one is the primary node and is responsible for the transactional workload and the other is the standby node responsible for the analytical workload. We choose replication mode ON.

Figure 3.7 shows the results for different scale factors. The fixed-T and fixed-A lines for all the scale factors are less slanted compared to the PostgreSQL experiments of Section 3.5.2. This behavior is more clear in the cases of SF10 and SF100 where the lines tend to be parallel and have the same length. This means that the $T(A)$ workload tend to be less affected by the increase of the $A(T)$ clients. Thus, as the scale factor increases the throughput frontier moves above the proportional line and in SF100 is close to the bounding box. These results show that PostgreSQL-SR is good at isolating the performance of T and A workloads and can serve the two workloads efficiently. Interestingly, the results are representative of the system's architecture. The primary and standby nodes have isolated resources and thus the interference of the T and A workloads is expected to be limited compared to HTAP systems that share resources.

As the scale factor increases we see a decrease in the maximum A throughput. In terms of the maximum T throughput there is a significant decrease in SF100 compared to SF1 and SF10. This is again related to the increased size of the indexes as it was discussed in Section 3.5.2.

In Figure 3.7 we report the freshness scores for all the scaling factors in the three T:A client ratios. However, we cannot compare the absolute freshness score of a specific T:A ratio across the



(a) Different replication modes.

(b) CDFs for mode ON.

Figure 3.8: Freshness results for PostgreSQL-SR.

different scale factors since they correspond to different number of clients.

In Figure 3.8b we show the CDFs of the freshness scores for the three ratios in SF10. For the client ratio 20:80, almost 90% of the executed queries return freshness score close to zero and the maximum freshness score seen is 1.1sec. Moving to the client ratio 50:50, the results show that 75% of the executed queries return freshness score close to zero and the maximum freshness score seen is 4.9sec. Finally, the client ratio 80:20 execution reports almost 55% of the queries with freshness score close to zero and the maximum value seen is 4.2 sec. These results indicate that the freshness scores are significantly affected by the number of the T clients. For example, the ratio 80:20 has the lowest percentage of fresh queries ($\sim 55\%$). This is reasonable since more transactional clients are performing more updates in the primary node which need to be send and applied to the standby replica. As a result, the standby node cannot keep up with the high rate of updates and thus, the analytical queries are executed in more outdated snapshots.

Varying replication mode. Next we experiment with different replication modes in PostgreSQL-SR and SF10. The results show that the performance of a system can be affected by the freshness that it provides.

Figure 3.8a shows the throughput frontiers in SF10 for two different replication modes, ON and RA. The figure includes also the freshness scores for the three client ratios in each mode. In RA mode every transaction in the primary server has to wait for the updates to be applied in the standby node before committing. Thus, the standby node remains always up-to-date and the

freshness scores are equal to zero for every query. On the contrary, in the ON mode the replay of the data in the standby server happens asynchronously and thus we see stale queries. Both the throughput frontiers of the ON and RA modes are above their proportional lines which means that the system in both modes can efficiently isolate the performance of T and A workloads. In the first half part of the frontiers the RA is above the ON frontier and for the rest half the ON frontier is above the RA. This means that in the RA mode more analytical queries are executed and in the ON mode more transactions. This is because the RA mode affects the latency of the transactions in the primary node and thus the *T* throughput is lower. The fact that less transactions are executed in the RA mode, leads also to a small increase in the *A* throughput.

In this experiment we see a trade-off between freshness and performance. To achieve fresh analytical queries, *T* performance is sacrificed. This trade-off can be easily understood by using our throughput frontier graphs and freshness measurements. Using the proposed metrics users can choose the appropriate configuration based on their preferences and application requirements.

3.5.4 System-X

In this section we use System-X to run the HATtrick benchmark and show results for different scale factors. The experiments show that System-X can guarantee freshness and as the size of the database increases, it becomes more efficient in handling the two workloads concurrently.

System design. System-X is a memory optimized engine designed to accelerate transactions. The experiments use the serializable isolation level which is achieved by optimistic MVCC without locking. The internal data structures are all latch-free and the threads are executed without stalling or waiting. System-X provides clustered column store indexes which can be also stored in memory and used to accelerate the *A* workload. When System-X is used for hybrid workloads, it can be configured to maintain two copies of the data in memory with each copy having a different data representation. Therefore, transactions can use the row store while analytical queries the column store copy. **Varying scaling factors.** Figure 3.9 shows the results for different scale factors. We identify similar patterns with the PostgreSQL results in Figure 3.5. Both the fixed-T and fixed-A lines are slanted which means that the *T* and *A* workloads are affecting negatively each other in all the scale factors. However, in System-X and SF100 the fixed-T lines are less affected by the increase of the *A* clients. Thus, the frontier of SF100 is above or close to the proportional line. Also,

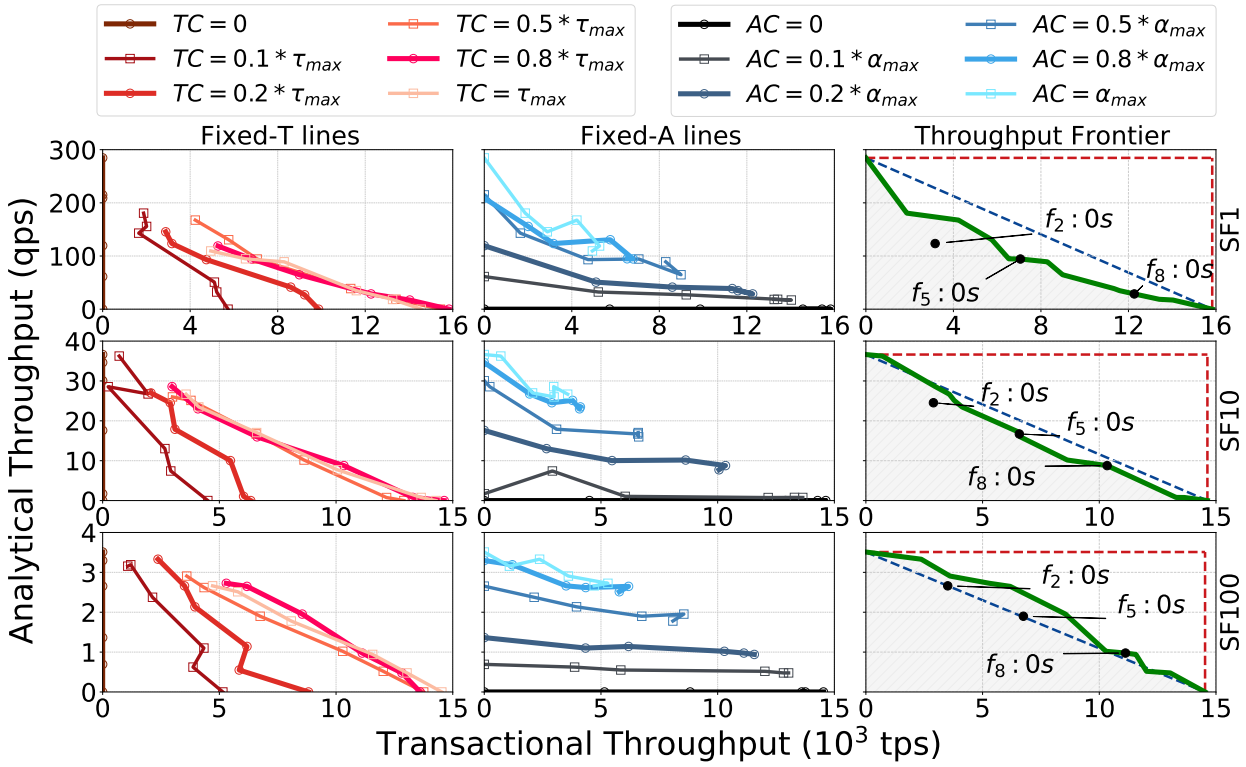


Figure 3.9: System-X for different scaling factors.

the column format of the data and the high efficiency of data compression boost the performance of analytics in System-X compared to PostgreSQL. In terms of maximum T throughput, System-X is able to provide an almost stable performance in all scaling factors since the transactional part does not need to ”pay” any cost for keeping the analytical data fresh.

Although System-X is lock and latch free and maintains two copies of data, the throughput frontiers of SF1 and SF10 capture competition for resources. It is important to mention that, transactions before committing in System-X need to pass a validation phase in which they validate their reads. If a transaction X is in validation phase and another transaction Y reads the changes X made, then Y becomes dependent on X and it blocks until X commits. When many T clients compete for modifying common data, especially in smaller database sizes (e.g, SF1 and SF10), the blocked transactions that are waiting to commit or abort are more numerous. This affects the T throughput as well as the A throughput since each analytical query must synchronise with transaction updates that have not yet been merged with the column store copy. It is important to mention that the frontier of System-X is representative of the system's design. System-X maintains

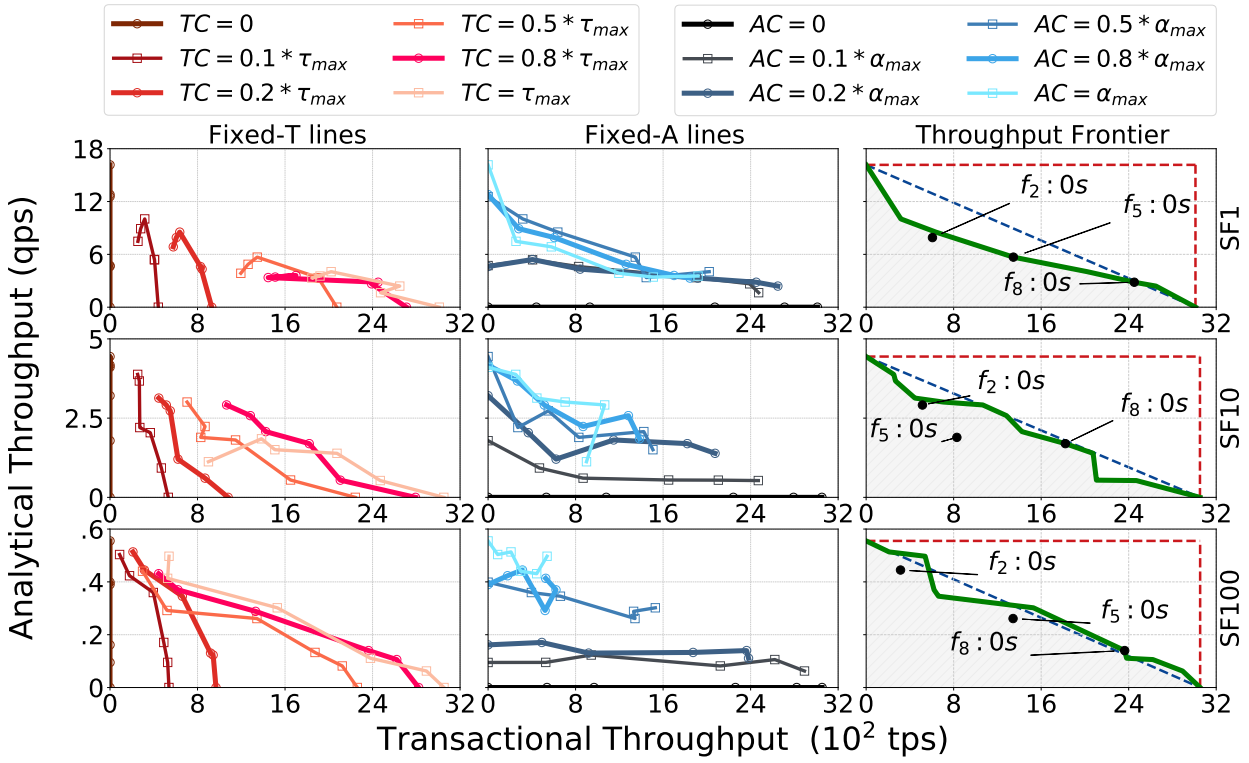


Figure 3.10: Single node TiDB for different scaling factors.

two copy of the data to boost the performance of each workload. However, both workloads share the same resources and thus, the shape of the frontier in SF100 is above or close to the proportional line. Compared to the frontier of PostgreSQL for SF100 of Figure 3.5 the scaling of System-X is better. This is expected since PostgreSQL has only one copy of the data and it is in row format.

For all scale factors, the freshness scores for the three client ratios are equal to zero. This is expected for System-X since based on its design, the latest updates from the operational data are always merged with the analytical data before the execution of a query.

3.5.5 TiDB

In this section we use TiDB and run HATtrick benchmark for different scale factos and deployment configurations (single node and distributed nodes). Our results show that TiDB can always guarantee fresh analytics. In terms of performance TiDB can serve efficiently the T and A workloads as the size of the database increases.

System design. TiDB is a Raft based HTAP system with a distributed storage layer. The storage layer consists of a row-based store called TiKV and a column-based store called TiFlash. The data stored in TiKV is an ordered key-value map partitioned into many *Regions*. Each Region has multiple replicas and a Raft consensus algorithm is used to keep the replicas consistent within a Region. The replicas of each region form a Raft group which is composed of a leader and followers. Each Raft group has also a learner node which asynchronously receives Raft logs from the leader of the group and transform the row-format tuples to columnar format. More specifically, the learner nodes receive a package of logs from the leader which they need to preprocess, decode into row-format tuples, and transform to columnar format. This replication from TiKV to TiFlash makes the fresh data available to the analytical queries and keeps synchronized the two copies.

Single node

Although TiDB is a distributed database, we chose the one server configuration for this experiment. We schedule the transactions to access the TiKV storage and the analytical queries to access the TiFlash storage. We choose for all the experiments the default isolation level of TiDB which is the repeatable read with snapshot isolated reads.

Varying scaling factors. Figure 3.10 shows the results of TiDB. We identify similar behavior with System-X in the fixed-T lines, fixed-A lines and the throughput frontier. In general as the size of the database increases the frontier moves closer to the proportional line. Similar to System-X, TiDB maintains two copies of data in different formats. Although the T and A workloads are executed in different copies, the two workloads share resources. Thus, the frontier for SF100 has a shape above or close to the proportional line. In terms of maximum A performance we see a drop in the absolute value which is related to the increase of the database size. The maximum T throughput remains almost stable across the different scale factors.

In all the scaling factors the measured freshness scores equal to zero. TiDB is designed to always merge the tail of the log with the analytical data before the execution of an analytical query. Therefore, the latest operational updates are always available to the snapshot of the analytical queries.

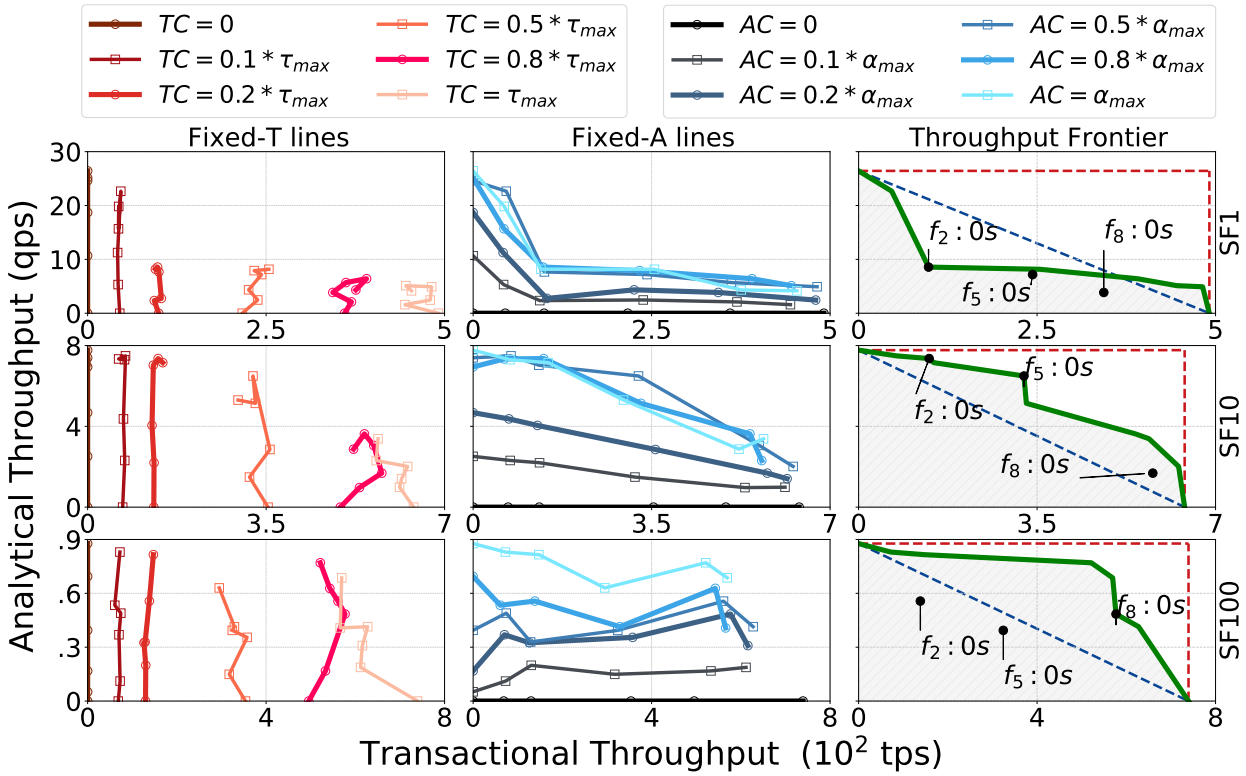


Figure 3.11: Distributed TiDB for different scaling factors.

Distributed nodes

In this experiment we deploy TiDB in distributed mode. TiKV is deployed in three servers and TiFlash in two servers. TiKV serves the transactional requests and TiFlash the analytical requests. The results show that TiDB in distributed deployment can always provide fresh analytics. Also, it achieves a frontier above the proportional line for SF10 and SF100.

Varying scaling factors. Figure 3.11 shows the results of distributed TiDB for the three scale factors. The fixed-T and fixed-A lines have similar behavior to PostgreSQL-SR in Figure 3.7. As the size of the database increases the negative interference of the T and A workloads is minimized and the frontier moves above the proportional line and close to the bounding box.

Compared to the results of Figure 3.10, TiDB in distributed deployment achieves good performance scaling. The shape of the frontier in the distributed deployment is representative of the system's architecture and shows that the system is close to achieving performance isolation. In terms of maximum T throughput there is a significant decrease compared to the one node TiDB

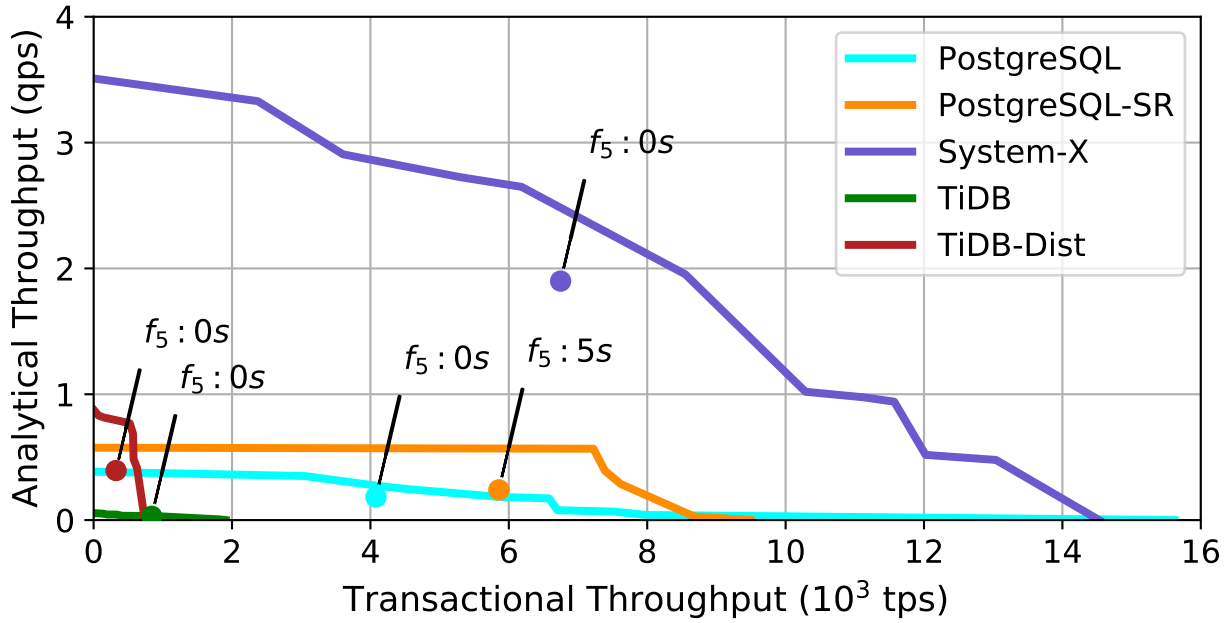


Figure 3.12: Throughput frontiers of compared systems.

which is caused by the high CPU-overhead of the TCP/IP stack and the limited network bandwidth. However, there is an increase in the maximum *A* throughput in the distributed deployment which is attributed to the more available resources in the TiFlash component.

3.5.6 Comparison across systems

In this section we compare all the HTAP systems evaluated above. When comparing different HTAP systems, the process of computing the throughput frontier and freshness scores of each system remains the same. For the comparison we follow a simple rule: If the throughput frontier region of a system A completely envelops that of another system B and system A has lower or same freshness scores compared to B, then system A is better. If not, then we need to dig into more details and to also consider application requirements. Including all the frontiers in one figure helps the user to extract conclusions faster. Figure 3.12 shows the throughput frontiers of PostgreSQL (one node), PostgreSQL-SR (two nodes), System-X (one node), TiDB (one node), and TiDB-Dist (ten nodes) running HATtrick with SF100. Also, we choose to include the freshness scores for the T:A=50:50 client ratio point for each system. Note that systems in this figure may use different number of nodes; we use this example as a point of reference for users that will use

HATtrick to compare performance across systems.

From Figure 3.12, we see that the throughput frontier of System-X envelops the throughput frontiers of all the other systems except for the case of PostgreSQL which has higher value of T_{max} throughput. However, System-X has better A-throughput values and better performance scaling compared to PostgreSQL since its frontier is close to or above the proportional line. We can say that with the workload under test, System-X has the best HTAP performance compared to the other systems.

Between PostgreSQL and PostgreSQL-SR, at first glance, it may not be clear as to which system is better. PostgreSQL-SR has a "higher" frontier and has better A-throughput values, but PostgreSQL has better T-throughput values. Also, PostgreSQL-SR has better performance scaling since the frontier is above its proportional line, while PostgreSQL's frontier– mostly below its propotional line– reveals that the T workload is highly affected by the A workload. Furthermore, PostgreSQL-SR cannot always provide fresh analytics while PostgreSQL does. Finally, PostgreSQL-SR uses twice the amount of hardware resources. Deciding between PostgreSQL and PostgreSQL-SR depends on the user's preferences and the application requirements. If the application requires fresh analytics then PostgreSQL is a better choice. However, if the freshness requirements are not so strict then the application can be benefited from the better performance scaling of PostgreSQL-SR.

Finally, between TiDB and TiDB-Dist, TIDB-Dist has better performance scaling and A-throughput values. However, TiDB has better T-throughput values. This behavior is expected since TiDB-Dist has distributed transactions. In overall, TiDB-Dist has better HTAP performance compared to TiDB.

3.5.7 Discussion

To summarize, the HATtrick benchmark can reveal various aspects of an HTAP system and it can also be used to compare diffe-rent HTAP systems. Specifically, HATtrick can discover information related to absolute T and A throughput, performance scaling in the hybrid workload, the interference of the T and A workloads, and the freshness of the database system. HATtrick combines the above information into a few simple metrics and presents them in a user friendly way, making the process of comparing different HTAP systems easier and more insightful.

We learned the following lessons when conducting this study. First, many current HTAP systems can provide fresh analytics, but this comes with a cost in the T or/and A performance. Second, the results show that the T-throughput is usually severely affected by the number of A-clients; in contrast, the A-throughput is less affected by the number of T-clients. Finally, current HTAP systems cannot achieve complete isolation between the T and A workloads. Future HTAP systems could aim to achieve better isolation between T and A workloads and minimize the impact on the freshness of the analytical query results.

3.6 Related Work

Recent work on benchmarking HTAP systems includes CH-Benchmark[116], HTAPBench [44] and Swarm64 [113]. Their schema is a combination of the TPC-C and the TPC-H benchmarks. The transactions and the analytical requests remain almost unchanged as in the original TPC-C and TPC-H benchmarks respectively.

CH-Benchmark uses the T and A performance along with the CPU utilization as metrics of the benchmark. The authors use the CH-Benchmark to discover how the freshness of the data, the flexibility in the transactions features or expressiveness, and the scheduling of the two workloads affect the performance of the HTAP system. They use Hyper [72] and SAP HANA [130, 55] for their evaluation. The results show that fresh analytical queries can result in a degradation of the system's performance. On the contrary, flexibility and scheduling can boost the T and A throughput.

The difference between HTAPBench and Swarm64 compared to CH-Benchmark is that they view one of the workloads as a primary. Usually the analytical workload is viewed as the disturbance of the transactional workload. The users of the HTAPBench and Swarm64 benchmarks specify a target throughput for the primary workload. Then the benchmarks constantly increase the A queries as soon as they do not affect the target throughput.

HTAPBench and Swarm64 propose a method for generating both new data and requests so that the A queries over recently updated data are comparable across runs. One difference between HTAPBench and Swarm64 is the way the distances between timestamps are computed. In HTAPBench the distances in the timestamps are computed without the need for a training

run. They use the fact that the data interval in TPC-H is 2405 days in all the scale factors. Thus, they use the number of orders in this interval to compute the average time distance between transactions. On the contrary, HTAPBench requires a training run for generating a linear scaling for the timestamps which is then used during the initial data population phase and execution phases.

3.7 Conclusion

In this chapter, we introduce a systematic way to evaluate different HTAP systems. We introduce two new metrics, the throughput frontier and the freshness score. The throughput frontier is a 2D graph that captures the overall performance of a database system in the HTAP space. The freshness score quantifies the recency of the data used by the analytical queries. We also propose a method to measure the freshness score of every HTAP system. We validate these metrics by designing a hybrid benchmark called HATtrick and test it in three different HTAP databases. The results show that the throughput frontier is able to show the performance scaling and the interfere of the T and A workloads. Moreover, the throughput frontier can discover the design category of an HTAP system. Finally, the results show that our measuring freshness method is able to capture the real freshness that each different HTAP system can provide based on their design.

Chapter 4

HERMES: An Off-the-Shelf Real-Time Transactional Analytics System

In this part of the thesis, we present the details of our proposed HTAP architecture. We begin in Section 4.1 by introducing the design principles behind the *off-the-shelf real-time analytics system*, outlining its goals and key ideas. We then provide an overview of our system prototype, HERMES, in Section 4.2, which implements this architecture. More specifically, Section 4.2.1 describes the overall architecture of HERMES, while Section 4.2.2 details how HERMES integrates with different Transactional Processing (TP) and Analytical Processing (AP) engines. The design and implementation of HERMES 's internal components are discussed in Section 4.2.3.

Next, we introduce the concept of *transactional analytics* in HERMES. We first discuss the challenges of executing transactional analytics workloads in an off-the-shelf architecture in Section 4.3.1, and then present our solution for supporting three different isolation levels within the HERMES layer in Section 4.3.2. The design of the transactional analytics workload used in our experimental evaluation is described in Section 4.3.3. We also explore possible extensions and future directions for HERMES in Section 4.4.

Our evaluation of HERMES is presented in Section 4.5. We first describe the experimental setup in Section 4.5.1, followed by the end-to-end performance results demonstrating the integration of HERMES with MySQL [106], FPDB [146], and DuckDB [117] in Section 4.5.2. We then compare HERMES to MySQL [106] and TiDB [66] using the HATtrick benchmark [98] in Section 4.5.3, and evaluate its performance compared to MySQL [106] and TiDB [66] under the TAW workload

across different isolation levels in Section 4.5.4. Finally, we review related work in Section 4.6 and conclude the chapter in Section 4.7.

4.1 Design Goals

This section introduces the two main goals an *off-the-shelf real-time analytics system* should achieve and discusses the approach taken by existing HTAP systems, thereby motivating our solution.

Goal 1: Support for pluggable engines. Most existing HTAP solutions [24, 66, 86, 50, 31, 40, 55, 82, 72, 102, 93, 31, 87, 133, 118, 100, 130, 55, 118, 43] require data migration due to tight integration between their computation and storage engines, making transitions to other compute engines or cloud storage challenging. Systems like F1-Lightning [145] and Hudi [5] offer some flexibility with pluggable TP and AP engines, but only support *near real-time* analytics, with updates delayed by $\sim$10 minutes. This makes them unsuitable for applications that require high freshness, which is critical in HTAP.

Key idea 1: Real-time analytics with existing TP/AP engines. An off-the-shelf real-time analytics system achieves real-time analytics on the latest transactional data without requiring engine migration; instead it uses existing TP/AP engines and storage services. This enables users to select optimized engines and storage for TP and AP, avoiding migration efforts. The trade-off is the need for efficient synchronization to maintain high or perfect *freshness*. A system achieves perfect freshness when each analytical query can read changes of all transactions that have committed (i.e., linearizability). The challenge lies in achieving fresh analytics without impacting TP/AP performance or modifying engine internals. In Section 4.2, we present our architecture, and our evaluation (Sections 4.5.2, 4.5.3) shows that our design avoids performance overhead while matching state-of-the-art HTAP systems.

Goal 2: Efficient Transactional Analytics. An analytical transaction consists of both transactional logic and analytical queries that are executed under the same isolation level [109]. The analytical part contains queries of varying complexity, which are significantly accelerated when processed in specialized engines (e.g., columnar databases) ensuring time and cost efficiency. The results of the queries can then be used to perform more operations on the transactional data (e.g.,

update a table). The analytical query must see the correct data based on the enforced isolation level.

- Application scenario:In fraud detection, real-time data on users' recent behavior is analyzed alongside historical data to proactively prevent or address fraud [17]. This analysis must occur at the same isolation level as other operations within the analytical transaction. Upon detecting fraud, the database should reliably revert to a known state by rolling back the transaction or continuing remaining operations, ensuring accurate fraud detection and robust management of potential fraudulent activities.

Efficiently executing transactional analytics is a major challenge for decoupled HTAP systems. Our evaluation (Section 4.5.4) shows that even leading HTAP systems struggle with performance under such workloads. Additionally, some systems only partially support this functionality, offloading concurrency control to clients—a complex, error-prone approach that adds engineering overhead [42].

Key idea 2: Efficient transactional analytics in off-the-shelf systems. An off-the-shelf real-time analytics system enables efficient transactional analytics by selecting the optimal engine for each workload component—transactional logic is executed in the TP engine and analytical in the AP engine. Its decoupled design, however, challenges consistency and correctness across isolation levels. To address this, analytical queries must operate on an accurate data snapshot. The system achieves this by retrieving log statements that match the correct snapshot and coordinating with the TP engine to enforce it, reducing TP-AP communication. In Section 4.3, we detail these challenges and our solution for supporting transactional analytics across isolation levels.

4.2 HERMES Overview

In this section, we introduce HERMES, an off-the-shelf real-time analytics prototype system, and discuss its architecture, integration with existing engines and design details.

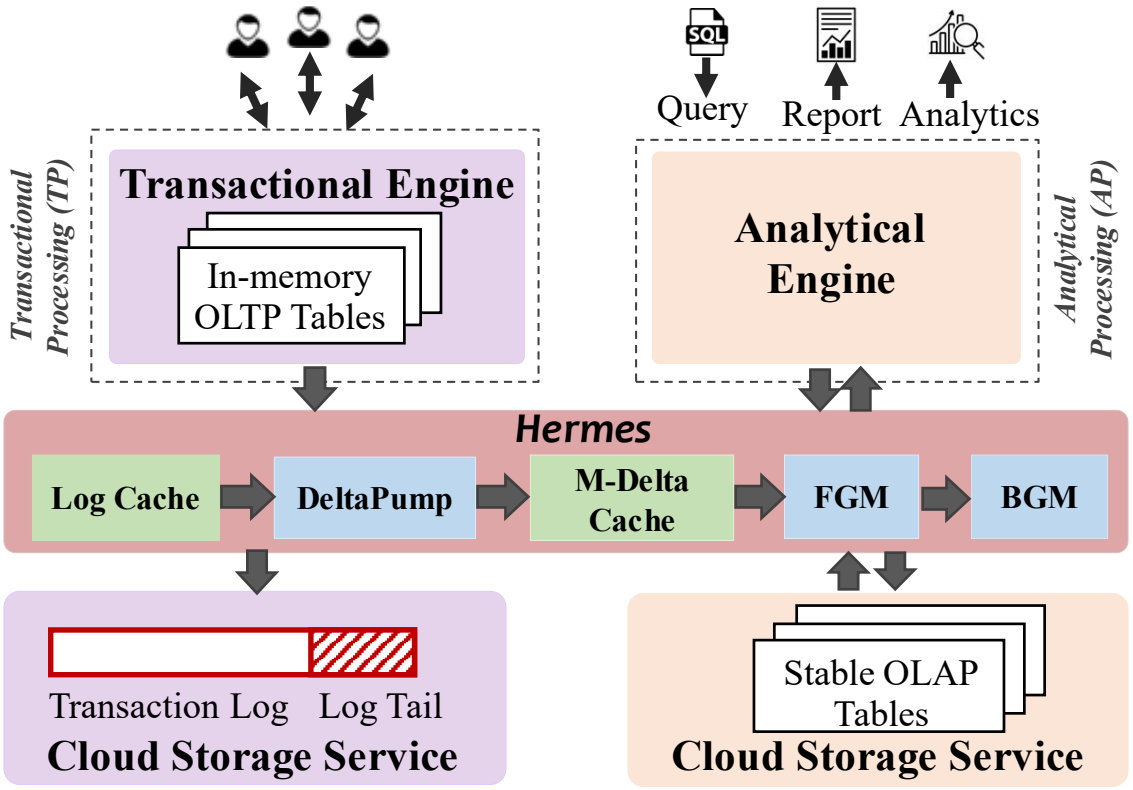


Figure 4.1: HERMES architecture and main components.

4.2.1 System Architecture

Figure 4.1 illustrates the architecture of a hypothetical organization using HERMES. The TP engine handles transactional requests, while the AP engine serves analytical queries. Data storage is decoupled, with AP data stored in a cloud storage service. The transactional log from the TP engine provides fresh data for the AP engine and is also persisted in a cloud storage service. HERMES acts as the synchronization hub between the TP and AP engines.

HERMES includes two in-memory caches: the Log Cache and M-Delta Cache, as well as three services: *DeltaPump*, *Foreground Merge (FGM)*, and *Background Merge (BGM)*. The Log Cache holds the transactional log tail from the TP engine in memory. DeltaPump processes and forwards the log tail to the M-Delta Cache. FGM merges the latest updates from the M-Delta Cache with stable data during analytical reads, while BGM merges asynchronously data from the M-Delta Cache into storage to prevent cache overflow.

Workflow. The TP engine directs the logs to HERMES (i.e, Log Cache), which forwards them to

cloud storage. When an analytical query arrives, the AP engine reroutes cloud storage requests to HERMES. Upon receiving the first request, HERMES triggers the DeltaPump service to transfer log tail data from the Log Cache to the M-Delta Cache. Simultaneously, HERMES retrieves stable data from cloud storage and uses the FGM service to merge it with the latest updates from the M-Delta Cache. The merged, up-to-date data is returned to the AP engine in the same stable format (e.g., Parquet [7]), enabling the rest of query execution to proceed. In the background, HERMES runs the BGM service asynchronously to prevent indefinite M-Delta Cache growth.

Since a transaction is considered committed once the commit record hits storage, the AP engine can, by definition, observe only committed transactions. Consequently, HERMES ensures freshness and snapshot consistency with the current OLTP copy for all queries.

4.2.2 HERMES Integration

In this sections we outline key requirements for the TP and AP engines to support off-the-shelf real-time analytics. In addition, we discuss the HERMES integration details with MySQL [106] as TP engine, and FPDB [146] and DuckDB [117] as AP engines.

TP Engine Interface

In HERMES, the TP engine handles transactional requests, with the transaction log serving as the main interface between HERMES and the TP engine to maintain data freshness in AP engine queries. Off-the-shelf real-time analytics requires the TP engine to provide a row-level log where the updates can be extracted and merged with analytical reads in real-time.

Log Granularity and Hermes Integration. Systems like MySQL [106], SQL Server [50] and IBM Db2 [67] are well suited for HERMES as they can generate row-level log. In contrast, systems like PostgreSQL [114] and Oracle [107] generate logs that capture changes at different levels of granularity such as data pages or blocks. In this case, integration with HERMES requires post-processing of the log, which involves three key steps: (1) identifying the rows that were modified, inserted, or deleted based on the physical changes recorded at the page/block level, (2) decoding these changes and translating them into logical operations that accurately represent the database's behavior (e.g., update, insertion, deletion), and (3) converting the row-level changes into

a standardized format (e.g., Avro) to ensure compatibility. Finally, systems that rely on command-level logging, such as VoltDB [94], are generally incompatible with HERMES; integration requires modifications to the TP engine internals, as post-processing of the log alone is insufficient.

MySQL Integration Details. We integrate HERMES with MySQL [106], an OLTP-optimized database management system (DBMS), by directing its row-based binary log to a network-accessible Log Cache on the HERMES server. For durability, HERMES forwards logs to AWS EBS. This integration requires only a log path update in MySQL's configuration, with no code modifications.

ACID Correctness in HERMES. By preserving TP engine's transactional integrity and merging only committed logs, HERMES ensures ACID compliance for transactions and analytical queries.

- Preserving ACID in Transactions. HERMES acts as a log forwarding layer, leaving TP engine's transaction processing unchanged and preserving its ACID guarantees. When a transaction commits, the TP engine writes its log to HERMES' Log Cache, which forwards it to persistent storage. Once stored, the storage acknowledges the Log Cache, which then confirms to the TP engine. Since the TP engine commits only after this acknowledgment, its logging process remains unchanged. If log forwarding fails, the TP engine does not receive the acknowledgment, triggering its standard recovery.

- Ensuring ACID in Queries. HERMES guarantees atomicity for queries by processing only fully committed transactions, preventing partial writes or exposure of incomplete data to the AP engine. If log forwarding fails, uncommitted log records are discarded, and only fully committed ones are replayed upon recovery. To maintain consistency, HERMES relies on the TP engine's transaction log as the single source of truth, merging only committed records. This prevents the AP engine from observing inconsistent intermediate states. Furthermore, isolation is upheld as HERMES ensures that in-progress transactions remain hidden from the AP engine. Only fully committed log records are made available, preventing anomalies from concurrent execution. Finally, durability is preserved since transactions are considered durable only after being written to the TP engine's dedicated storage. HERMES adheres to this protocol, making log records accessible to the AP engine only after they are persistently stored.

AP Engine Interface

In HERMES, AP engines are critical in serving analytical requests. For real-time analytics, AP engines should direct their storage engine requests to HERMES. This setup renders the analytical storage transparent to the AP engine, shifting the responsibility to HERMES for providing the latest data. Consequently, HERMES necessitates that the AP engine consistently reads data directly from storage. As such, the AP engine cannot leverage its local data cache. To address this limitation, we propose offloading the AP cache to HERMES.

FPDB and DuckDB Integration Details. We integrated HERMES with two AP engines: FPDB [146], a cloud OLAP DBMS, and DuckDB [117], an embeddable OLAP DBMS. For both integrations, we followed a similar approach. Specifically, we redirected scan operators' data requests to HERMES, aligning with each AP engine's data access pattern (e.g., reading data from multiple partitions simultaneously). To direct requests to HERMES, we used an RPC protocol like Apache Thrift [8] for server-client communication. Each request specifies the data HERMES should provide, and HERMES returns the updated data to the AP engine for the rest of the query execution. Overall, we added fewer than 100 lines of code across both engines, primarily to implement HERMES and AP engines communication.

4.2.3 HERMES Design Details

This section outlines HERMES design, focusing on the Foreground and Background Merge algorithms, and its storage organization.

Data Organization

The data in HERMES is organized into three categories: stable data, transaction logs, and deltas.

Stable Data. HERMES manages multiple segments of data during the merging process, as shown in Figure 4.1. The AP data of HERMES are stored in a distributed cloud storage service; we call them *stable data*. Stable data is horizontally partitioned based on the primary key and sorted by the primary key within each partition. Each partition is saved as a separate file in the cloud storage and contains all columns corresponding to the rows within that partition. Currently, HERMES supports stable data in CSV and Parquet formats [7] and can handle other industry-standard

formats (e.g., XML, JSON, Avro) [6] without modifying its internal functionality.

Transaction Log. The row-level transaction log records the history of changes to the TP data as data events. These events capture table row operations such as insertions (INS), updates (UPD), and deletions (DEL). In addition to storing the log in persistent storage, HERMES also maintains it in the Log Cache to enable faster access.

Deltas. HERMES uses DeltaPump to parse the transaction log and extract the most recent updates from its tail, referred to as *tail-deltas (t-deltas)*. Each t-delta contains the after-images of row changes along with the type of change (e.g., INSERT, UPDATE, DELETE). The t-deltas are sorted by the primary key, and DeltaPump assigns a timestamp to each t-delta, marking the time the log tail was parsed.

Upon retrieval, t-deltas are stored in the *M-Delta Cache* alongside previously fetched deltas; called *memory deltas* (*m-deltas*). All deltas share the same data format, but differing in their timestamps. The M-Delta Cache is designed to enhance the performance of foreground merges by keeping the most recent log data in memory. Older m-deltas are asynchronously written to cloud storage during background merges, referred to as *disk deltas* (*d-deltas*).

Foreground Merge (FGM)

In this section we introduce our Foreground Merge algorithm and two optimizations that we designed to improve its performance: the *M-Delta Merge Optimization* and the *BitMap Caching Optimization*.

FGM Algorithm. The problem that FGM algorithm solves can be described as follows: for a particular data partition, the inputs of the FGM algorithm include a stable data file, several m-delta files, and one t-delta file; all of them are sorted based on primary key of the table. The goal is to merge all these data sources, such that if records with the same primary key exist in multiple deltas, only the latest record should appear in the merge results.

Figure 4.2 shows an example of stable data, an m-delta from M-Delta cache, and a t-delta that have already arrived from DeltaPump. For space saving purposes, we assume a single column for primary key (PK), and we include only the primary keys of the data, omitting the rest of the columns. Note that a new column (Type) is added to the m-deltas and t-deltas showing the type of the transactional statement that causes the modification in the data, which can be INS, UPD, or

PK	Columns
998	[…]
999	[…]
1000	[…]
1001	[…]
1002	[…]
1003	[…]

(a) Stable data

M-Delta, TS=50

PK	Columns	Type
1001	<…>	UPD
1002	<…>	UPD
1003	<…>	DEL

(b) M-Delta

T-Delta, TS=100

PK	Columns	Type
998	{…}	UPD
1002	{…}	UPD
1004	{…}	INS

(c) T-Delta

BM(S)	BM(M)	BM(T)
Value	Value	Value
0	1	1
1	0	1
1	0	1
0		
0		
0		

(d) Bitmaps

Figure 4.2: A tiny example of stable data, m- and t-delta and their bitMaps. (a) Stable data stored in the cloud storage engine. (b) M-delta with updates for stable data and timestamp 50 stored in M-Delta cache. (c) T-delta with updates for stable data and m-delta and timestamp 100, just arrived in cache. (d) BitMaps for stable data, m- and t-delta generated with the FGM algorithm.

DEL. A timestamp (TS) value is associated with every m- and t-delta showing their arrival time. A greater timestamp value means that the delta is more recent. Stable data, by definition, have a TS=0. After the execution of the FGM process in this example, the algorithm keeps all the entries (PK=998, 1002, 1004) from the t-delta which are the most recent versions. For the m-delta, the FGM algorithm keeps only the first entry (PK=1001). The entry with PK=1002 is overwritten by the t-delta update and the entry with PK=1003 is deleted. Similarly, for the stable data, the entries with PK=998 and 1001–1003 are overwritten by the m- and t-delta, and the algorithm keeps only the entries with PK=999 and 1000.

A naive design of FGM algorithm would use a merge algorithm similar to the sort-merge join. In this case, the algorithm compares the smallest primary keys from each data source (stable data, m-deltas, and t-delta), outputs one row at a time and moves to the next comparison. We found

this naive design to be too slow and not able to catch up with the speed of analytical processing. To optimize this naive design, we leverage vector processing (e.g., SIMD and code generation in Gandiva [4]) to speedup the merge process. To enable this optimization, we introduce the *bitMap vector* data structure.

BitMap Vector. The bitMap is a vector of binary bits (0 or 1). We extract one bitMap for each data source. Each bitMap has size equal to the corresponding data source. The entries in the bitMap with value 1 correspond to the data source entries that will be retained, while the entries in the bitmap with value 0 correspond to the data source entries that have been overshadowed by a later change and will not be kept. Figure 4.2d shows the bitMaps which correspond to the data sources of our tiny example (see Figure 4.2). The BM(S) bitMap corresponds to the stable data, the BM(M) bitmap to the m-delta and the BM(T) to the t-delta.

FGM phases. Foreground merging (FGM) consists of two phases: the *bitMap generation* phase and the *filtering* phase. The bitMap generation phase uses the *k-way merge* algorithm to calculate k bitMaps, one for each data source that participates in the merging. The filtering phase uses the k bitMaps generated in phase one and filters out unwanted entries from each data source.

We perform a separate FGM for each partition of each table that participates in the analytical query given that the table has undergone changes. In a typical scenario, the different data sources that participate in the merging are the stable data and the deltas. Note that for a specific partition and table, there may exist multiple m-deltas stored in M-Delta cache. Each m-delta corresponds to a different point in time depicted by its timestamp. The newest delta is always the last delta that comes from the log (t-delta). Therefore, the stable data, the t-delta and $k-2$ m-deltas, with $k \geq 2$, will be the input to the FGM algorithm.

Our FGM algorithm uses the k-way merge algorithm to extract the bitMap for each of the k data inputs. We follow a very simple rule; the changes in the t-delta are the latest, based on the timestamp, and should overwrite the stable data or/and the $k-2$ m-deltas. Then, for the rest $k-2$ m-deltas, assuming always that the $k-2$ m-delta is more recent than the $k-3$ m-delta, the changes in the $k-2$ m-delta will override the stable data and the rest $k-3$ m-deltas. More specifically, we initiate k pointers which keep track of the current position in each data input. Each time, we compare the primary keys of the first element from each data input and we extract the minimum element(s). Note that the same primary key can appear in multiple data inputs and

we use the timestamp rule we explained earlier to update the bitMaps. If the minimum primary key occurs in multiple data inputs, we determine the data input with the most recent timestamp among them. For this data input we set to one the entry of the bitMap corresponding to the position of the minimum primary key. For the rest data inputs (less recent timestamps) in which the minimum primary key occurs, we set to zero the entry of the bitMap corresponding to the position of the minimum primary key. Finally, we increment the pointer of each data input from which the smallest primary keys were extracted and we continue with the next comparison until we see all the entries of each data input and all the bitMap vectors are populated. The asymptotic complexity of the algorithm is O(−stable− + $\sum$−m-delta$_n$− + −t-delta−), where −x− represents the size of each data input and $n \in [1, k-2]$.

After the generation of bitMaps, FGM is ready to move to the second phase, the filtering. In the filtering phase, the entries that are set to 0 in the bitMaps of phase one are filtered out from the corresponding data. To make the filtering phase efficient, we use the Gandiva [4] expression compiler. Gandiva uses LLVM to generate efficient native code for filtering, and it is designed to take advantage of the Arrow memory format and modern hardware (e.g., SIMD instructions). After the filtering, the remaining records from each data source are the full snapshot of the data as of the time the analytical query arrived for execution. These records are combined and sent for processing to the next operator of the query execution.

Note that the FGM does not apply the changes of the deltas to the stable data by creating a new version of the stable data and deleting the old one (e.g., copy on write). FGM generates an up-to-date snapshot of the data for the current query without reusing the result of the merge. The m-detlas and the stable data remain unchanged after the execution of the FGM. When a new query arrives for execution, a new FGM process (for each data partition) will be initiated to create a new snapshot of the data including the latest changes from DeltaPump. Our method is similar to the multi-versioning idea. As time passes, more and more deltas for a specific partition and table will accumulate in memory and each will have a timestamp. To answer a query, the correct version of the data needs to be generated based on the arrival time of the query.

M-Delta Merge Optimization. The first phase of the FGM algorithm uses k-way merge to generate k bitMaps of each data input $\in k$. As time passes, more m-deltas with different timestamps will be gathered in the M-Delta cache for the same partition of the data. However, the time

complexity of the k-way merge algorithm increases with the increase of k. To alleviate this problem, we introduce the *m-delta merge optimization* that enforces the *one m-delta rule* in cache.

We change the original FGM algorithm to always return the merged version of the input t-deltas and m-deltas. The merged version of the deltas is saved in the M-Delta cache, replacing the input deltas. Using this method, there will always be one m-delta in the M-Delta cache, and three data inputs will participate in the k-way merge: the stable data, one m-delta, and the new t-delta.

To achieve the one delta rule, after executing the FGM, we replace the current m- and t-deltas in cache with the combined version of the remaining entries (after filtering) of the m- and t-deltas.
BitMap Caching Optimization The time complexity of the bitMap generation phase is negatively affected by the number of data rows in the three data sources (e.g., stable data, m- and t-delta). We observe that for a particular data source, the bitmap vector changes only slightly from query to query, and the changes are monotonic — a bit can change only from 1 to 0, because it is overwritten by the entries of data sources with more recent timestamps, but not the other way around. To accelerate the creation of bitMaps we propose to cache the bitMaps of stable data in memory and avoid the cost of constant regeneration. These bitMaps are small in size and can be cached with small memory footprint. For large cold data partitions where the bitMaps are not cached, they can be regenerated on demand.

Since the size of the t-deltas is almost constant, the time complexity of the bitMap update is also constant. The limitation of bitMap caching optimization is the extra space required for caching the stable data bitMaps. Note that every bitMap needs to store only one bit for each entry of the stable data. The total number of bitMaps is equal to the number of table partitions. Therefore, the total size of the bitMaps is expected to be relatively small and not prohibitive for caching. In the case that bitMaps cannot be cached, we can fall back to recalculating the bitMaps on the fly for each FGM.

Background Merge (BGM)

As time passes and more queries are executed, the size of each delta in the m-delta cache will grow. As a result, the size of the cache occupied by the deltas will increase. To mitigate this problem, we design a background process that periodically moves the m-deltas to the cloud storage

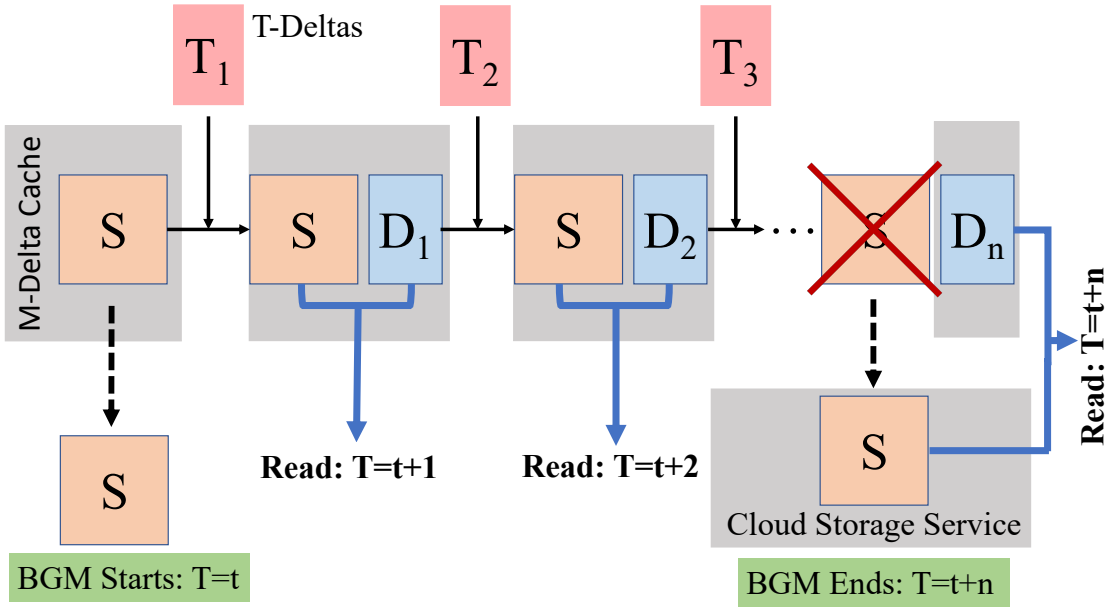


Figure 4.3: M-Delta Cache organization during BGM.

service. In this section, we introduce our *Background Merge* (BGM) process and the *D-Deltas Merge* optimization.

BGM Algorithm. The BGM process is triggered once the size of the m-delta cache exceeds the threshold T_{cache}. Then, each m-delta whose size exceeds a threshold T_{delta} is evicted to the storage engine. To avoid blocking the execution of upcoming queries, a dedicated thread is responsible for evicting the current m-deltas from cache and copy them to the cloud storage. This thread is different from the thread that handles the m-delta and bitMap cache as described in Section 4.2.3. Depending on the size of the data that have to be uploaded in the cloud storage, the BGM process might take longer time to complete. To achieve time efficiency, we introduce a new m-delta cache mode.

Figure 4.3 shows the high level idea of the new cache mode during the execution of the BGM. In this example, we assume that S is the snapshot of the m-delta cache at the time $T = t$. At the time $T = t$ BGM is initiated for the snapshot S of the m-delta cache. This means that all the m-deltas included in the S snapshot have to be uploaded in the cloud storage and then deleted from the cache. While the dedicated thread of the BGM is working on uploading the data, the deltas of the S snapshot continue to lay in cache until they become visible in cloud storage and is safe to be deleted from cache. To make the deletion of the snapshot S faster, when the BGM is

completed, we do not merge the incoming t-deltas with the snapshot *S* as described in Section 4.2.3. We leave the snapshot *S* untouched while merging the new t-deltas separately. For example, in Figure 4.3 after the arrival of the T_1 t-deltas (one for each table partition), a new snapshot D_1 is created in cache which corresponds to time $T = t + 1$. Upon the arrival of the next query, the T_2 t-deltas will arrive in cache and will be merged with the previous D_1 snapshot. The new merged snapshot D_2 corresponds to the time $T = t + 2$. Note that a query that arrives at time $T = t + 2$ has to read deltas from both the *S* and the D_2 data snapshots for the query result to be correct. The two snapshots will co-exist in the m-delta cache until the data of *S* is uploaded in the cloud storage and is visible to the next queries. The figure shows that at time $T = t + n$ the deltas of *S* are saved on the cloud storage. At that time, the deltas of *S* are deleted from cache while the deltas of D_n remain and the BGM is terminated. A query that arrives at time $T = t + n$ needs to read both the deltas of D_n and the deltas of *S* from the cloud storage.

Each delta of the *S* snapshot is uploaded on the cloud storage as a new CSV or Parquet file with the timestamp of the delta attached in the name of the file. We call these files disk-deltas (d-deltas). The d-deltas will be included in the future FGM processes, and they will be treated similarly to an m-delta with an older timestamp. More specifically, a separate bitMap will be generated for the d-delta of a partition that will be cached and updated by the new t-delta entries.

D-Deltas Merge Optimization. The BGM process can be triggered multiple times creating many d-deltas with different timestamps for the same table partition. This affects negatively the execution time of the FGM process. To alleviate the problem, we introduce the *d-deltas merge* optimization.

A dedicated thread in the background merges the d-deltas of the same partition into one d-delta. Since we already have in cache the bitMaps of each d-delta, we use them to generate the merged version of all the current d-deltas using the filtering phase of FGM. Once the merged version is extracted it is saved in the cloud storage as a new CSV or Parquet file with the most recent timestamp added in the file's name. Once the merged version is visible and no running queries are using the unmerged d-deltas they can be safely deleted.

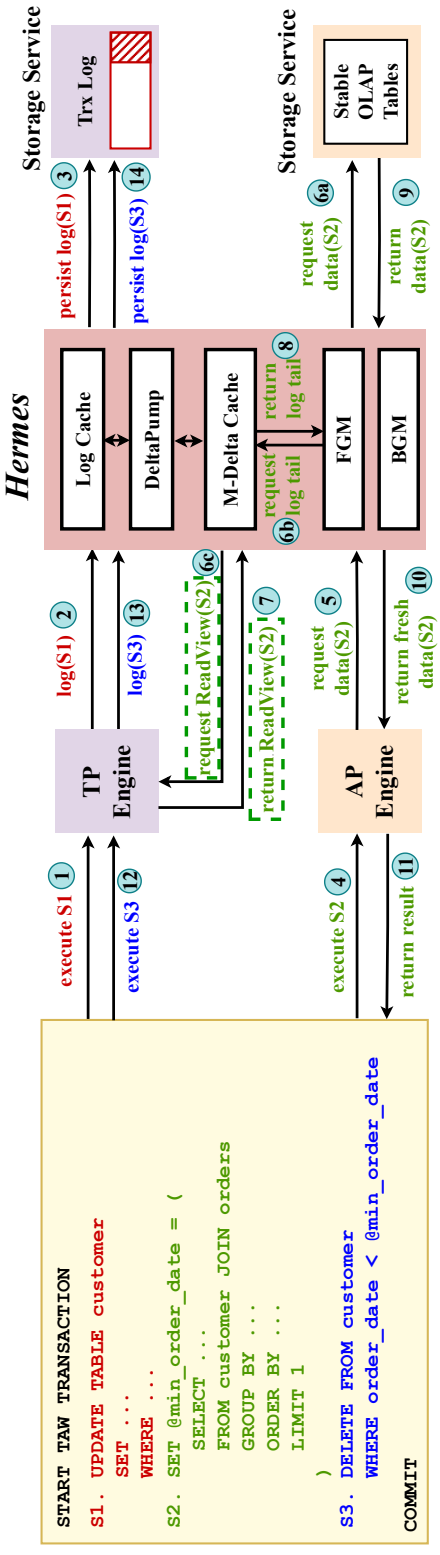


Figure 4.4: Execution flow of an analytical transaction in HERMES with Snapshot Isolation (SI). Note that the workflow remains unchanged whether HERMES is present or not. An additional coordination exists between the TP engine and HERMES, enabling HERMES to receive the list of visible transactions necessary for achieving SI. This coordination integrates seamlessly.

4.3 Transactional Analytics with HERMES

HERMES not only achieves off-the-shelf real-time analytics but also enables real-time transactional analytics.

4.3.1 Design Challenges

We identify the following two key challenges when integrating transactional analytics into off-the-shelf real-time analytics system.

- **Efficient Engine Selection.** To optimize the execution of transactional analytics within an off-the-shelf real-time analytics system, it is essential to process transactional statements in the TP engine and analytical statements in the AP engine. This approach ensures that each workload type is executed in its respective specialized engine, thus maximizing efficiency and performance.
- **Isolation Level Consistency.** Efficient engine selection mandates that analytical statements within an analytical transaction executes in the AP engine. Maintaining consistent isolation levels for both analytical and transactional statements across different engines is crucial [77, 150]. This synchronization responsibility falls to the TP engine, ensuring that the AP engine accesses accurate data based on the selected isolation level. Ideally, transactional analytics integration should align with TP/AP engines' inherent characteristics, avoiding internal logic modifications.

4.3.2 HERMES' Isolation Levels Solutions

This section introduces HERMES' generalized solutions for transactional analytics under Snapshot Isolation, Serializable, and Read Committed, as well as MySQL-specific implementations.

Snapshot Isolation

Snapshot Isolation (SI) [32] ensures that each transaction sees a consistent snapshot of the database as it existed at a specific point in time, typically at the start of the transaction. In database systems,

SI is typically implemented using Multi-Version Concurrency Control (MVCC) [33]. In this design, a transaction accessing a table with SI must determine the visible snapshot and read only the data within that snapshot. Different systems represent the snapshot in different ways; some use a single timestamp [34, 50, 90] and others use a compact representation of a list of transaction IDs [106, 114]. We assume the list of transaction IDs in the following discussion but the solution apply to both scenarios.

The list of visible transaction IDs determines which versions are included in a transaction's snapshot. At the start of a transaction's execution, the TP engine gathers this list and shares it with HERMES. By the time the first analytical query, within the analytical transaction, is about to execute in the AP engine, HERMES will have the list and can use it to retrieve the correct log events.

This solution applies to any TP engine supporting SI with MVCC and a row-level transaction log, needing only minor code changes to transmit the transaction ID list to HERMES.

Workflow Example. Figure 4.4 illustrates the execution flow of an analytical transaction with HERMES under snapshot isolation, comprising three statements: (S1) an update, (S2) an analytical query, and (S3) a delete. The transaction begins with S1 executed in the TP engine (step 1), generating a log saved in HERMES's Log Cache (step 2) and persisted to the Storage Service (step 3). Next, S2 is sent to the AP engine (step 4), which requests data from HERMES (step 5). HERMES retrieves stable data from the Storage Service (step 6a), fetches the log tail (step 6b), and obtains the ReadView from the TP engine for snapshot consistency (step 6c). After preparing the fresh data (steps 7-9), HERMES delivers it to the AP engine (step 10). The query result is returned to the client (step 11) and used in S3, executed in the TP engine (step 12) and persisted via HERMES (steps 13-14). With S3 complete, the transaction is ready to commit.

Implementation Details for MySQL. InnoDB, MySQL's default storage engine, uses Undo Logs as part of its MVCC implementation. Each transaction in InnoDB has a set of undo log records, enabling access to previous record versions. Every transaction is assigned a unique ID, and when a consistent read is needed, InnoDB creates a snapshot, or read view, that includes: (1) IDs of active transactions, (2) a lower bound of committed transaction IDs, and (3) an upper bound of future transaction IDs. InnoDB uses this snapshot to access appropriate data versions from the undo log, ignoring records with transaction IDs above the upper bound and those between the bounds if

Algorithm 1: HERMES API for enabling Transactional Analytics with Snapshot Isolation in MySQL, with changes to the InnoDB storage engine highlighted in gray.

```
1  Function InnoDB::CreateSnapshot(request)
2      readView ← {active_trxs, l_bound, u_bound}
3      readViewMap ← {}
4      for each trxID in readView do
5          trxLogID ← MySQL::getTrxLogID(trxID)
6          readViewMap += {trxID, trxLogID}
7      return readViewMap
8  Function HERMES::ReadFromLog(readViewMap)
9      data ← {}
10     for each trxLogID in LogTail do
11         if trxLogID in readViewMap.active_trxs then
12             continue
13         else if trxLogID > readViewMap.u_bound then
14             continue
15         data += {after image of trxLogID entry}
16     return data
17 Function HERMES::EnableTAW(request)
18     readViewMap ← InnoDB::CreateSnapshot(request)
19     data ← HERMES::ReadFromLog(readViewMap)
20     return data
```

active. This process allows InnoDB to achieve a snapshot for reads, aligning more closely with SI than the Repeatable Read isolation level claimed by MySQL.

As shown in algorithm 1, HERMES requires two pieces of information for transactional analytics with SI in MySQL: (1) the read view from the InnoDB engine (line 2) and (2) a mapping between InnoDB transaction IDs and those in the transaction log (lines 3-7). Note that MySQL's log transaction IDs differ from InnoDB's transaction IDs. This mapping, along with the ReadView, allows HERMES to retrieve the correct data snapshot for analytical queries (lines 8-16). To provide this data, we made minor modifications to MySQL (lines 3-7), extending the InnoDB's read view with MySQL log transaction IDs and adding code in InnoDB to send the updated read view to HERMES via Thrift clients. Overall, the modifications and additions made to MySQL are fewer than 100 lines of code.

Serializable

Serializable (SR) [32] isolation level is the strictest isolation level, ensuring transactions execute in a manner equivalent to a serialized order of execution. A conventional method to achieve SR isolation is through a variant of Two-Phase Locking (2PL) [34]. By using 2PL, the database system maintains read and write locks, guaranteeing conflicting transactions execute in a defined sequence, resulting in serializable execution schedules.

To achieve SR transactional analytics in HERMES, the TP engine should prevent concurrent modifications during the execution of the analytical queries within the analytical transaction. When a table needs to be read on the AP side, the entire table should be locked on the TP engine for the duration of the transaction.

This solution can be applied using any TP engine that offers SR using 2PL and supports granularity locking. Implementing it necessitates adjustments to the locking logic within the TP engine—hold locks even if the data is not accessed in the TP engine.

Workflow Example. In the SR isolation level, the workflow described in Figure 4.4 differs slightly in terms of communication between the TP engine and HERMES. Specifically, compared to steps 6c and 7 of the SI level, the SR level requires somewhat enhanced coordination between the two servers. However, the rest of the workflow remains unchanged.

Implementation Details for MySQL. MySQL's InnoDB storage engine ensures serializability

through the implementation of 2PL and granular locking mechanisms. At the SR isolation level, when a read operation is performed within a transaction, InnoDB employs granular locks—such as row, range, or next-key locks—on the necessary data. These locks are acquired at the start of the transaction and held until its completion, effectively preventing other transactions from writing to the locked data.

In our system architecture, when Hermes receives a request to scan data for a specific query, it communicates with MySQL to acquire exclusive locks on the relevant tables, utilizing the InnoDB API to manage these locks at the table level. Once MySQL secures the exclusive locks, it notifies Hermes to proceed with the data scanning operation. This locking mechanism ensures that during Hermes's scanning process, the tables remain isolated from concurrent transactions, preventing any updates that could compromise data integrity. The communication between MySQL and Hermes is facilitated through RPC (e.g. Apache Thrift). Implementing these changes in MySQL required adding fewer than 150 lines of code.

Read Committed

In Read Committed (RC) [32] isolation, transactions view only committed data stored in the transaction log. HERMES achieves RC for transactional analytics by having analytical queries read all committed transactions of the relevant table from the log. This approach applies to any TP engine with row-level transaction logging.

Workflow Example. For RC isolation level, the workflow described in Figure 4.4 remains unchanged except that steps 6c and 7 are omitted. HERMES achieves RC by reading each committed transaction directly from the log, without TP engine coordination.

Implementation Details for MySQL. In HERMES, achieving RC isolation level with MySQL required no code changes or additions.

4.3.3 Transactional Analytics Workload (TAW)

This section explores the significance of *Transactional Analytics*, emphasizing on how existing benchmarks lack generalization compared to TAW and explains the TAW's design principles and details.

Motivation of Transactional Analytics

Transactional analytics provide real-time insights in HTAP systems by integrating transactional and analytical operations within a single workflow. Standard HTAP benchmarks typically assign separate clients for transactional and analytical workloads, addressing only strictly separated request types. However, prior research [109, 149, 73] shows that hybrid workloads often require mixed transactional and analytical operations within the same transaction. Full HTAP support enables analytical queries on fresh data both post-commit and within the same transaction, allowing subsequent actions based on query results in real-time. This integrated approach supports consistent isolation, avoiding partial updates or stale reads in workflows needing instant decisions on the latest data.

Transactional analytics are critical for applications like fraud detection, personalized healthcare, and supply chain optimization. HyBench [149] uses them for risk control, triggering actions like transaction rollbacks, while PocketData [73] focuses on data management, leveraging nested sub-queries for data deletion. Despite their benefits, existing benchmarks lack a generalized framework for diverse transactional analytics scenarios.

Generalized Transactional Analytics with TAW

TAW evaluates HTAP systems under generalized transactional analytics scenarios and models diverse transactional analytics patterns, rather than being restricted to a single, predefined workflow such as HyBench [149]. Specifically, the synthetic nature of TAW provides fine-grained control over access patterns—allowing updates, insertions, or deletions to occur before, after, or between analytical queries within the same transaction.

Our experiments show that varying the sequence of operations impacts query plans in HTAP systems (see Section 4.5.4), emphasizing TAW's ability to test broader scenarios. In general, TAW highlights limitations in current approaches and advocates for more adaptable benchmarks to address diverse TA patterns.

TAW Design Details

To create TAW, we build on HATtrick [98], an HTAP benchmark. HATtrick combines an adapted version of TPC-C [15] for transactional tasks and the Star-Schema Benchmark (SSB)[110] for analytical queries. Its transactional workload (issued by transactional clients T-clients) includes three types of transactions—*NewOrder*, *Payment*, and *CountOrders*—where *NewOrder* is an insertion transaction, *Payment* involves updates and insertions, and *CountOrders* is read-only. The analytical workload (issued by analytical clients A-clients) consists of 13 SSB[110] queries.

For TAW, we take the transactional component of HATtrick and, within each transaction (*NewOrder*, *Payment*, and *CountOrders*), append one of the 13 SSB [110] analytical queries either after or interleaved within the original workload. By adding SSB queries to each transaction, TAW integrates analytics within the same transactional request [109]. A random SSB query number is selected for each transaction to ensure equal probability across all queries. These adapted requests form one part of the TAW workload, issued by dedicated transactional analytics clients (TA-clients), while the original HATtrick transactional requests are issued by T-clients.

4.4 HERMES Potential Extensions

This section explores potential enhancements to the HERMES design, which are considered for future work.

4.4.1 Cache Offloading to HERMES

Integrating the AP engine with HERMES prevents it from using its local cache, necessitating cache offloading. However, this introduces challenges. First, remote cache access incurs network latency. Second, HERMES must align with the AP engine's caching mechanisms. Third, the optimizer may struggle to generate efficient plans without direct cache metadata. Finally, offloading may disrupt index-based query optimizations.

To mitigate latency, co-locating HERMES with the AP engine minimizes network overhead, enabling near-native caching. HERMES bridges remote caching with the AP engine's optimizer by sharing metadata—index structures, materialized views, and statistics—facilitating efficient

execution plans. To maintain consistency, HERMES periodically synchronizes metadata with the AP engine, ensuring that query execution reflects the latest cache state. Moreover, HERMES supports indexing techniques such as min-max indexing and range partitioning, dynamically adjusting index boundaries to enhance accuracy and efficiency.

4.4.2 HERMES in a Distributed Setup

In distributed TP/AP environments, HERMES enhances scalability and resource efficiency, enabling real-time analytics without disrupting TP/AP engine functionality.

HERMES with Distributed TP. Distributed TP systems typically follow two architectures: (1) a single primary node with multiple read-only replicas [88] and (2) a partitioned shared-nothing model [135]. We describe how HERMES operates in each.

- Primary and Replica. Here, a primary node manages writes while read-only replicas handle queries. The TP engine ensures consistency via replication consensus algorithms. With the HERMES integration the storage layer remains the single source of truth. HERMES guarantees fresh data delivery to the AP engine by verifying transaction log durability before merging logs with stable data.

- Partitioned Shared-Nothing. In this architecture, data is partitioned across independent nodes, requiring a global transaction order to maintain consistency. Existing protocols, such as two-phase commit [34] and timestamp-based mechanisms in Multi-Version Concurrency Control, ensure a consistent transaction sequence. HERMES uses these mechanisms to merge log entries with the correct data partitions while preserving system integrity.

HERMES with Distributed AP. In both distributed and non-distributed AP engines, the interface between an AP node and the storage engine (e.g., S3) remains consistent and HERMES integrates seamlessly without architectural modifications. To meet distributed AP engines' I/O demands, HERMES scales by partitioning data across multiple servers, each responsible for specific table partitions. A consistent partitioning strategy ensures logs are routed correctly without altering HERMES' internal design.

4.4.3 HERMES Advancing Middle Layers

HERMES acts as an intermediary between database servers and storage services, enhancing cloud database performance. Traditional middle layers optimize transactions [25], accelerate filtering and aggregation [10, 147], and support caching for query optimization [62]. HERMES enhances these capabilities by supporting real-time and transactional analytics while preserving compatibility with existing architectures. It integrates seamlessly by first applying the latest TP engine updates, then performing pushdown computations to process relevant data before returning results to the AP engine.

4.5 Experimental Evaluation

This section evaluates off-the-shelf real-time analytics against baseline systems, highlighting three key aspects of HERMES.

- HERMES can seamlessly integrate with existing TP/AP engines without introducing additional overhead (Section 4.5.2).
- HERMES' overall TP/AP performance is competitive to well-established solutions (Section 4.5.3).
- HERMES offers superior performance for transactional analytics compared to existing solutions (Section 4.5.4).

4.5.1 Experimental Setup

Cloud Server Configuration

The experiments are conducted on compute-optimized AWS EC2 instances in the US-West-2 region. We use three different instance types: (1) c5.4xlarge ($0.68 per hour) with 16 vCPU, 32 GB memory and 10-Gbps network bandwidth, (2) c5.9xlarge ($1.53 per hour) with 36 vCPU, 72 GB memory and 10Gbps network bandwidth, and (3) c5.12xlarge ($2.14 per hour) with 48 vCPU, 96 GB memory, and 12Gbps network bandwidth.

Systems Setup Configuration

In this section we present the different systems setups for HERMES, MySQL, and TiDB.

HERMES Setup. We use three instances, one for each component: (1) the HERMES server uses a c5.9xlarge, (2) the AP-engine (FPDB or DuckDB) uses a c5.4xlarge, and (3) MySQL uses a c5.4xlarge. DeltaPump uses the MySQL binary log connector [129] to parse the log. Both the TP and AP engines maintain a copy of the database with the same schema. The AP-engine's copy is stored in Amazon S3 in Parquet [7] format, while the TP-engine's copy is stored on disk; the TP-engine logs to AWS EBS, through HERMES.

- Storage Cost. In our setup, MySQL stores one copy of the data on AWS EBS, while DuckDB stores another copy in AWS S3. The cost of a General Purpose SSD (gp3) is $\$0.08$ per GB-month, and MySQL uses 55GB, resulting in a monthly cost of $\$4.40$. The cost of S3 Standard storage is $\$0.023$ per GB for the first 50 TB per month, and our data size in S3 is 10GB, leading to a monthly cost of $\$0.23$. Therefore, the total storage cost for HERMES is $\$4.63$ per month.

  Note that, the AP data stored in the storage engine occupies 10GB in Parquet, whereas the same data requires 55GB in MySQL due to additional storage overhead. This 55GB consists of the base data, InnoDB metadata, and default TP indexes, which increase storage consumption. In contrast, Parquet's compressed columnar format optimizes storage for analytics, resulting in a smaller footprint.

- Memory consumption. In our HERMES setup, the AP engine cache is not offloaded to HERMES. DuckDB, by default, avoids caching when reading from a storage engine, and FPDB's caching is disabled. HERMES allocates memory for three caching mechanisms: the Log Cache, M-Delta Cache, and bitmaps for FGM and BGM, as detailed in Sections 4.2.3 and 4.2.3, to accelerate merging processes.

MySQL Setup. We setup MySQL with InnoDB storage engine in a c5.12xlarge instance. Moreover, we created B+ tree indexes to optimize the analytical workload and fine-tuned several MySQL parameters to ensure optimal performance.

- Storage Cost. MySQL baseline uses AWS EBS for database storage. Based on actual usage, MySQL stores 115GB of data, resulting in an estimated storage cost of $9.20 per month. This storage consists of two main components: (1) TP data ($\sim$ 55GB), which includes InnoDB metadata and default TP indexes, and (2) AP indexes ($\sim$ 60GB), which improve query performance but significantly increase storage overhead.

TiDB Setup. We deploy twelve c5.4xlarge instances following TiDB's recommended configuration [139, 11]: two TiDB servers, six TiKV servers (three replicas per region), and four TiFlash servers. Utilizing TiFlash's disaggregated storage and compute architecture [11], we allocate two write nodes and two compute nodes for TiFlash. The write nodes handle logs from TiKV, convert them to columnar format, and periodically upload updated data to cloud storage. The compute nodes execute queries, accessing the latest data from the write nodes and remaining data from cloud storage.

- Storage Cost. The TiDB setup utilizes six TiKV nodes with three replicas using a space of $\sim$ 117GB in total. TiFlash nodes store data in AWS EBS ($\sim$ 7GB) and in AWS S3 ($\sim$ 8GB). This results in a total storage cost of $\$10.10$ per month.

Baseline Selection. The selection of these systems is driven by their established strengths in their respective domains. MySQL, with its proven transactional processing capabilities and widespread use in cloud environments [141, 29, 9], serves as a key baseline for transactional workloads. DuckDB is a high-performance analytical engine, aligning with trends in data lakes and cloud-native analytics, essential to our architecture. Additionally, FPDB is included to evaluate HERMES' adaptability with less conventional AP engines. FPDB demonstrates HERMES ' flexibility in integrating with a diverse range of engines providing valuable insights into the system's versatility. Finally, TiDB was selected as the state-of-the-art (SOTA) HTAP system due to its increasing adoption by major companies [140], which leverage TiDB's ability to manage large-scale transactional and analytical workloads concurrently in real-time. TiDB's robust support for hybrid workloads makes it an ideal baseline for evaluating HERMES ' performance in HTAP.

Workloads

We use two workloads for evaluation: the HATtrick benchmark [98] and the *Transactional Analytics Workload (TAW)*, an adapted version of HATtrick. We discuss their characteristics in Section 4.3.3. To test HERMES under more demanding conditions, we modify HATtrick and TAW to simulate different update/insertion patterns, ensuring an update to every partition of the schema tables with a probability of one.

Metrics

HATtrick extracts the following metrics, a *throughput frontier graph* and a *freshness score* for every system under test.

The throughput frontier graph is a 2D plot with transactional throughput (T-Throughput) and analytical throughput (A-Throughput) on the x- and y-axis. It is generated by running various client mixes, showing the system's performance across the HTAP spectrum and its isolation capabilities. Ideally, the frontier aligns with the *bounding box*, defined by maximum T-Throughput and A-Throughput values, indicating perfect isolation. A frontier close to or below the *proportional line* suggests poor HTAP performance. The average freshness score is measured in seconds. A freshness score of $f_{avg} = 0$ indicates the database always provides the most recent operational data to analytical queries.

Similar to HATtrick, TAW generates a throughput frontier for each database, reflecting comparable insights. Note that, in TAW, the y-axis represents transactional analytics throughput (TA-Throughput), measured in analytical transactions per second (taps). In the experiments with HATtrick and TAW, we use scale factor 50 databases, resulting in data sizes of approximately 10GB in Parquet format.

Measurement Methodology

To extract one of the throughput frontiers of Section 4.5.3, we execute multiple experiments with different ratios of T- and A-clients. More specifically, we keep a single A-client and vary the number of T-clients from zero to number of clients that maximizes the T-Throughput in each database and scale factor, we call it M_{max}. Each experiment includes a warm-up phase followed

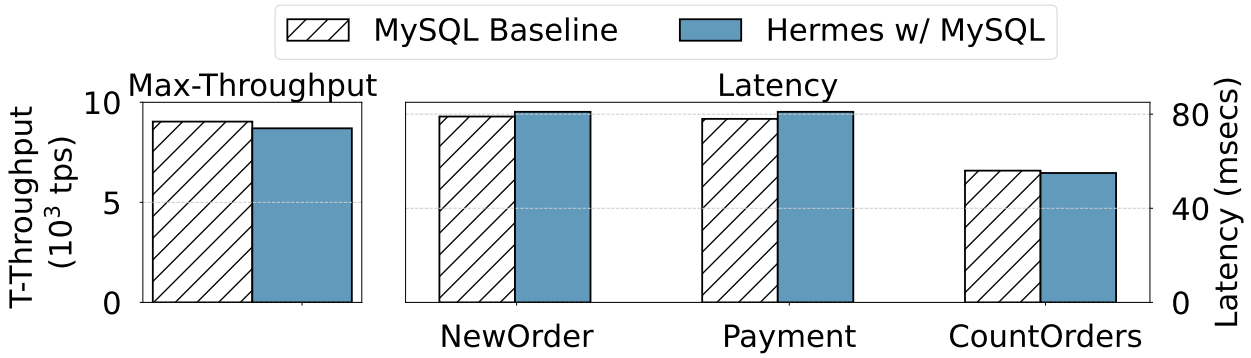


Figure 4.5: Transactional throughput (T-Throughput) results in tps (left) and transactions' latency results in msecs (right) when executing HATtrick in HERMES w/ MySQL vs. the standalone MySQL executing transactions.

by a measurement phase. For the results of Section 4.5.2 we execute a single experiment in which we fix the number of T-clients to M_{max} and the number of A-clients to one and we extract the latency results.

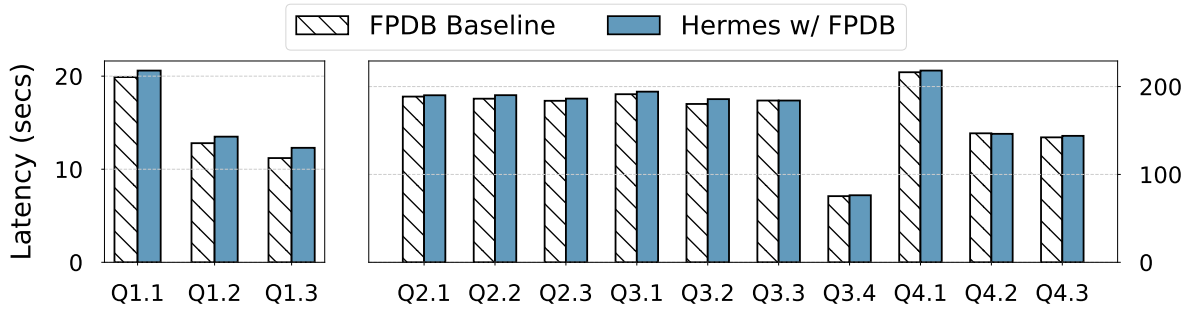
The TAW clients operate similarly to HATtrick. However while HATtrick features an A-client, TAW features a TA-client that issues analytical transactions to extract the throughput frontier results.

4.5.2 HERMES Evaluation

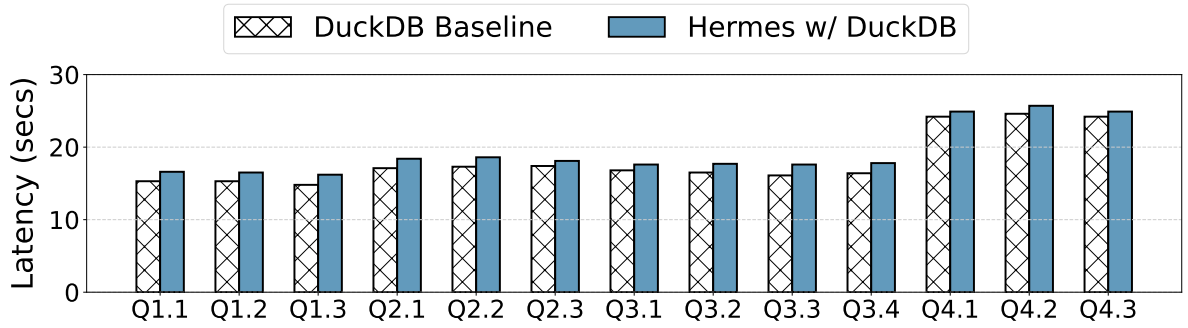
This section presents the end-to-end results of HERMES integration with MySQL [106], FPDB [146], and DuckDB [117], demonstrating that integration does not impact their original performance. It also provides results on HERMES resource utilization.

HERMES Integration with MySQL

Figure 4.5 (left) displays the maximum T-Throughput achieved in MySQL baseline and the HERMES with MySQL setup for the HATtrick benchmark. In MySQL baseline, only the transactional portion of HATtrick is executed, with analytical queries disabled. In contrast, for the HERMES setup, the results include the concurrent execution of analytical queries by the AP engine (FPDB or DuckDB). Both configurations use MySQL under Snapshot Isolation. Additionally, Figure 4.5 (right) shows the corresponding transaction latencies.



(a) FPDB vs. HERMES w/ FPDB.



(b) DuckDB vs. HERMES w/ DuckDB.

Figure 4.6: Latency (secs) of the HATtrick analytical queries in the HERMES & FPDB setup vs. the case where FPDB executes the queries without using HERMES.

MySQL baseline achieves a maximum T-Throughput of 9,035 tps, while HERMES with MySQL reaches 8,700 tps. Latencies are similar in both setups, with the HERMES integration introducing up to 4% overhead—a minor trade-off for added functionality.

In the next sections, we discuss the latency results of Figure 4.6, focusing on HERMES' performance with FPDB and DuckDB. These measurements were taken with HERMES connected to MySQL operating at a fixed T-Throughput of 8,700 tps, as shown in Figure 4.5. For all queries, the updates merged with stable data correspond to this throughput, remaining consistent throughout the experiments.

HERMES Integration with FPDB

Figure 4.6a illustrates the latency of the HATtrick queries when integrating HERMES with FPDB, compared to FPDB baseline. For each query the left bar corresponds to latency of the original query execution in FPDB baseline and the right bars correspond to the execution of the same

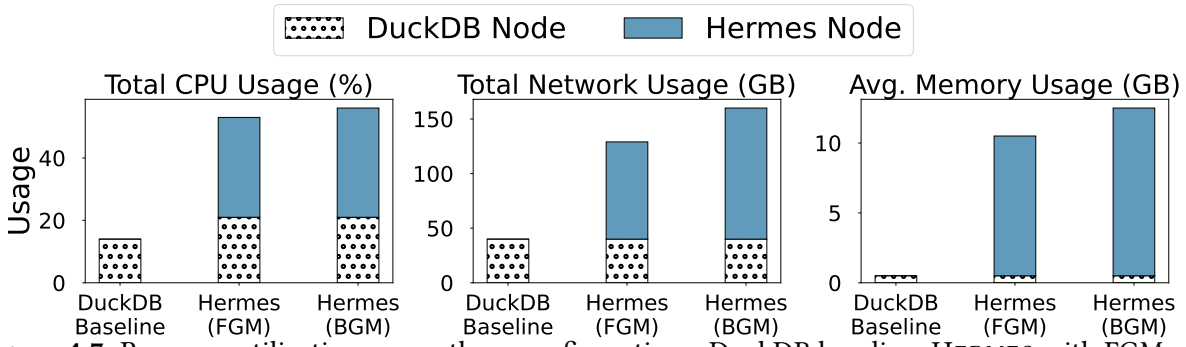


Figure 4.7: Resource utilization across three configurations: DuckDB baseline, HERMES with FGM, and HERMES with BGM. Figure shows Total CPU Usage across all vCPUs (%), Total Network Usage (GB) and Average Memory Usage (GB) for both the DuckDB and HERMES nodes.

query in HERMES with FPDB. In general, the closer the latency results of the HERMES with FPDB setup are to the original FPDB latency, the better for the overall performance of HERMES.

The results demonstrate an overhead of less than 4% in the latency across the 13 queries executed with HERMES, indicating that it imposes minimal performance impact on FPDB baseline.

HERMES Integration with DuckDB

Figure 4.6b shows the latency of HATtrick queries executed in HERMES with DuckDB, compared to the DuckDB baseline. For each query, the left bar represents the latency of the original DuckDB execution, while the right bar shows the latency in the HERMES with DuckDB setup. The results indicate that HERMES with DuckDB incurs only about a 2% latency increase compared to the baseline, demonstrating that even with a high-performance AP engine like DuckDB, HERMES can deliver real-time analytics with minimal impact on query latency.

Overall, the results in Sections 4.5.2, 4.5.2 and 4.5.2 demonstrate that the performance of MySQL, FPDB, and DuckDB remain stable after integration with HERMES. The next section shows the resource utilization of the HERMES with MySQL and DuckDB setup.

Resource Utilization

Figure 4.7 presents resource utilization for the experiment in Section 4.5.2, focusing on the integration of HERMES and DuckDB. It displays total CPU usage across all vCPUs (%), total network usage (GB) as the sum of received and sent data, and average memory usage (GB). For

the HERMES setup, utilization is split into the HERMES and DuckDB nodes, with separate bars for FGM and BGM. FGM shows usage during Foreground Merges only, while BGM includes both Foreground and Background Merges. The DuckDB baseline is included for comparison.

CPU Usage. Figure 4.7 shows that, compared to the DuckDB baseline, the DuckDB node's CPU usage increases from 14% to 21% in the HERMES setup, consistently across both FGM and BGM configurations. This rise is attributed to the deserialization process on the DuckDB node, which incurs additional overhead as data serialized for network transmission is reconstructed upon receipt.

In the HERMES node, CPU usage increases from 32% to 35% when BGM is enabled alongside FGM. This is expected, as BGM requires the HERMES node to handle an additional background task.

Network Usage. Figure 4.7 indicates that the DuckDB node's total network usage remains consistent between the DuckDB baseline and the HERMES setup. This is expected, as HERMES integration does not alter the volume of data DuckDB receives from storage.

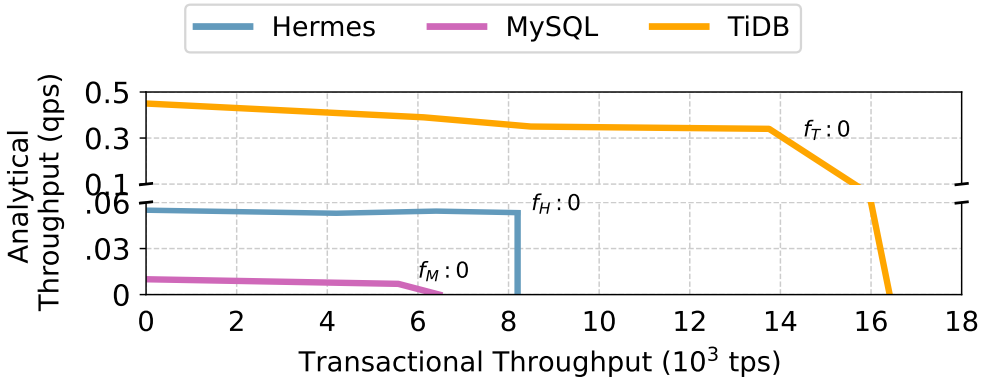
In the HERMES setup, the HERMES node's network usage exceeds the DuckDB node's due to receiving data from the storage engine and sending updates to DuckDB, effectively doubling usage. As expected, network usage rises further when BGM is active.

Memory Usage. Figure 4.7 shows that the DuckDB node's average memory usage remains consistent between the DuckDB baseline and the HERMES setup. This is expected, as integrating HERMES does not require additional data caching on the DuckDB node.

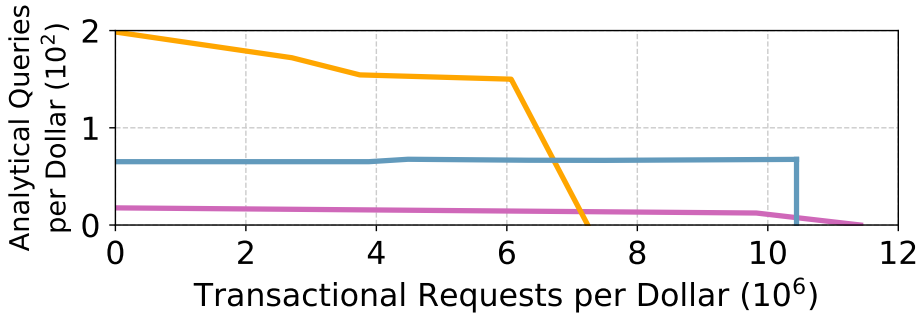
In the HERMES setup, the HERMES node utilizes memory mainly for components such as the Log Cache, M-Delta Cache, and bitmap caching, which are essential for accelerating FGM and BGM. The memory usage increases during BGM due to the additional process.

Integrating HERMES with DuckDB slightly increases DuckDB's CPU usage, mainly due to deserialization, while memory and network usage remain unchanged compared to the DuckDB baseline.

Estimated Cloud Cost. HERMES runs on one c5.9xlarge instance ($1.53/hour) and two c5.4xlarge instances ($0.68/hour each). Assuming continuous usage over 30 days, the total compute cost amounts to $2081 per month. For storage, HERMES uses a 64GB EBS volume for TP data in MySQL and a 10GB volume in S3 for AP data, resulting in a total storage cost of $5.53 per month. Thus,



(a) Throughput frontiers and avg. freshness scores.



(b) Operations per dollar frontiers.

Figure 4.8: Throughput frontiers, freshness scores (f_{DB}) and operations per dollar frontiers for HERMES (with MySQL and DuckDB), MySQL, and TiDB when executing HATtrick.

the combined compute and storage cost for HERMES is $2086.53 per month.

DuckDB, running on a single c5.4xlarge instance, incurs a compute cost of $490 per month under the same conditions. It stores 10GB of data in S3, contributing an additional $0.23 per month in storage costs. As a result, the total compute and storage cost for DuckDB is $490.23 per month.

4.5.3 HATtrick Evaluation Across Systems

This experiment compares HERMES, MySQL [106], and TiDB [66] using the HATtrick [98] benchmark, with MySQL and DuckDB as HERMES's TP- and AP-engines. The aim is to show that HERMES achieves performance comparable to established HTAP systems.

Comparison Results.

The discussion will focus on throughput frontier shapes, absolute throughput values, and cost frontiers.

Throughput Frontier Shapes. Figure 4.8a shows HATtrick results for each system. The HERMES frontier (blue) aligns closely with its bounding box, demonstrating excellent *performance isolation* and minimal TP and AP workload interference. MySQL's frontier (purple) falls between its bounding box and proportional line, indicating resource contention. TiDB's frontier (yellow) initially follows its proportional line, with A-Throughput decreasing as T-clients grow, but later approaches its bounding box, mitigating this effect.

Absolute Throughput. TiDB hast the highest T- and A-Throughput values in Figure 4.8a. This is expected, particularly for T-Throughput, as TiDB distributes transactional requests across two TiKV servers. The consistently high A-Throughput is due to TiDB's ability to cache frequently accessed data on the local SSDs of TiFlash compute nodes [11]. However, as T-clients increase, A-Throughput declines since frequent updates make cached data outdated.

Freshness Values. We used HATtrick benchmark to measure the freshness of the analytical queries in HERMES, MySQL, and TiDB. Our results show that all the three databases achieve zero freshness, indicating that all queries are always executed on up-to-date data.

Cost Frontiers. Note that the three curves in Figure 4.8a are generated using different hardware settings. For a more fair comparison, we normalize the monetary cost and report the throughput per dollar in Figure 4.8b. The figure highlights that HERMES can execute more transactional requests per dollar than TiDB. Conversely, TiDB outperforms HERMES in analytical queries per dollar, but as the number of T-clients increases, this difference becomes smaller.

Our results in Sections 4.5.2 and 4.5.3 show that HERMES inherits the stability of its underlying TP/AP engines. Specifically: (1) Figures 4.5 and 4.6 show that HERMES maintains the original performance of each engine, and (2) Figure 4.8 confirms that this stability holds across varying client combinations. The shape of HERMES' throughput frontier highlights its ability to deliver stable HTAP performance, allowing concurrent transactions and analytics without mutual impact. HERMES achieves this stability while matching leading HTAP systems in performance and offering a cost-effective solution.

4.5.4 TAW Evaluation Across Systems

In this section we assess the performance of HERMES, MySQL [106], and TiDB [66] when executing the TAW in three different isolation levels, Read Committed, Snapshot Isolation and Serializability.

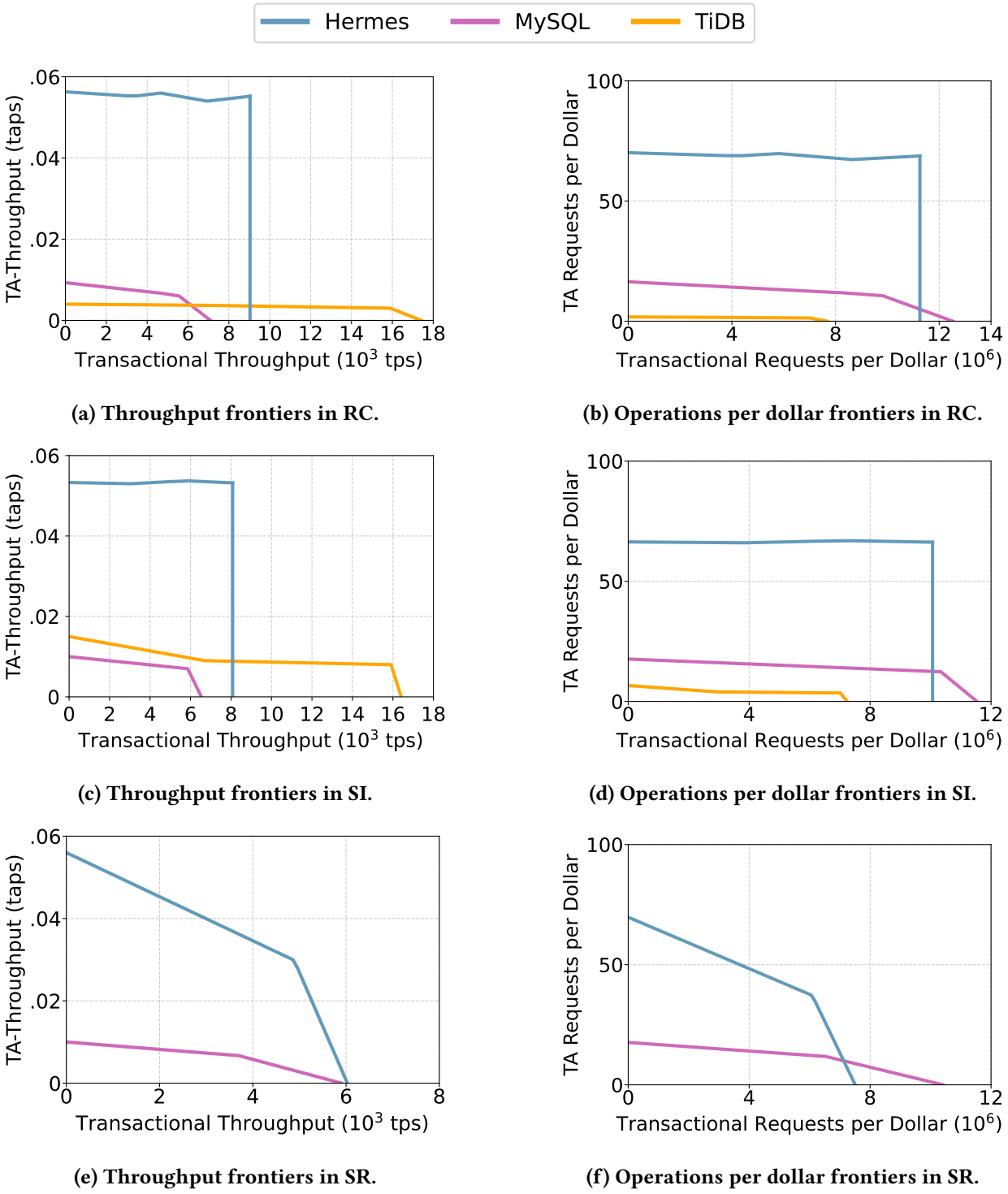
Comparison Results

Figure 4.9 presents the results of executing the TAW in HERMES (blue), MySQL (purple), and TiDB (yellow) across three different isolation levels. Each isolation level includes a graph depicting the throughput frontiers and another graph showing the corresponding operations per dollar frontiers.

Read Committed (RC) Results. Figure 4.9a illustrates the throughput frontiers, while Figure 4.9b presents the corresponding operations-per-dollar frontiers in RC. HERMES (blue) demonstrates nearly perfect performance isolation, as its throughput frontier closely aligns with its bounding box, indicating minimal impact from the transactional workload. In contrast, MySQL (purple) and TiDB (yellow) show frontiers between their proportional lines and bounding boxes, revealing a significant decline in TA-throughput as the number of T-clients increases. This decline is evident in the sharp drop towards the end of their frontiers.

In terms of absolute performance, HERMES achieves the highest TA-throughput (0.056 taps), followed by MySQL (0.009 taps) and TiDB (0.004 taps). TiDB leads in T-throughput with 18,000 tps, compared to HERMES (9,000 tps) and MySQL (7,000 tps). HERMES outperforms both competitors in transactional analytics per dollar (Figure 4.9b) and ranks second in transactional requests per dollar.

Snapshot Isolation (SI) Results. Similar to RC, Figure 4.9c presents throughput frontiers, and Figure 4.9d illustrates operations-per-dollar frontiers for SI. HERMES (blue) maintains strong performance isolation, with its frontier near the bounding box. In contrast, MySQL (purple) and TiDB (yellow) exhibit frontiers between their proportional lines and bounding boxes, reflecting a decline in TA-Throughput under higher transactional workloads.



(a) Throughput frontiers in RC.

(b) Operations per dollar frontiers in RC.

(c) Throughput frontiers in SI.

(d) Operations per dollar frontiers in SI.

(e) Throughput frontiers in SR.

(f) Operations per dollar frontiers in SR.

Figure 4.9: Throughput frontiers and operations per dollar frontiers results for HERMES, MySQL, and TiDB when executing TAW at scale factor 50 under three different isolation levels: Read-Committed (RC), Snapshot-Isolation (SI), and Serializable (SR).

In terms of absolute performance, HERMES once again achieves the highest TA-Throughput (0.054 taps), followed by TiDB (0.015 taps) and MySQL (0.01 taps). TiDB leads in T-Throughput with 16,500 tps, followed by HERMES at 8,000 tps and MySQL at 6,500 tps. HERMES maintains its lead in transactional analytics requests per dollar, while ranking second in transactional requests per dollar.

Serializability (SR) Results. Figure 4.9e and Figure 4.9f depict the throughput and operations-per-dollar frontiers under SR, respectively. TiDB is omitted as it does not support SR. HERMES (blue) exhibits a distinct frontier with a unique shape compared to RC and SI cases, reflecting the dependent nature of transactional and analytical workloads under SR. In both HERMES and MySQL, traditional and analytical transactions compete for lock access, leading to a decline in TA-Throughput as T-clients increase. This explains why HERMES ' frontier deviates from its bounding box, even though analytical queries are executed on the DuckDB side. MySQL (purple) follows a similar pattern but achieves lower TA-Throughput.

HERMES leads with the highest TA-Throughput (0.057 taps), followed by MySQL (0.01 taps). Both achieve a T-Throughput near 6,000 tps. HERMES excels in transactional analytics per dollar and ranks second in transactional requests per dollar.

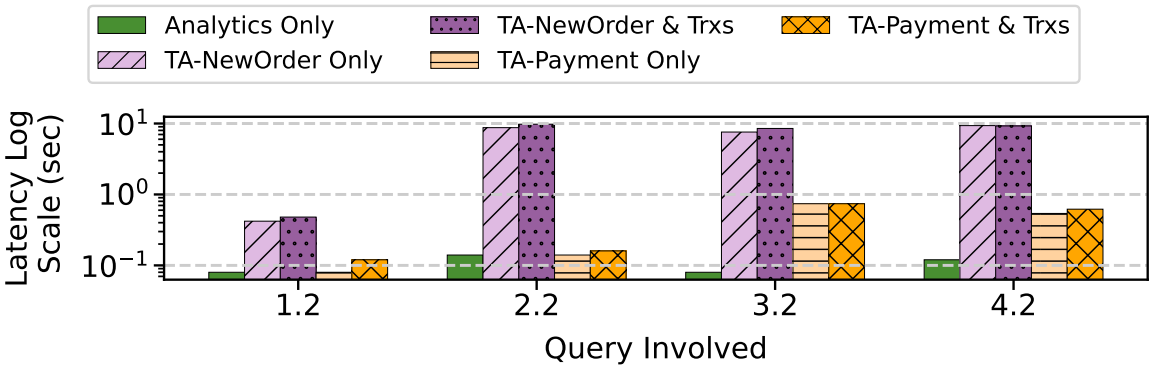
Overall, HERMES surpasses MySQL and TiDB in TAW across all isolation levels, in absolute performance and performance isolation.

TiDB Analysis

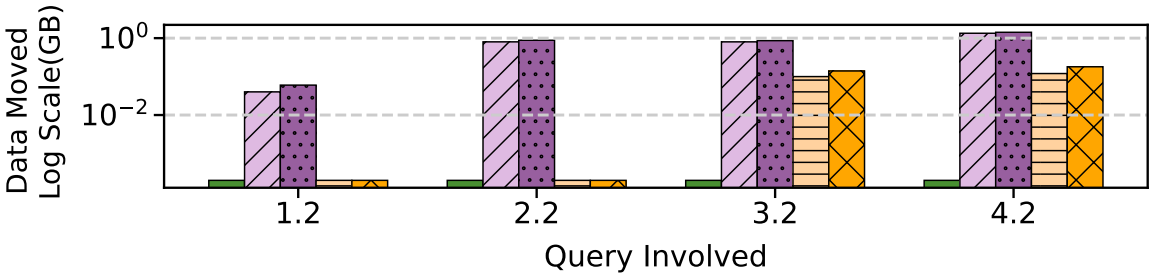
This section explores TiDB's results in detail, highlighting key findings and explaining why TiDB's performance in TAW falls significantly short of HATtrick.

Workload Configurations. We use three workload configurations to analyze TiDB's performance differences between transactional analytics in TAW and traditional analytics in HATtrick. First, Analytics-Only runs only the analytical component of the HATtrick benchmark, measuring TiDB's performance on traditional analytics without transactional interference. Next, TA-X workloads (where X is NewOrder or Payment) execute TA-X analytical transactions alone, isolating the impact of transactional analytics. Finally, TA-X & Trxs includes both TA-X analytical transactions and regular transactions, revealing TiDB's limitations under mixed workloads.

Comparison Results. Figure 4.10a shows TiDB latency results for selected SSB queries across



(a) Latency measured in seconds (secs) for the involved queries.



(b) Total data moved measured in gigabytes (GB).

Figure 4.10: Figure 4.10a and 4.10b display data from experiments conducted on TiDB, featuring various workloads: Analytics Only (no Trxs), TAW with TA-NewOrder Only (no Trxs), TAW with TA-NewOrder and Trxs, TAW with TA-Payment Only, and TAW with TA-Payment and Trxs. Figure 4.10a show query latency and Figure 4.10b total data transferred during execution.

different workload configurations, omitting other queries with similar performance patterns (e.g., Queries 1.1 and 1.3 resemble 1.2). Figure 4.10b displays the total data transferred from TiFlash nodes to the TiDB server node during execution.

Figure 4.10a shows that the lowest latency occurs in the Analytics-Only workload, where queries run on TiFlash nodes optimized for analytics, aligning with HATtrick results (see Figure 4.8a) showing the highest A-Throughput for TiDB. The figure also reveals significantly higher latencies for TA-NewOrder (Only/& Trxs) and TA-Payment (Only/& Trxs) workloads compared to Analytics-Only, consistent with TiDB results under TAW (see Figures 4.9a, 4.9c, 4.9e).

Additionally, Figure 4.10a illustrates that the execution time for TA-NewOrder & Trxs and TA-Payment & Trxs consistently surpasses that of TA-NewOrder Only and TA-Payment Only, respectively. This emphasizes the influence of concurrent transaction execution on the latency of transactional analytics in TiDB.

Query Plan Analysis. In the Analytics-Only workload, queries run entirely on TiFlash compute nodes optimized for analytics. In contrast, TA-NewOrder and TA-Payment query processing extends beyond TiFlash. Data from updated tables—LINEORDER for TA-NewOrder and SUPPLIER and CUSTOMER for TA-Payment—are first retrieved from TiFlash nodes, then transferred to TiDB server nodes for processing by the *UnionScan* operator. This operator likely ensures isolation by merging recent data from TiFlash write nodes with S3-accessed data. Parts of the query then run across both TiFlash nodes and the TiDB server, causing data transfers (see Figure 4.10b). The extent of data movement depends on computation level in TiFlash, and larger tables like LINEORDER require more merging time, explaining the higher latency for TA-NewOrder (Only/& Trxs) compared to TA-Payment (Only/& Trxs).

TiDB's TAW performance is hindered by query plan changes introduced by transactional analytics. In contrast, HERMES maintains consistent query plans, making it well-suited for both workloads.

4.6 Related Work

This section reviews current solutions for (near) real-time analytics.

HTAP Systems. HTAP systems unify TP and AP to enable real-time analytics. Single-system architectures often employ shared [23] or optimized [87, 86, 50, 72, 102, 133, 82, 130, 55] data structures for the two workloads, ensuring immediate availability of transactional data for analytical queries. This approach eliminates replication latency but may increase contention. Other HTAP systems separate TP and AP engines, either sharing the same storage layer [43, 82, 100, 56] for immediate data visibility or using decoupled storage [66, 145] to isolate resources and allow independent scaling. In decoupled setups, transactional changes are periodically propagated to the AP layer via Change Data Capture (CDC) or log-based replication, with minimal latency. Most HTAP systems tightly couple compute and storage components, though exceptions like F1 Lightning [145] theoretically support pluggable engines, albeit without verification.

Change Data Capture (CDC) Tools. CDC tools are designed to monitor and replicate changes—such as inserts, updates, and deletions—in source databases to maintain data consistency across systems. They typically analyze transaction logs (e.g., PostgreSQL's WAL or MySQL's binary logs) to detect

modifications and then stream these changes in standardized formats (e.g., JSON, Avro) to target systems. While CDC tools are essential for data replication, migration, and synchronization between systems, they generally do not perform complex data processing. Their primary focus is to ensure that target systems accurately reflect the latest changes from source systems in real time. Notable CDC tools include Debezium [47], GoldenGate [108], pg_logical [13], and StreamSets [68].

Streaming Data Platforms (SDP). SDPs are designed for real-time ingestion, transportation, and processing of data streams from various sources. Unlike CDC tools, which primarily replicate database changes, SDPs offer advanced data processing capabilities such as windowing, aggregations, and joins, essential for real-time analytics and event-driven architectures. While SDPs can integrate CDC tools to capture and stream database changes in real-time, their primary function is to facilitate the flow of diverse data types—including logs, metrics, sensor data, and other event streams—across systems. They enable low-latency, high-throughput data movement and support robust integration options for real-time data pipelines across different systems. Examples of SDPs include Apache Kafka [78, 1], Apache Pulsar [22], Amazon Kinesis [21], and Google Pub/Sub [60].

Cloud-Based Storage Services. Storage services such as Delta Lake [24] and Hudi [5] are designed to add transactional capabilities over cloud-based object storage, enabling reliable data management for large-scale analytical and transactional processing. These services implement structured data formats (e.g., Parquet, ORC) and define access protocols, supporting transactions on data stored in distributed object storage. For example, Delta Lake utilizes versioned metadata and transaction logs to track changes, ensuring data consistency. However, Delta Lake typically requires modifications when integrated with various TP engines. In contrast, Hudi emphasizes flexibility, providing native support for multiple TP and AP engines. Hudi benefits from CDC tools for capturing data changes and SDPs for efficiently processing data streams, thereby enhancing its ability to manage evolving datasets in real-time. Both Hudi and Delta Lake follow principles of Lambda and Kappa architectures [75], with real-time and batch processing layers that support the continuous integration and historical accuracy of data.

4.7 Conclusion

In this chapter we introduce off-the-shelf real-time transactional analytics, a system design that uses the existing TP and AP engines of an organization and achieves fresh real-time transactional analytics. Following this design, we develop a new service called HERMES, which serves as an intermediate layer between computation and storage. Our evaluation shows that HERMES can outperform current HTAP systems by a factor of 3 in transactional analytics.

Chapter 5

Conclusions

In this chapter, we first provide a summary of the main contributions of this dissertation (Section 5.1). Next, we discuss potential future research directions inspired by this work (Section 5.2). Finally, we conclude the dissertation with closing remarks in Section 5.4.

5.1 Summary

In this dissertation, we focus on two main problems observed in current Hybrid Transactional and Analytical Processing (HTAP) solutions: (1) there is no systematic methodology for evaluating HTAP systems based on their ability to achieve real-time analytics, and (2) there is no solution that enables organizations with existing Transaction Processing (TP) and Analytical Processing (AP) engines to adopt real-time capabilities without costly and time-consuming migrations to new HTAP systems.

Our first goal is to develop a systematic and intuitive evaluation framework that assesses HTAP systems based on two key dimensions: performance isolation and analytical query freshness. This methodology is designed to help users easily compare multiple HTAP solutions and select the one that best matches their real-time analytics needs.

Our second goal is to propose a novel HTAP architecture that enables organizations to achieve real-time analytics using their existing TP and AP engines, without requiring modifications or replacements. The architecture is designed to deliver fresh analytical query results, maintain performance isolation, achieve low-latency analytics, ensure end-to-end transactional consistency,

and support correct isolation levels for transactional analytics.

5.1.1 A Systematic Evaluation Framework for HTAP Systems

In the first part of this dissertation, we propose a systematic evaluation framework for assessing how well a system supports real-time analytics.

We introduce two fundamental metrics that uniquely characterize each system's capabilities. The first metric, *throughput frontier*, measures how the system performs and how effectively it shares resources between transactional and analytical workloads while minimizing interference during concurrent execution. The second metric, *freshness*, measures how up-to-date analytical query results are when executed.

To operationalize these metrics, we present a benchmark called HATtrick, which extracts the throughput frontier and freshness metrics from each system under evaluation. These metrics provide actionable insights into system performance, design trade-offs, and bottlenecks. Furthermore, we introduce a visualization methodology that makes it intuitive for users to compare multiple systems and assess which one best fits their application needs.

We use HATtrick to evaluate several systems with HTAP capabilities (e.g., TiDB [66]), demonstrating how users can systematically compare different platforms based on extracted results. Our findings show that current HTAP systems have made significant progress in improving performance and query freshness, though there remains considerable room for further optimization.

5.1.2 Off-the-Shelf Real-Time Transactional Analytics

In the second part of this dissertation, we propose a novel HTAP architecture tailored for organizations that already operate separate transactional processing (TP) and analytical processing (AP) engines, but seek to achieve real-time analytics without costly migrations to new HTAP systems.

We introduce an *off-the-shelf* architecture for real-time analytics, which builds on *existing TP and AP engines* with minimal or no modifications, and supports pluggable engine choices. The key insight is to insert a lightweight middle layer between database engines and storage, merging transactional logs with analytical reads on the fly. This approach avoids mandatory migration while enabling fresh analytical queries and delivering performance competitive with native HTAP

systems.

A critical goal of our design is to support efficient *Transactional Analytics*. Analytical components execute on the AP engine and transactional components on the TP engine, while the middle layer ensures end-to-end transactional consistency by enforcing the correct isolation level for analytical transactions.

To validate this architecture, we built HERMES, a prototype real-time transactional analytics system for the cloud. HERMES intercepts storage interactions from TP engines (e.g., logging to AWS EBS) and AP engines (e.g., reading from AWS S3), merging live transactional updates with analytical reads while preserving isolation guarantees.

We evaluate HERMES using MySQL [106] as the TP engine and FlexPushdownDB [146] and DuckDB [117] as AP engines. Our results show that HERMES introduces minimal overhead to existing engines. Compared against MySQL and TiDB [66] on standard HTAP benchmarks, HERMES achieves competitive performance and cost. To evaluate transactional analytics, we introduce the *Transactional Analytics Workload (TAW)*, an extension of existing HTAP workloads. Our experiments demonstrate that HERMES outperforms current solutions by up to $3\times$, confirming the feasibility of off-the-shelf real-time and transactional analytics.

5.2 Future Work

In this section, we outline potential directions for extending the work presented in this dissertation and highlight new research avenues that emerge from our findings.

5.3 Vision for HERMES

HERMES has the potential to evolve into a more powerful and versatile data orchestration layer. We envision several key areas of future expansion that will enhance its performance, scalability, and flexibility. These include offloading cache management, supporting distributed deployments, adding intelligent middle-layer services, integrating multiple engine types, and enabling continuous analytics through incremental view maintenance.

Distributed and Scalable HERMES. HERMES could be extended into a fully distributed system, where each node manages a specific partition of the cloud storage data. This sharded, replicated architecture would enhance resilience—ensuring that node failures affect only a subset of the data—and improve elasticity by allowing nodes to be dynamically added or removed based on workload demands [48, 38, 83, 30]. To manage partition ownership and correctly route transactional log entries from TP engines to the appropriate HERMES node, a lightweight coordinator is needed. This coordinator could be implemented using a replicated Raft service [104, 85] or a decentralized, coordinator-less approach (e.g., gossip-based membership and consensus [48, 83]), ensuring robust and efficient partition management. A distributed HERMES design would not only provide high availability and fault tolerance but also naturally enable more advanced features, such as support for multiple TP and AP engines, as we discuss next.

Advancing Middle-Layer Services. HERMES 's position as a cloud-native middle layer makes it ideal for unifying a range of performance optimizations—metadata layers to enable transactional support [25], caching layers for post-pushdown caching and query acceleration [143, 18], and pushdown layers to offload filtering and aggregation [10, 147]—all within a single, cohesive framework. Building on this foundation, HERMES can be extended to automatically detect schema changes in transactional engines (e.g., added columns, altered types, new indexes) and broadcast updated metadata to all subscribed analytical engines, eliminating manual coordination when schemas evolve. It can also perform lightweight in-transit data cleaning—such as trimming invalid values, standardizing formats, or enriching records via lookups—ensuring that analytical engines always receive high-quality, consistent data [79]. Furthermore, HERMES can incorporate adaptive pushdown: after merging the latest transactional updates, it selectively executes filters, projections, or aggregates in the most efficient engine—transactional or analytical—based on real-time cost estimates [28], thereby minimizing data movement and query latency. By bringing these optimizations together—schema management, metadata propagation, data cleaning, caching, and cost-aware pushdown—HERMES can serve as a powerful, all-in-one middle layer for modern cloud database architectures.

Multi-Engine and Multi-Tenant Support. A full deployment of HERMES could extend its capabilities to support multiple heterogeneous TP and AP engines simultaneously, allowing organizations to flexibly choose the best engines for their specific application requirements while still enabling real-time transactional analytics through HERMES 's unified interface [53]. Different TP engines could manage independent databases, while AP engines could execute queries that span multiple sources seamlessly, provided that a global timestamp oracle ensures a single, consistent snapshot across all engines—for instance, by using tightly synchronized clocks as in TrueTime [45] or a Hybrid Logical Clock scheme [80]. Furthermore, HERMES could be enhanced to support multi-tenant environments, providing strong isolation across tenants, enforcing resource quotas, and guaranteeing fair performance, thus ensuring that multiple users or applications can safely share a single HERMES deployment without interference.

Continuous Materialized Views. HERMES could further extend its functionality by maintaining continuously updated materialized views for frequently accessed analytical queries [63]. By continuously applying transactional updates to pre-defined view definitions, HERMES could keep summaries and rollups live at all times. Incoming queries could be automatically routed to these up-to-date materialized views for faster response times, significantly reducing query latency. Additionally, HERMES could monitor query patterns to detect popular or expensive analytical queries and proactively maintain their results. This approach would enable near-instantaneous responses for hot queries, while less frequent queries would fall back to accessing raw data. In essence, HERMES would offer a built-in, low-latency query caching mechanism, keeping materialized views synchronized seamlessly with the underlying transaction stream.

5.4 Concluding Remarks

In this dissertation, we address two critical gaps in the landscape of Hybrid Transactional/Analytical Processing (HTAP). First, we identify the absence of a unified, systematic methodology for evaluating how effectively HTAP systems support real-time analytics. Second, we observe that no existing solution enables organizations to extend their current Transaction Processing (TP) and Analytical Processing (AP) engines with real-time analytics capabilities without undergoing costly

and labor-intensive migrations to new HTAP platforms.

To address the first challenge, we introduced HATtrick, a comprehensive benchmark designed to capture two orthogonal dimensions of HTAP performance: the *throughput frontier*, which quantifies both the absolute system throughput and the degree of interleaving between transactional and analytical workloads; and *freshness*, which measures the recency of analytical query results relative to the latest transactional updates. We demonstrated how HATtrick 's visualizations of these metrics can help users systematically evaluate, compare, and tune HTAP systems, while also exposing subtle trade-offs that traditional performance benchmarks fail to reveal.

For the second challenge, we proposed a novel *off-the-shelf* HTAP architecture realized in the HERMES prototype. By inserting a lightweight middle layer between existing TP and AP engines and the underlying storage, HERMES delivers real-time analytics transactional analytics without requiring any changes to the engines themselves. Our evaluation showed that HERMES adds negligible overhead, outperforms native solutions by up to $3\times$ on transactional-analytics workloads, and proves the viability of non-intrusive HTAP adoption.

While these contributions lay a strong foundation, they also open several promising directions for future work. Extending HERMES into a fully distributed architecture would enhance resilience, elasticity, and support multi-tenant deployments at scale. Enriching the middle layer with capabilities such as adaptive schema evolution, metadata propagation, and lightweight in-stream data cleaning would ensure data quality while expanding HERMES's ability to support a wider range of applications. Furthermore, supporting heterogeneous, federated query processing across multiple TP and AP engines—and maintaining continuously updated materialized views for frequently accessed queries—would drive even lower query latencies and further broaden HERMES's applicability.

In closing, this dissertation lays a foundation for the next generation of HTAP systems—systems that are not only rigorously evaluated but also practically deployable. By uniting precise benchmarking with a non-intrusive, flexible architecture, we aim to lower the barriers to adopting true real-time analytics. We hope this work encourages the HTAP community to advance evaluation frameworks, design plug-and-play architectures, and establish clear design principles—ultimately enabling organizations of all sizes to seamlessly integrate real-time insights into their operations

and decision-making processes.

Bibliography

[1] 2024. Amazon Kafka. https://kafka.apache.org.

[2] 2024. Amazon S3. https://aws.amazon.com/s3/.

[3] 2024. Apache Flink. https://flink.apache.org/.

[4] 2024. Apache Gandiva. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/.

[5] 2024. Apache Hudi. https://hudi.apache.org.

[6] 2024. Apache Iceberg: The open table format for analytic datasets. `https://iceberg.apache.org`.

[7] 2024. Apache Parquet. https://parquet.apache.org.

[8] 2024. Apache Thrift. https://thrift.apache.org/about.

[9] 2024. Google Cloud SQL for MySQL. `https://cloud.google.com/sql/docs/mysql`.

[10] 2024. S3 Select and Glacier Select. https://aws.amazon.com/blogs/aws/s3-glacier-select/.

[11] 2024. TiFlash Disaggregated Storage and Compute Architecture and S3 Support. https://docs.pingcap.com/tidb/stable/tiflash-disaggregated-and-s3.

[12] Version 1.14.0. 2011. TPC BENCHMARK™ E.

[13] 2ndQuadrant Ltd. 2024. pglogical: Logical Replication for PostgreSQL. `https://www.2ndquadrant.com/en/resources/pglogical/`. Accessed April 2025.

[14] Vesion 3.0.0. 2011. TPH BENCHMARK™ H.

[15] Revision 5.11. 2009. TPC BENCHMARK™ C.

[16] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 967–980.

[17] Aisha Abdallah, Mohd Aizaini Maarof, and Anazida Zainal. 2016. Fraud detection system: A survey. *Journal of Network and Computer Applications* 68 (2016), 90–113.

[18] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European conference on computer systems*. 29–42.

[19] Amazon Web Services. 2023. Amazon Athena Documentation. `https://docs.aws.amazon.com/athena/latest/ug/what-is.html`. Accessed: 2024-04-27.

[20] Amazon Web Services. 2023. Amazon Aurora: Design Considerations for High Availability and Performance. AWS Whitepaper.

[21] Amazon Web Services. 2024. Amazon Kinesis Streams Developer Guide. `https://docs.aws.amazon.com/kinesis/`. Accessed April 2025.

[22] Apache Software Foundation. 2024. Apache Pulsar: Cloud-Native, Distributed Messaging and Streaming. `https://pulsar.apache.org`. Accessed April 2025.

[23] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research*.

[24] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[25] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, Vol. 8. 28.

[26] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[27] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598.

[28] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 261–272.

[29] Microsoft Azure. 2024. Azure Database for MySQL. `https://learn.microsoft.com/en-us/azure/mysql`.

[30] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services.. In *CIDR*, Vol. 11. 223–234.

[31] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, Rene Mueller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam J Storm, Yuanyuan Tian, Pinar Tözün, et al. 2017. Evolving Databases for New-Gen Big Data Applications.. In *CIDR*.

[32] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[33] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Transactions on Database Systems* 8, 4 (1983), 465–483.

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

[35] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[36] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. 2019. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data*. 1773–1786.

[37] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, K. P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul McJones, Jim W. Mehl, Irvin L. Traiger, William Wade, and Robert A. Watson. 1981. A History and Evaluation of System R. *Commun. ACM* 24, 10 (1981), 632–646.

[38] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[39] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *ACM Sigmod record* 26, 1 (1997), 65–74.

[40] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. [n.d.]. ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency. ([n. d.]).

[41] Weisi Chen, Zoran Milosevic, Fethi A. Rabhi, and Andrew Berry. 2023. Real-Time Analytics: Concepts, Architectures, and ML/AI Considerations. *IEEE Access* 11 (2023), 71634–71657. https://doi.org/10.1109/ACCESS.2023.3295694

[42] Chaoyi Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D Bond, and Yang Wang. 2023. Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases. In *13th Annual Conference on Innovative Data Systems Research (CIDR'23). January 8-11, 2023, Amsterdam, The Netherlands.*

[43] Google Cloud. 2024. AlloyDB: A fully managed PostgreSQL database service. https://cloud.google.com/products/alloydb?hl=en.

[44] Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, and Rui Oliveira. 2017. Htapbench: Hybrid transactional and analytical processing benchmark. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 293–304.

[45] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's Globally-Distributed Database. *ACM Transactions on Computer Systems* 31, 3 (2013), 8:1–8:22.

[46] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[47] Debezium Community. 2024. Debezium: Stream Changes from Your Database. `https://debezium.io`. Accessed April 2025.

[48] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[49] Ravishankar Ramamurthy David J DeWitt and Qi Su. 2002. A Case for Fractured Mirrors. In *Proceedings 2002 VLDB Conference: 28th International Conference on Very Large Databases (VLDB)*. Elsevier, 430.

[50] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

[51] Barbara Dinter, Carsten Sapia, Gabriele Höfling, and Markus Blaschka. 1998. The OLAP market: state of the art and research issues. In *Proceedings of the 1st ACM international workshop on Data warehousing and OLAP*. 22–27.

[52] Robert J Earle. 1994. Method and apparatus for storing and retrieving multi-dimensional data in computer memory. US Patent 5,359,724.

[53] Aaron J Elmore, Jennie Duggan, Michael Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. 2015. A demonstration of the bigdawg polystore system. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1908.

[54] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.

[55] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[56] Christian Garcia-Arellano, Hamdi Roumani, Richard Sidle, Josh Tiefenbach, Kostas Rakopoulos, Imran Sayyid, Adam Storm, Ronald Barber, Fatma Ozcan, Daniel Zilio, et al. 2020. Db2 event store: a purpose-built IoT database engine. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3299–3312.

[57] Jana Giceva and Mohammad Sadoghi. 2019. Hybrid OLTP and OLAP.

[58] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1716–1727.

[59] Google Cloud. 2024. *Google Cloud Dataflow.* Available at `https://cloud.google.com/dataflow`.

[60] Google Cloud. 2024. Google Cloud Pub/Sub: A Google-Scale Messaging Service. `https://cloud.google.com/pubsub/docs/overview`. Accessed April 2025.

[61] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

[62] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data.* 1917–1923.

[63] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[64] Riyaz Ahamed Ariyaluran Habeeb, Fariza Nasaruddin, Abdullah Gani, Ibrahim Abaker Targio Hashem, Ejaz Ahmed, and Muhammad Imran. 2019. Real-time big data processing for anomaly detection: A survey. *International Journal of Information Management* 45 (2019), 289–307.

[65] Daniel Hieber and Gregor Grambow. [n.d.]. Hybrid Transactional and Analytical Processing Databases: A Systematic Literature Review. ([n. d.]).

[66] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[67] IBM Corporation. 2024. *IBM DB2 Database.* IBM Corporation. `https://www.ibm.com/products/db2-database` Version 11.5.8.

[68] IBM Corporation. 2024. IBM StreamSets: Seamless Hybrid and Multicloud Data Integration. `https://www.ibm.com/products/streamsets`. Accessed April 2025.

[69] StreamSets Inc. 2024. StreamSets Data Collector Documentation. `https://streamsets.com/products/data-collector/`. Accessed: 2025-04-26.

[70] William H Inmon. 2005. *Building the data warehouse*. John wiley & sons.

[71] Arun Kejariwal, Sanjeev Kulkarni, and Karthik Ramasamy. 2017. Real time analytics: algorithms and systems. *arXiv preprint arXiv:1708.02621* (2017).

[72] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[73] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. 2016. Pocket data: The need for TPC-MOBILE. In *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference, TPCTC 2015, Kohala Coast, HI, USA, August 31–September 4, 2015. Revised Selected Papers 7*. Springer, 8–25.

[74] Ralph Kimball and Margy Ross. 2011. The data warehouse toolkit: the complete guide to dimensional modeling. John Wiley & Sons.

[75] Martin Kleppmann. 2019. Designing data-intensive applications.

[76] Thomas Koch. 2000. *Oracle: The Complete Reference*. Oracle Press.

[77] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID Transactions across Diverse Data Stores. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2742–2754.

[78] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of the NetDB Workshop*. 1–7. Athens, Greece.

[79] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment* 9, 12 (2016), 948–959.

[80] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical physical clocks. In *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. Springer, 17–32.

[81] Cockroach Labs. 2021. CockroachDB: Architecture Overview. Technical White Paper. urlhttps://www.cockroachlabs.com/docs/architecture/overview/.

[82] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[83] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[84] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173* (2012).

[85] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[86] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[87] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1598–1609.

[88] Feifei Li. 2023. Modernization of databases in the cloud era: Building databases that run like Legos. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4140–4151.

[89] Guoliang Li and Chao Zhang. 2022. HTAP databases: What is new and what is next. In *Proceedings of the 2022 International Conference on Management of Data*. 2483–2488.

[90] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 21–35.

[91] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[92] Roger MacNicol and Blaine French. 2004. Sybase IQ multiplex-designed for analytics. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 1227–1230.

[93] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 37–50.

[94] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 604–615.

[95] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[96] Microsoft Corporation. 2000. *Microsoft SQL Server Analysis Services Multidimensional Performance and Operations Guide*.

[97] Microsoft Corporation. 2023. SQL Server 2022 Documentation. https://learn.microsoft.com/en-us/sql/sql-server/. Accessed April 18, 2025.

[98] Elena Milkai, Yannis Chronis, Kevin P Gaffney, Zhihan Guo, Jignesh M Patel, and Xiangyao Yu. 2022. How good is my HTAP system?. In *Proceedings of the 2022 International Conference on Management of Data*. 1810–1824.

[99] Elena Milkai, Xiangyao Yu, and Jignesh M Patel. 2025. Hermes:Off-the-Shelf Real-Time Transactional Analytics. *Proceedings of the VLDB Endowment*.

[100] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics.. In *CIDR*.

[101] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[102] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[103] Alex Nordeen. 2020. *Learn Informatica in 24 Hours: Definitive Guide to Learn Informatica for Beginners*. Guru99.

[104] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[105] Oracle Corporation. 2024. *The InnoDB Storage Engine*. MySQL 8.0 Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html Accessed April 18, 2025.

[106] Oracle Corporation. 2024. *MySQL 8.0 Reference Manual*. https://dev.mysql.com/doc/refman/8.0/en/

[107] Oracle Corporation. 2024. *Oracle Database*. Oracle Corporation. https://docs.oracle.com/en/database/oracle/oracle-database/23/index.html Version 23c.

[108] Oracle Corporation. 2024. Oracle GoldenGate: Replicate and Transform Data. https://www.oracle.com/integration/goldengate/. Accessed April 2025.

[109] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1771–1775.

[110] O'Neil Pat, O'Neil Betty, and Chen Xuedong. 2009. The Star Schema Benchmark.

[111] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner (2014, January 28) Available at https://www. gartner. com/doc/2657815/hybrid-transactionanalyticalprocessing-foster-opportunities* (2014), 4–20.

[112] PostgreSQL. 2021. PostgreSQL Streaming Replication Documentation. https://www.postgresql.org/docs/current/warm-standby.html.

[113] PostgreSQL. 2021. Swarm64 HTAP Benchmark for PostgreSQL. (2021).

[114] PostgreSQL Global Development Group. 2023. *PostgreSQL Database*. `https://www.postgresql.org/` Version 16.

[115] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling highly contended OLTP workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.

[116] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. 2014. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 97–112.

[117] Mark Raasveldt and Hannes Mühleisen. 2023. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[118] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[119] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2043–2054.

[120] Kun Ren, Jose M Faleiro, and Daniel J Abadi. 2016. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*. 1583–1598.

[121] John Russell. 2013. *Cloudera Impala.* " O'Reilly Media, Inc.".

[122] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2016. L-store: A real-time OLTP and OLAP system. *arXiv preprint arXiv:1601.04084* (2016).

[123] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1, 3 (2016), 145–164.

[124] Mohit Saxena, Benjamin Sowell, Daiyan Alamgir, Nitin Bahadur, Bijay Bisht, Santosh Chandrachood, Chitti Keswani, G Krishnamoorthy, Austin Lee, Bohou Li, et al. 2023. The story of AWS Glue. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3557–3569.

[125] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[126] Dennis Shasha and Philippe Bonnet. 2002. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann.

[127] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. 2013. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1068–1079.

[128] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[129] Stanley Shyiko. 2022. MySQL Binary Log connector. `https://github.com/shyiko/mysql-binlog-connector-java`.

[130] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 731–742.

[131] Eugene Siow, Thanassis Tiropanis, and Wendy Hall. 2018. Analytics for the internet of things: A survey. *ACM computing surveys (CSUR)* 51, 4 (2018), 1–36.

[132] Alex Skidanov, Anders J. Papito, and Adam Prout. 2016. A column store engine for real-time streaming analytics. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1287–1297. `https://doi.org/10.1109/ICDE.2016.7498332`

[133] Alex Skidanov, Anders J. Papito, and Adam Prout. 2016. A column store engine for real-time streaming analytics. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1287–1297. `https://doi.org/10.1109/ICDE.2016.7498332`

[134] J Sreemathy, S Nisha, Gokula Priya RM, et al. 2020. Data integration in ETL using TALEND. In *2020 6th international conference on advanced computing and communication systems (ICACCS)*. IEEE, 1444–1448.

[135] Michael Stonebraker. 1986. The case for shared nothing. *IEEE Database Eng. Bull.* 9, 1 (1986), 4–9.

[136] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[137] Michael Stonebraker and Uğur Çetintemel. 2018. " One size fits all" an idea whose time has come and gone. In *Making databases work: the pragmatic wisdom of Michael Stonebraker*. 441–462.

[138] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM SIGMOD Record* 15, 2 (1986), 340–355.

[139] PingCAP TiDB. 2024. Deploy a TiDB Cluster Using TiUP. `https://docs.pingcap.com/tidb/stable/production-deployment-using-tiup`.

[140] PingCAP TiDB. 2024. TiDB Customers. `https://www.pingcap.com/customers/`.

[141] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[142] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[143] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 13–24.

[144] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment* 9, 5 (2016), 444–455.

[145] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

[146] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. Flexpushdowndb: Hybrid pushdown and caching in a cloud DBMS. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2101–2113.

[147] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1805.

[148] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[149] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. *Proceedings of the VLDB Endowment* 17, 5 (2024), 939–951.

[150] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and consistent cross-engine transactions. In *Proceedings of the 2022 International Conference on Management of Data*. 34–48.