Perseus: Achieving Strong Consistency and High Data Freshness for Scalable Geo-distributed HTAP

HAOZE SONG, The University of Hong Kong, Hong Kong SAR
XUSHENG CHEN*, Huawei Technologies Co., Ltd., China
RUIJIE GONG, The University of Hong Kong, Hong Kong SAR
ZEKAI SUN, The University of Hong Kong, Hong Kong SAR
TIANXIANG SHEN, The University of Hong Kong, Hong Kong SAR
CHENG LI, University of Science and Technology of China, China
HAO FENG, Huawei Technologies Co., Ltd., China
SEN WANG, Huawei Technologies Co., Ltd., China
HEMING CUI*, The University of Hong Kong, Hong Kong SAR

The rise of global data-driven applications has made geo-distributed hybrid transactional and analytical processing (HTAP) databases increasingly desirable. Existing distributed HTAP systems provide users with good performance on both transactions and analytical queries, and this good performance is scalable across a large number of data nodes. Unfortunately, these systems either provide weak consistency or incur bad data freshness when deployed geographically. In this paper, we present Perseus, a scalable HTAP database that enforces strong consistency for both transactions and analytical queries. To handle consistency efficiently, Perseus augments the classical dependency graph in concurrency control protocols to explicitly record the versions of data and their complete dependencies, implying which data need to be read together in a snapshot. To minimize data staleness on analytical queries (another important goal of HTAP), Perseus further introduces a new dynamic snapshot algorithm that chooses updates selectively.

Extensive evaluation results show that, compared to the HTAP databases with even weaker consistency, Perseus achieves up to ~ 90% lower visibility delay, a metric of data freshness, capturing the time interval during which transactional updates are committed to the database can be visible to analytical queries. Besides, Perseus is scalable across many nodes and robust to network instability.

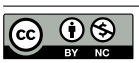
CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Hybrid Transactional and Analytical Processing, Consistency, Replication

ACM Reference Format:
Haoze Song, Xusheng Chen, Ruijie Gong, Zekai Sun, Tianxiang Shen, Cheng Li, Hao Feng, Sen Wang, and Heming Cui. 2025. Perseus: Achieving Strong Consistency and High Data Freshness for Scalable Geo-distributed HTAP. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 260 (September 2025), 29 pages. https://doi.org/10.1145/3749178

*Corresponding authors.

Authors' Contact Information: Haoze Song, hzsong@cs.hku.hk, The University of Hong Kong, Hong Kong SAR; Xusheng Chen, chenxusheng6@huawei.com, Huawei Technologies Co., Ltd., Shenzhen, Guangzhou, China; Ruijie Gong, rjgong@cs.hku.hk, The University of Hong Kong, Hong Kong SAR; Zekai Sun, zksung@cs.hku.hk, The University of Hong Kong, Hong Kong SAR; Tianxiang Shen, stx635@connect.hku.hk, The University of Hong Kong, Hong Kong SAR; Cheng Li, chengli7@ustc.edu.cn, University of Science and Technology of China, Hefei, Anhui, China; Hao Feng, feng.hao1@huawei.com, Huawei Technologies Co., Ltd., Shenzhen, Guangzhou, China; Sen Wang, wangsen31@huawei.com, Huawei Technologies Co., Ltd., Shenzhen, Guangzhou, China; Heming Cui, heming@cs.hku.hk, The University of Hong Kong, Hong Kong SAR.



ACM 2836-6573/2025/9-ART260
https://doi.org/10.1145/3749178

1 Introduction

Cloud providers make it easier to deploy online transaction processing (TP) applications across geo-distributed data centers [11, 12, 19]. Recently, the emergence of ***real-time decision-making applications*** (e.g., business intelligence [25, 93], dynamic pricing [80], and fraud detection [16, 76]) necessitates online analytical processing (AP) to be performed over the latest data generated by TP. This motivates the development of geo-distributed hybrid transactional and analytical processing (HTAP) systems, which handle both transactions and analytical queries in a single database.

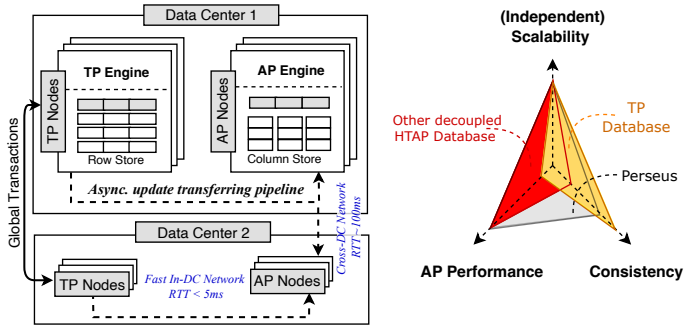
HTAP databases typically feature a TP engine for handling transactions and an AP engine for analytical queries. To efficiently manage mixed workloads, many distributed HTAP databases (e.g., [9, 34, 37, 51, 52, 66, 95, 97]) independently optimize two data copies: one for TP using row-oriented storage and another for AP using column-oriented storage. They employ an *update transferring pipeline* to capture and apply TP updates to AP data nodes asynchronously. By doing so, these decoupled HTAP databases achieve high performance for both TP and AP. We show an example of geo-distributed and decoupled HTAP systems in Figure 1a.

Achieving strong consistency in decoupled HTAP systems is challenging but essential. We define strong consistency in HTAP as the whole system operating as a single system for providing consistent external views [21, 88]. Specifically, the system should not only ensure strong isolation (e.g., serializability) but also provide a firm contract on data staleness for both TP and AP (e.g., regularity). In the literature, regularity [94] is a data synchronization property formally introduced by Lamport in regular registers [49]. Regularity guarantees real-time ordering among all writes (i.e., without any compromise) while allowing reads to return either the value written by the most recent complete write or a value written by a concurrent write. In the context of HTAP, regularity suggests that both TP reads and AP queries can at least see all TP writes that are committed before the TP transactions or AP query starts in real-time order.

Strong consistency has been advocated by many cutting-edge HTAP applications (e.g., [16, 70, 75]) as these applications may use the results of analytical queries to conduct real-time decisions automatically: clients may use the results of AP queries as the input of an automatic decision-making algorithm (e.g., AI-assisted), and then the algorithm automatically decides the payloads of the follow-up TP transactions [33, 44]. For instance, a real-time fraud detection application [94] may use AP queries to aggregate subgraph statistics to identify fraudulent communities and then freeze fraudsters' accounts through follow-up transactions. Weak consistency, such as snapshot isolation, allows AP queries to view snapshots without staleness bounds, potentially leading to outdated statistics and more false-positive case reports.

We show a more detailed example in §2.1. Our observation of customers' HTAP workloads also validates that strong consistency is increasingly desirable in real-time decision-making workloads, as the boundary between TP and AP is becoming increasingly blurred. Customers prefer to use transactions and AP queries interchangeably in their applications' logic. Prior to HTAP, application developers executed these analytical queries inside a TP database, which essentially provides the strongest consistency for all operations. HTAP makes it possible to process analytical queries on independently optimized storage and engines in favor of performance (e.g., $\times 7 - \times 10$ speed-ups [4, 5, 84]). We compare the capabilities of TP and HTAP systems in Figure 1b. Unfortunately, to our knowledge, ensuring the ***strongest consistency*** (strict serializability) is still an open problem for any decoupled HTAP database. Given this, our paper aims to build a scalable HTAP system with a strong consistency model (but not the strongest) that is capable of supporting our applications while ensuring high data freshness and scalability.

We present Perseus, an HTAP system that provides regular sequential serializability (RSS). RSS [35] is a strong consistency model in the literature and is tailored for read-only transactions.



(a) Geo-distributed HTAP. (b) Design Space of HTAP.

Fig. 1. (a) shows a geo-distributed HTAP; (b) compares the performance of TP database, PERSEUS, and other decoupled HTAP databases: PERSEUS achieves stronger consistency than other decoupled HTAP databases and better AP performance than TP databases.

We extend its definition from TP to HTAP by incorporating AP queries. In particular, RSS requires all read operations to observe a consistent snapshot that reflects the most recently committed updates and preserves causality. Note that causality describes the relationship where one operation influences the occurrence of another, which is essential for correctness. In real-world scenarios, causality can be captured through our system's internal message passing (such as requests originating from the same client session) or through users' preclaims for out-of-the-box causality (e.g., two requests from two causally unrelated client sessions).

We formally define RSS in §2.3 and discuss its desirability in §2.1. Compared to strict serializability (the strongest consistency), RSS forgoes only the real-time ordering requirements caused by order transitivity when a causally unrelated read-only request observes writes of a concurrent transaction. This mitigation does not affect the correctness of decision-making applications, as any mismatch should only occur within brief time windows (a few seconds). We further illustrate this in §2.1.

PERSEUS tackles two unique research challenges in ensuring RSS. The first challenge is to ensure that AP queries can observe all RSS ordering requirements in a lightweight manner. To address this challenge, PERSEUS uses a new piggybacking mechanism in the concurrency control layer to gossip complete transaction orders between TP nodes with minimal performance overhead. PERSEUS leverages dependency graphs for gossiping, which represent transactions as vertices and the order dependency between them as edges. Unlike the classical dependency graph that is only used for ordering direct conflict TP transactions, we augmented it into the Augmented Dependency Graph (ADG) to carry indirect dependency order and transmit it to AP nodes for preparing RSS-consistent snapshots. ADG indicates which updates must be read together in an RSS-consistent snapshot, enabling the efficient tracing of dependent transactions. Such a design also enables PERSEUS to ensure ***independent scalability*** between TP and AP: both good TP and AP performance can be maintained with an increasing number of either TP or AP nodes, as the task for ordering AP queries is detached from the critical path of ordering transactions.

The second challenge is to achieve low staleness for AP queries, where the data transfer delays for different updates can be largely different. To address this challenge, PERSEUS implements a new dynamic snapshot algorithm with two features: (1) it estimates the delivery of updates according to the recent network statistics; (2) it selectively chooses the updates to be included in a snapshot to minimize data staleness while still satisfying RSS requirements. This is possible because consistency models (including RSS) let a system freely determine whether a *concurrent transaction* should be

Ordering Invariants	SS	RSS	SSI	Comments
I_1. $Q \otimes W \wedge Q \rightarrow W \Rightarrow Q \prec W$	✓	✓	✓	Q cannot see future writes
I_2. $Q \otimes W \wedge W \rightarrow Q \Rightarrow W \prec Q$	✓	✓	✗	Q sees updates of all transactions finishing before it starts
I_3. $W \rightsquigarrow Q \Rightarrow W \prec Q$	✓	✓	✗	Q sees all causally dependent writes
I_4. $\exists R : (W \prec R) \wedge (R \rightarrow Q) \Rightarrow W \prec Q$	✓	✗	✗	Q sees newer view than all transactions finishing before it starts

Table 1. Representative ordering requirements of different consistency levels.

observed in the snapshot. Therefore, Perseus can proactively incorporate concurrent transactions committed after the query starts for better data freshness (see §4.4).

We prove that Perseus satisfies RSS consistency and independent scalability in §4.3. We implemented Perseus based on a widely studied TP codebase [64]. Using four standard benchmarks (TPC-C [24], TPC-H [23], TPC-E [22], and YCSB-T [26]), we implemented three HTAP workloads. We compared Perseus to three notable HTAP systems (TiDB [37], F1 Lightning [97], and Janus-HTAP [9]) and two relevant TP systems (Spanner-RSS [35] and CockroachDB [89]). Among these HTAP systems, only Perseus provides strong consistency (RSS), while others provide, at best, serializable snapshot isolation [31]. Our evaluation and analysis show:

- Perseus provides regular sequential serializability.
- Perseus achieves low data staleness. Perseus's visibility delay is up to 86.8%, 93.2%, 90.1% lower than Lightning, TiDB, and Janus-HTAP, especially for local queries.
- Perseus's TP and AP performance are independently scalable. Perseus's good TP performance was maintained as the number of AP nodes increased; vice versa.

2 Background and Motivation

2.1 Consistency Models

Consistency Models in HTAP. A consistency model is a contract between a system and its clients, specifying the acceptable return values for a set of operations. Traditional TP systems prioritize strong consistency to limit possible return values, simplifying the task for programmers to develop mission-critical applications on top of them. In turn, traditional AP systems are typically crafted with weaker consistency to improve performance. Unlike the two extremes, HTAP systems have been designed to cater to both TP and AP workloads within a unified system, enabling analytical queries to work on recently generated transactional data. A typical HTAP workload weaves transactions and analytical queries together: analytical queries can read the latest writes of transactions, and transactions may use queries' results to make real-time decisions. This complex interleaving of TP transactions and AP queries motivates us to ensure strong consistency for the whole HTAP system; otherwise, users have to manually identify transactions and analytical queries, carefully monitor their interleaves, and guess the potential side effects, which can be costly and challenging. Therefore, strong consistency is desirable for HTAP, particularly for certain types of emerging decision-making applications.

Motivating Decision-making Applications. We present an example of a fraud detection application in Figure 2. The logic of transactions and analytical queries is simplified for clarity. The application uses the same schema from TPC-E [22] (which was originally known as a TP workload). To assist fraud detection, the analytical query fetches the latest trade history from the TRADE_HISTORY table and conducts security checks by analyzing statistics from a large number of data rows. As shown in the example workflow (at the bottom of Figure 2), transactions and analytical queries collaborate in a round-robin manner. Upon discovering a suspicious trade via analytical queries, a new transaction will be made to mark the trade and its trader. Future analytical queries use these marks to detect risk and fraud. These marks are important for the following analytical queries since fraudsters often create fake accounts and perform fictitious trades between

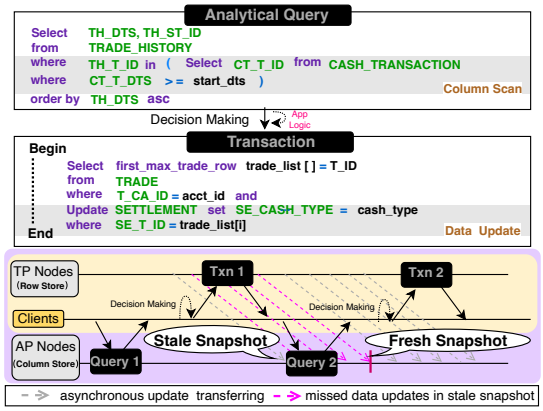


Fig. 2. This diagram shows an AP query and TP transaction from derived TPC-E [22]. The transaction uses the results of the query, interleaving transactions and AP queries together.

them [42]. With weak consistency (e.g., no regulation on snapshot staleness), anomalies can happen and be harmful to the quality of the fraud detection application.

We then emphasize the necessity of strongly consistent HTAP. Compared to running analytical queries on a pure TP system as read-only transactions (which ensures strong consistency by default), HTAP speeds up analytical queries with its optimized AP engine and columnar storage. Compared to running analytical queries on a traditional OLAP system, where new data is loaded through a time-consuming extract-transform-load process, HTAP provides valuable insights from fresh data.

Insight of Real-world Applications. Besides fraud detection, dynamic pricing [80] is another real-world decision-making application. Dynamic pricing uses data analytics to adjust prices in real-time based on customer demand. In such an application, TP continuously processes user transactions to update supply and demand, while AP makes real-time decisions and generates transactions to adjust prices. Real-time data analysis (e.g., reading the latest statistics) is critical to maximizing applications' revenue. Otherwise, AP queries might miss crucial updates (e.g., a sudden surge in demand), resulting in delayed or suboptimal decisions.

Furthermore, weak consistency can cause subtle anomalies unless extra application logic prevents them. For instance, consider an analyzer that detects an opportunity to lower a product's price and successfully commits the adjustment (e.g., applying a 20% discount). Under weak consistency, however, the analyzer may still read a stale snapshot even after making the change. Consequently, the analyzer might attempt another adjustment, unaware that the supply-demand dynamics have already shifted after its initial action. In practice, applications can be carefully structured (e.g., delay decisions or rely on specific timestamp fields) to avoid such duplicate decisions. However, ensuring correctness in this way requires expert knowledge, making the development challenging. In contrast, strong consistency ensures that AP queries always reflect the latest TP writes (I_2 in Table 1), preventing these anomalies and greatly simplifying application layer design.

The rationale for choosing RSS in PERSEUS. We begin by examining existing consistency models. Snapshot isolation ensures that reads (including TP reads or AP queries) operate on a snapshot without imposing additional constraints on operation order. Serializability differs from snapshot isolation in write skew anomalies of TP while providing the same guarantees for reads. There are a number of consistency models proposed atop serializability. *Strict serializability (SS)* [69] provides an abstraction that appears to execute transactions sequentially, one at a time, in an order consistent with the transactions' real-time order (i.e., the order observed by a hypothetical wall clock). *Serializable snapshot isolation (SSI)* [15] requires transaction executions that are equivalent

to *some* serial schedule on a single data copy. SSI does not necessarily restrict the set of allowed serializable schedules. Thus, all serializable orders are allowed in SSI. Between the two extremes, *Regular sequential serializability (RSS)* [35] enhances the requirement of SSI. It preserves causality and regularity for all read and write operations, and the real-time order for write operations.

Table 1 shows the comparison of the three serializable consistency models. We list four ordering invariants for the comparison. We define the used notations as follows: R denotes a read-only transaction; W denotes a read-write transaction; Q denotes a read-only AP query; O denotes any operation among R, W, and Q. $O_1 \otimes O_2$ denotes that O_1 *conflicts* with O_2: they access the same data, and at least one of them is W (i.e., write–write conflict, read-write conflict, or write-read conflict); $O_1 \longrightarrow O_2$ means O_1 finishes before O_2 begins according to a global wall clock; $O_1 \rightsquigarrow O_2$ means O_2 causally depends on O_1, e.g., O_2 is issued after O_1 in the same session; $O_1 \prec O_2$ means O_1 orders before O_2 in global historical log.

Intuitively, I_1 indicates that a read cannot see future writes, I_2 indicates the "read-the-latest-writes" property, I_3 indicates causality, and I_4 indicates real-time order between read operations. Among the three models, SS is the strongest and preserves all the invariants. Compared to SS, the invariant that RSS forgoes is I_4: a read-only operation from a client may see the updates that are temporarily not visible to other read-only operations from other clients. Beyond SS and RSS, SSI prevents "write skew" anomalies in snapshot isolation but still allows read operations to return outdated data versions and does not provide any constraint on data staleness.

After examining these available consistency models, we chose to build Perseus with RSS since I_2 and I_3 are important for providing staleness guarantees in a logical order between transactions and AP queries; thus, critical to the correctness of real-time decision-making applications. I_4 can be relaxed for decision-making applications because the anomalies only happen between read-only operations without involving updates. Decision-making applications use transactions and analytical queries in an interleaving manner, and any mismatch between analytical queries will not lead to incorrect transaction payloads. Meanwhile, we do not choose SS since SS is costly to achieve in geo-distributed HTAP. We formally discuss the performance and scalability issues of SS in supplementary material-§8.2 [83]. Briefly, as the update transferring pipeline between TP and AP nodes is asynchronous and non-transactional, it is expensive for an AP query to observe all other ongoing read-only transactions and then determine a total order among them.

2.2 Scope of our Paper

2.2.1 Read Semantics: Strong Read vs. Snapshot Read. Our paper focuses on how to achieve efficient and strongly consistent reads for AP queries. However, it should be noted that incorporating strong consistency is not essentially incompatible with weak consistency (e.g., stale snapshot reads). With our method, an HTAP system can still provide stale snapshots based on existing mechanisms (e.g., epoch-based snapshots) to improve query latency for non-critical long-running AP queries. Essentially, many TP systems (e.g., Spanner [21] and CockroachDB [89]) provide both strong and stale reads to users. Stale reads are still valuable for many HTAP scenarios (e.g., statistical reports).

2.2.2 Compare with Stream Processing (SP). Both SP and HTAP target real-time analytics. However, they differ in their data input, analytical operators, and consistency models, making them complementary to each other. Based on how SP is utilized for mixed workloads, we classify existing approaches into three categories (see Figure 3).

#1: SP as an independent AP engine. In this approach, transactions are managed by a stand-alone TP database, while the SP engine processes events triggered by each TP shard. To ensure high data freshness, the SP engine does not guarantee atomicity for transactions spanning multiple queues (e.g., the pink-colored events); instead, it offers eventual consistency: guaranteeing that

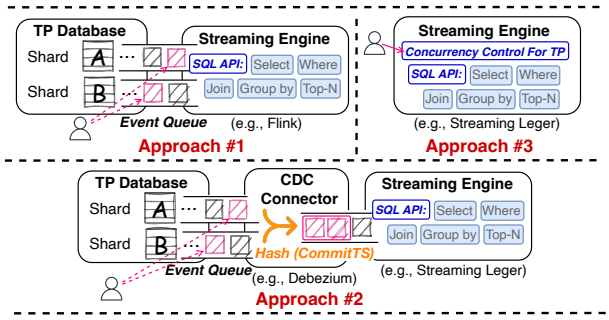


Fig. 3. Streaming Processing Architecture.

		Data Input for AP	Operators	Consistency	Freshness	Decoupled	TP Perf.
SP	Approach #1	Events (Patial History)	Window based	Eventual	Very High	✓	High
	Approach #2			Eventual	Medium	✓	High
	Approach #3			Strong	Very High	✗	Medium
Decoupled HTAP		Full Data Copy	Batched SQL	Strong	High	✓	High

Table 2. A comparison between streaming processing (SP) and HTAP.

updates made in the TP database will eventually be reflected in the SP engine. This limitation is not significant when using a non-shared RDBMS and can be tolerated in trend analysis scenarios. This is because non-shared RDBMSs can produce a centralized logical log (e.g., MySQL's binlog), and trend analysis typically does not require fully accurate data. Different from this, our work focuses on geo-distributed transactions and strongly consistent analysis.

#2: TSP as an AP engine. Recently, transactional stream processing (TSP) [1, 7, 59, 61, 99, 100] has been introduced to support shared mutable states inside SP engine (e.g., maintaining a streaming ledger inside SP for analysis). However, given an external shared TP database as an events source, it relies on a CDC (change data capture) connector to capture input streams. CDC connector provides two essential functionalities: (1) aggregating multiple data updates from different TP nodes into transactions, and (2) reconstructing logical statements from physical logs. Thus, the data freshness of this approach largely depends on the performance of the CDC connector. It commonly costs several seconds to minutes with existing advanced tools [2, 14]. Besides, this approach still offers only eventual consistency, since the TP database and TSP engine are independent, and the CDC connector guarantees atomicity but not real-time ordering of the two components. In contrast, HTAP uses in-database replication, eliminating the need for the CDC connector and thus can potentially offer stronger consistency and fresher data.

#3: TSP as both TP and AP engine. Another approach involves using TSP directly for both TP and AP, so that both operate on the same copy of data and naturally achieve strong consistency. However, to the best of our knowledge, this approach has not been widely adopted and shares the same limitations as coupled HTAP: limited optimization of the storage layer (§5.1) and poor performance isolation between TP and AP (Figure 15b). Furthermore, TSP's TP performance is sub-optimal due to additional costs for stream management. Therefore, using TSP directly for TP may be suitable for workloads that require light transaction support, whereas our work treats TP as a first-class citizen.

Takeaways: Table 2 further compares SP and HTAP in terms of their ability. In summary, SP and HTAP each have their own strengths in real-time analytics. Essentially, HTAP emerges as a real-time data warehouse and supports analysis over historical data. In contrast, SP supports multiple data sources and performs window-based analysis. In real-world deployments, SP and HTAP are often complementary and cannot fully replace one another. We believe that both will continue to coexist in the long term.

2.3 Regular Sequential Serializability

We use the RSS definition from [35], employing the same notation as Table 1 (see §2.1).

*Definition 2.1 (**Conflict Equivalence**).* Two schedules S_1 and S_2 are conflict-equivalent if (1) they are over the same set of TP transactions and AP queries, and (2) for every pair of conflicting operations (reads and writes), the relative order is the same.

*Definition 2.2 (**RSS in HTAP**).* An HTAP system is RSS if the execution of all operations is conflict-equivalent to being executed in a serializable schedule S that: (1) preserves causal ordering among operations (2) for any read-write transaction W and operation O, where O is not read-only or $O \otimes W$, if $W \rightarrow O$, then $W \prec O$.

2.4 Concurrency Protocols of Decoupled HTAP

We classify existing concurrency protocols of decoupled HTAP [9, 13, 37, 52, 58, 77, 79, 81, 82, 97] into two categories, and outline their benefits and limitations to pave the way for Perseus.

The first category allocates an order for each AP query individually. Typically, they use a centralized timestamp generator to coordinate transactions and AP queries together. For instance, TiDB [37] and HANA ATR [52] use global timestamps (or logical counters) for all transactions and queries. These mechanisms have the potential to provide strong consistency guarantees at the cost of scalability and freshness. Suppose they want to ensure that AP queries can see all TP writes that are finished before the query starts (i.e., I_2 in Table 1). A query must retrieve the latest timestamp from the global counter and wait for all updates belonging to the previous timestamps to become visible at AP nodes before execution. As a result, AP queries should always be blocked until the slowest ongoing transaction becomes visible at the AP nodes, even if the AP query does not read the updates generated by the transaction. This can lead to poor data freshness in a geo-distributed deployment, where obtaining a valid timestamp from a centralized service and waiting for updates involve multiple WAN RTTs. We evaluate TiDB in §5.3.

Advanced timestamp mechanisms [6, 8, 18, 30, 32, 38, 39, 57, 96] (e.g., hybrid logical clock) may overcome the limitations of centralized order assignment. However, to the best of our knowledge, they are not well-customized for HTAP systems, and naive implementations can lead to performance issues (e.g., compaction and parallelization are common optimizations in AP nodes, so tagging timestamps for each key in the column store can be expensive). We leave the search for possible solutions as our future work.

The mechanisms in the second category do not obtain a valid order for each query. Instead, they periodically generate consistent snapshots and allow AP queries to work on these snapshots. For instance, Google F1 lightning [97] uses "safe timestamps", a watermark-style mechanism. The maximum safe timestamp indicates that F1 lightning has ingested all changes into AP nodes up to that timestamp and can be used by AP queries. In this way, F1 lightning provides consistent but stale snapshots for AP queries. Vegito [81] adopts a gossip-style scheme to allocate consistent epoch numbers for all transactions. Each transaction and its dependent transaction are promised to be processed within or before the assigned Epoch. AP queries can then use a closed Epoch ID to read data. When deployed in a geo-distributed environment, it incurs poor data freshness since the time for gossiping a globally legal epoch ID can be costly. Janus-HTAP [9] features a batch graph for solving consistency between TP and AP. Transactions are first grouped into batches, and AP queries are required to find a set of cuts in the batch graph to retrieve data in AP nodes. To conclude, these mechanisms can be efficient and scalable, but fail to provide strong consistency since the snapshots are generated for AP queries without the knowledge of ongoing transactions.

There are two database systems called Janus. To distinguish them, we call the HTAP system Janus-HTAP [9] and the TP system Janus-TP [65].

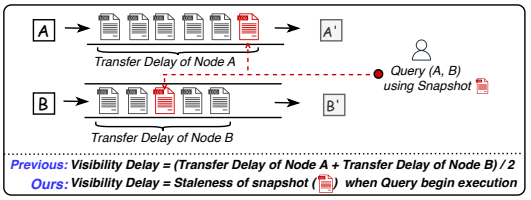


Fig. 4. A comparison of visibility delay metrics.

PERSEUS generates a snapshot and obtains a valid order for each AP query and thus belongs to the first category. However, unlike the mechanisms discussed in the first category, PERSEUS's snapshot includes only the relevant keys accessed by the AP queries while still capturing all ordering requirements (i.e., a partial view). Compared to global snapshots that contain the state of the whole database, partial snapshots can largely reduce the time for snapshot generation, especially for local queries (§4.2). Compared to the mechanisms in the second category, PERSEUS is able to achieve stronger consistency by capturing fine-grained transaction orders.

To implement such a snapshot mechanism, PERSEUS starts from a typical dependency graph protocol and handles several new research challenges (see §3.2). The intuition is that PERSEUS proactively records transaction dependency during transaction processing and attaches each update with its direct and indirect dependencies in an augmented dependency graph, indicating which updates must be read together in a lightweight manner.

2.5 Data Freshness and Visibility Delay

Data freshness is widely recognized as a common design goal of HTAP [44, 53]. To gauge freshness, a metric should quantify how recent the snapshot of TP is seen by AP queries. We use visibility delay for performance comparison, which is also studied in the previous papers [17, 37, 81, 95]. However, our definition of visibility delay differs slightly. We measure the staleness associated with the snapshot as seen when a query begins, rather than the time delay at which updates are transferred from TP nodes to AP nodes. We show a comparison in Figure 4. In short, we capture visibility delay at the granularity of snapshots rather than individual updates. This is because prior studies primarily compare the performance of data transfer services, whereas our study focuses on the latency of snapshot generation. Note that the average update transferring delay does not account for the time required to generate and prepare consistent snapshots for AP queries.

The formal definition of visibility delay used in this paper is provided below. Assume TP nodes have generated a series of data versions by performing new transactions (each transaction creates a version): V_0, V_1, ..., V_t. Thus, V_t should be the data version that includes the most recent update. We assume AP nodes have received the versions V_0, V_1, ..., V_a ($V_a \leq V_t$), and V_a is the latest version that has been installed on AP nodes. When a query Q arrives at AP nodes, we assume the AP engine selects V_q for execution, where V_q is a version suggested by the database. V_q can be either larger or smaller than V_a. If $V_q \leq V_a$, the query can be executed immediately based on available versions on AP nodes. Otherwise, the query needs to wait for more updates to be installed on AP nodes until the available version becomes larger than V_q. ***The visibility delay of*** Q is from the generation time of V_q (on TP nodes) to when Q's execution starts (on AP nodes). It consists of the time for determining V_q (i.e., the coordination time) and the time for V_q to become visible at the AP nodes.

In practice, we calculate the visibility delay of Q as the difference between the timestamp when Q begins execution and the timestamp of the snapshot used (i.e., the commit timestamp of the latest transaction in the snapshot V_q). While accurately evaluating such a visibility delay may require well-synchronized clocks (both for the visibility delay defined in our paper and in previous works), a widely available NTP clock is sufficient for evaluation purposes. This is because visibility delay is usually measured in tens of milliseconds [17, 37].

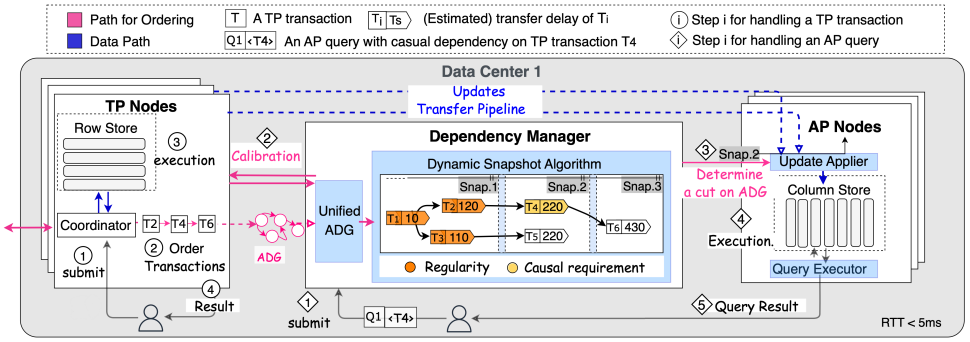


Fig. 5. Perseus has three key components: TP nodes serve transactions and generate ADG to transfer orders from TP to AP; Dependency Managers (DMs) track ADG from TP nodes and generate RSS snapshot (a consistent cut on the combined ADG) with high freshness; AP nodes absorb asynchronously transferred updates from TP nodes and serve queries based on the given snapshot.

3 Perseus Overview

Perseus is designed for geo-distributed HTAP. We consider a typical deployment. Nodes don't have globally synchronized clocks. Coarsely synchronized clocks (NTP [62]) are used for visibility delay evaluation and freshness optimizations, but are not used for correctness.

3.1 System Model

Architecture. Perseus consists of a group of *TP nodes* and a group of *AP nodes*. TP nodes maintain a row-oriented data copy of the database. Each TP node is responsible for a specific database partition, and the partition can be replicated to ensure high availability (referred to as TP replicas). TP nodes run an order-before-execution TP protocol [65] to serve client requests and update the database. Order-before-execution is widely-used for optimizing geo-distributed TP protocols [30, 78, 90, 98]. They avoid the use of expensive TP replication protocols by ensuring different TP replicas always independently produce the same results as long as they agree on the same order of input transactions. In particular, Perseus ensures that all TP replicas (whether local or geo-distributed) of a shard agree on transaction orders during the pre-ordering phase and execute transactions in the same order during the commit phase. However, since our paper focuses on AP snapshot generation, and geo-replication does not affect freshness results (as visibility delay is measured after a transaction is committed to all TP replicas), we omit the discussion on geo-distributed TP replicas for simplicity.

AP nodes have a column-oriented data copy. Each AP node can subscribe to updates from a set of TP nodes, allowing frequently accessed data to be grouped together. Transactions' orders and updates are asynchronously shipped from TP to AP nodes. Besides TP and AP nodes, Perseus has *Dependency managers (**DMs**)* that are responsible for collecting order history and generating consistent snapshots. AP nodes and DMs are deployed in pairs.

3.2 System Overview

To collect order history, Perseus uses dependency graphs to carry order information and manage snapshots, which is a serialization graph based on the transaction execution history. Each vertex in the graph represents a transaction. Each edge represents a dependency between two transactions due to conflicting data access. In particular, if transactions T_1 and T_2 make a conflicting access to some data item and a TP node executes T_1 before T_2, then $T_1 \longrightarrow T_2$.

Nevertheless, dependency graphs used in TP systems for ordering transactions (e.g., Janus [65], Rococo [63], and Aria [55]) are insufficient for scalable HTAP. In particular, Dependency graphs in TP protocols record only direct dependencies. TP databases do not require indirect orders since

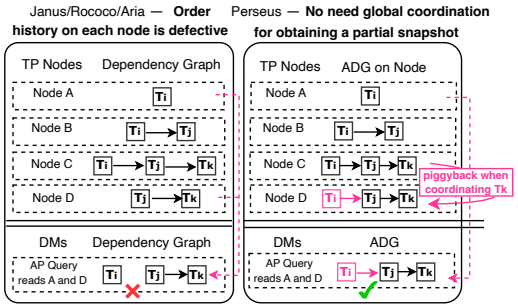


Fig. 6. A comparison between the typical dependency graph and augmented dependency graph (ADG).

all transactions share the same data copies. When a new transaction needs a valid snapshot, it contacts the relevant TP nodes (i.e., those storing the accessed data), where conflicts are detected and resolved locally [55, 65]. However, in HTAP systems, AP queries read data from AP nodes and do not participate in TP concurrency control. Instead of requiring AP nodes to query all TP nodes for a complete order history (i.e., obtaining a global snapshot by global coordination), PERSEUS enhances the dependency graph to gossip order information among TP nodes.

Based on such an observation, we propose augmented dependency graphs (ADGs) that include indirect order dependencies for each transaction. We show a comparison in Figure 6. In this example, we assume three transactions: T_i accesses nodes A and B; T_j accesses nodes B and C; T_k accesses nodes C and D. TP nodes execute transactions in the serializable order $T_i \prec T_j \prec T_k$. Typical dependency graph records only direct dependencies between transactions, and thus, a DM for nodes A and D cannot obtain a complete order history based on the records on TP nodes A and D. That is, using the information T_i on TP node A and using the information $T_j \prec T_k$ on node D, one cannot tell the order between T_i and T_j. Therefore, when an AP query wants to read data from both A and D at AP nodes, the order history is defective. The AP query may observe updates from T_k while missing updates from T_i, even if T_i precedes T_k in the serilizable order. Note that, in HTAP systems, TP nodes are typically row-partitioned, and AP nodes are column-partitioned (see §3.3). AP queries can aggregate data across a subset of row-partitioned nodes (i.e., node A and D), making such an order history essential. Violating serializable order can easily lead to read anomalies (e.g., I_1 in Table 1). To infer missed dependencies for correctness, a strawman approach involves having AP nodes communicate with all TP nodes to obtain a globally complete snapshot, which is costly and undermines scalability. In contrast, ADG handles this issue by recording indirect dependencies $T_i \prec T_j$ when coordinating T_k and piggybacks it from node C to D in its TP protocol (see §4.2).

ADG is crucial for PERSEUS to achieve RSS. Without ADGs, PERSEUS would need to query all TP nodes to ensure an AP query observes all committed transactions in order, which is prohibitively expensive in geo-distributed environments. We prove in §4.3 that ADG is sufficient for consistent partial snapshots and satisfies all RSS ordering requirements. Intuitively, with ADG, a partial snapshot always identifies the latest transaction in the dependency graph (e.g., T_k), and its dependencies are complete since all predecessors are ordered by TP nodes and gossiped. Our evaluation (§5.6) shows that ADG introduces acceptable overhead.

System Workflow. We then give a high-level description of how ADGs are generated, transferred, and used for snapshot generation. As shown in Figure 5, in PERSEUS, transactions are executed on TP nodes (left side), while AP queries are executed on AP nodes (right side). ADGs are generated on TP nodes and collected by DMs. To execute a transaction, clients submit their payloads to a nearby coordinator. The coordinator resolves conflicts between transactions in the ADGs by reordering them. Once the conflicts are resolved, an execution order is determined across TP nodes, and transactions are executed according to this order.

Meanwhile, the determined ADG is asynchronously transferred from TP nodes to DMs. DMs proactively track the ADG from TP nodes and generate a *partial snapshot*, which is essentially a consistent cut of the collected ADGs without including the actual data.

To invoke an AP query, clients first submit it to a DM. The DM then *calibrates ADG* with TP nodes to enforce RSS. This calibration step prompts TP nodes to send the latest ADGs to DMs, which includes all transactions committed before the request. This ensures AP queries (initiated before calibration) observe all prior committed transactions to enforce "read-latest-write". We detail the coordination and calibration algorithm in §4.2 and correctness in §4.3.

Freshness Optimization. Besides ensuring RSS, DMs employ a dynamic snapshot algorithm that eliminates non-critical dependencies in ADGs to optimize the freshness of AP queries. Recall that RSS ensures a query observes all transactions that were completed before the query began. Once RSS is satisfied, Perseus can freely decide which concurrent transactions (and their updates) to include in the snapshot. This presents a new opportunity for better freshness. However, there is a dilemma: including more updates may cause delays if these transactions or their predecessors' updates must be shipped from a remote data center, since the shipping time may exceed the essential time required for snapshot generation.

To address this dilemma, Perseus annotates each transaction in the ADG with its estimated available time and prunes straggler branches that are not required to satisfy RSS. An example is shown in Figure 5, and the algorithm is detailed in §4.4. In this example, Q_1 is submitted with a causal dependency T_4. Same as other works [35, 54], Perseus detect causal dependency through message passing. In Perseus, a client can send both transactions and analytical queries. The dependency in this example is detected since both T_4 and Q_1 are from the same client session. Then, the DM generates a fresh partial snapshot (i.e., $Snap.2$) for it. $Snap.2$ includes T_1 to T_3 to ensure regularity (including serializability and "read-the-latest-writes") and includes T_4 for causality. Since T_5 has a similar available time as T_4, Perseus proactively includes it in the snapshot. T_6 is discarded.

3.3 Transaction and Query Model

Transaction Model and its Limitation. Like other order-before-execution TP protocols (§6), Perseus avoids contention-caused aborts by pre-ordering. Perseus supports dependent transactions, whose inputs depend on execution results, by permitting inter-server communication during execution. To ensure deterministic execution, Perseus prohibits non-deterministic logic within transactions and disallows user-initiated aborts. These restrictions may limit Perseus's support for interactive transactions [3], which remains an open research problem for all deterministic approaches. In our prototype, transactions are implemented as stored procedures, which are commonly used in real-world workloads [24, 74].

AP Model and Data Partition. Perseus defines two types of AP queries: local and geo-distributed. A local query accesses data from TP nodes within a single data center, while a geo-distributed query accesses data across multiple data centers. Perseus optimizes freshness for both types by generating partial snapshots. To achieve this, Perseus requires prior knowledge of which data and data centers are accessed by Q, allowing it to obtain a partial snapshot by communicating only with the relevant data centers. In a geo-distributed database, this knowledge is easy to obtain.

Although in Perseus, the TP nodes are partitioned by rows and the AP nodes are partitioned by columns, relational tables are first partitioned by data centers (or regions), a common practice in geodistributed databases (e.g., regional tables in CockroachDB [47] and geo-partitioning in Spanner [87]). A table for a region can be identified by its table name. For instance, a table containing customer data from New York may use customer_NY as its table name. Thus, it allows an AP query to determine relevant regions and nodes by table names and columns. Note that columns are easy to detect in *SQL* and can be automatically achieved with existing tools [28, 90].

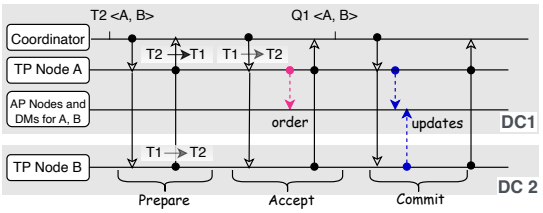


Fig. 7. This diagram shows PERSEUS's message flow to coordinate T_2. We assume T_2 accesses TP nodes A and B, while there is another ongoing T_1 that also accesses TP nodes A and B. T_1 conflicts with T_2 ($T_1 \otimes T_2$).

4 Protocol and Implementation

4.1 TP Protocol and ADG Generation

Overview. PERSEUS resolves conflicts between TP transactions by having coordinators determine a single execution order. Since multiple independent coordinators process transactions simultaneously, they must first contact participating TP nodes to collect conflicting transactions (potentially issued by other coordinators) and resolve these conflicts. As shown in Figure 7, PERSEUS's TP protocol consists of three phases: *Prepare*, *Accept*, and *Commit*. In the Prepare Phase, the coordinator detects conflicting transactions from participating TP nodes. In the Accept Phase, the coordinator provides TP nodes with the determined single order. Finally, in the Commit Phase, the TP nodes are informed to execute the transaction.

TP Protocol. Algo 1 shows the TP protocol on coordinators and Algo 2 shows the protocol on participating TP nodes. When a client submits a TP transaction to the coordinator, the coordinator handles the transaction by evoking *DoTransaction(T)*. We use the example in Figure 7 to illustrate the process. In this example, nodes A and B of T_2 are participating TP nodes. After receiving the prepare message, nodes A and B add the transaction to their local ADGs and send the combined ADG back to the coordinator. When adding conflict transactions into the local ADG, TP nodes order them according to the received order. In Figure 7, node A implies $T_2 \longrightarrow T_1$, and node B implies $T_1 \longrightarrow T_2$. Then, the coordinator combines the collected ADGs from TP nodes (i.e., combining the nodes and edges in received ADGs together) and potentially forms cycles in the graph ($T_1 \leftrightarrow T_2$). To ensure a single order, the coordinator cuts the graph to be acyclic. If and only if the combined ADG is originally acyclic, the coordinator can skip the *Accept Phase* and commit the transaction (Line 10, Algo 1). Otherwise, in the *Accept Phase*, the coordinator should notify nodes A and B about the determined order. Nodes A and B then update their local ADGs according to $T_1 \longrightarrow T_2$.

Upon the determined order has been received, TP nodes asynchronously pass the finalized ADGs to DMs and AP nodes (the pink arrow in Figure 7), which is ahead of the updates made by the transactions, since updates are generated in the commit phase. Transferring orders ahead of updates does not hurt correctness since each TP node has persisted the ADGs in its local storage. If failures occur in the commit phase, the ordered transactions can be re-executed accordingly (note that the transaction model prevents non-deterministic transaction logic, §3.3). Finally, in the *Commit Phase*, the transaction is executed. Updates are transferred to AP nodes streamingly.

Incoparting ADG into TP Protocol. Figure 7's workflow is adequate for coordinating a TP transactions. However, recall the issues in Figure 6. PERSEUS should carry additional indirect order dependencies to facilitate snapshot generation on AP nodes. Thus, in the real implementation of *Prepare Phase*, TP nodes send the ADG consisting of all recorded dependencies of T to the coordinator instead of the direct dependencies. Reusing the example in Figure 6, when coordinating T_k, node C replies the complete ADG ($T_i \longrightarrow T_j \longrightarrow T_k$). Then, when the coordinator combines the

Algorithm 1: Code for TP coordinator in PERSEUS

```
1  state D_T ← Dependency of transaction T
2  state F_T ← Set of nodes have received T and D_T
3  function TPCoord::DoTransaction(T):
4      G_coord ⟨D_⋆, F_⋆⟩ ← ∅
5      N ← CaculateNodes(T)
       // Phase.1: Prepare - Detect transaction conflicts
6      send PreAccept(T) to n ∈ N
7      wait receive PreAcceptRep(G_n) from n ∈ N
8      G_coord ← ToBeAcyclic(G_coord)
9      if ∀ n ∈ N, G_s ⟨D_T, F_T⟩ are identical :
10         G_coord ← Union(G_coord, ∪_{n∈N} G_n)
11         goto Commit Phase
12     else:
13         N' ← { n ∈ Majority(D_T) }
14         G_coord ← Union(G_coord, ∪_{n'∈N'} G_{n'})
15         goto Accept Phase
       // Phase.2: Accept - Determine transaction order
16     send Accept(T, G_coord) to n ∈ N
17     wait receive AcceptReply(Ack.) from n ∈ N
18     goto Commit Phase
       // Phase.3: Commit - Notify the determined order results
19     send Commit(T) to each n ∈ N
20     return to client after receiving execution results
```

ADGs and determines the order, the dependency of $T_i \rightarrow T_j$ are piggybacked to node D in the *Accept Phase* (or the *Commit Phase* when the graph is originally acyclical).

Implementation-level Optimization. Transferring ADGs incurs additional overhead. PERSEUS optimizes it by letting each TP node be notified of the existence and the dependency of each transaction only once. To do so, PERSEUS binds a set $\mathcal{F}_T$ with each transaction in the ADG (see Algo 1). A TP $node_i$ is in $\mathcal{F}_T$ if and only if the transaction and its dependency are already known to $node_i$. When receiving a transaction T, the TP node will filter the graph of T in the local ADG $\mathcal{G}_l$, and only piggyback the transaction order in $\mathcal{G}_l$ that are unknown to other involved TP nodes to avoid redundant work. For example, if all other participating nodes are in the $\mathcal{F}_T$ of a dependency transaction, the TP node will not send it to the coordinator. Similarly, in the *Accept Phase*, the coordinator does not send the dependency transaction to a participating TP node if the node is in the $\mathcal{F}_T$ of the transaction.

4.2 RSS-consistent Snapshot Generation

Overview. §4.1 has illustrated how to generate ADGs and updates on TP nodes and transfer these ADGs and updates to DMs/AP nodes; we now show how to leverage ADGs to generate snapshots for AP queries. As pointed out in §2.4 and §3.2, PERSEUS generates an *on-demand partial snapshot* and satisfies RSS consistency. On-demand property refers to generating and preparing snapshots for each AP query independently, instead of letting a batch of AP queries work on the same snapshot. This is critical for achieving RSS's "read-the-latest-write". Partial property refers to obtaining the snapshot without cross-data-center global coordination: DMs should only communicate with the nodes where the data has been accessed. This property is crucial for ensuring low visibility latency and excellent scalability (particularly for local queries).

Thanks to ADGs, PERSEUS can realize *on-demand partial snapshot* since AP nodes and DMs can now communicate with only relevant TP nodes to get the latest and complete orders of transactions. "Read-the-latest-write" is achieved by a calibration step. Before generating a snapshot, DMs trigger calibration requests, prompting TP nodes to send the latest ADGs to DMs, which include all transactions committed before the request. This ensures AP queries (initiated before calibration)

Algorithm 2: Code for TP nodes in PERSEUS

```
1  state G_s ← Local dependency graph maintained by TP Node
2  function TPNode::PreAcceptRecv(T):
3      AddNewTxn(G_s,T)
4      G_res ← TraceDep(T)
5      return G_res
6  function TPNode::AcceptRecv(T,G_c):
7      ForceUpdate(G_s,G_c)
8      return Ack.
9  function TPNode::CommitRecv(T,G_c):
10     if AcceptPhaseIsSkipped : Union(G_s,G_c);
11     res ← Exection(T)
12     return res
```

can observe all prior committed transactions in real-time order. Note that such calibration is sufficient to ensure RSS. The tricky part is that, compared to strict serializability, RSS relaxes the linearizable order between AP queries (§2.1). Thus, being unaware of the latest reads from other DMs/AP nodes is safe. There is no synchronization overhead between DMs and AP nodes.

AP Protocol. Same as TP protocol, PERSEUS let the first visited AP node (with its co-located DM) be the coordinator of the query. The coordinator assigns a snapshot to the AP queries, combines the execution results, and sends the execution results back to clients. Algo 3 shows the code of AP coordinators. Algo 4 shows the code of other DM participants. Executing an AP query has two phases. First, the coordinator finds an appropriate snapshot. Second, the coordinator and other non-coordinator AP nodes execute the AP query according to the given snapshot. Our paper focuses on the first phase (i.e., the snapshot generation) and uses existing works for query execution [56, 95].

Let us assume a client issues an AP query Q to the coordinator. The DM calculates the set of columns C and the set of TP nodes $\mathcal{N}$ that will be accessed by Q. C can be obtained by analyzing SQL, and $\mathcal{N}$ can be obtained by partition mapping (see §3.3). With $\mathcal{N}$, PERSEUS can know whether Q is a local query or a geo-distributed query and which TP nodes should be communicated for calibrating ADGs. As the AP query may need to access multiple AP nodes, the coordinator should send a prepare message to all participants (Line 4, Algo 3) and let them calibrate ADGs from $\mathcal{N}$.

When the AP nodes receive the prepare message, each DM sends a calibrating request to relevant TP nodes in $\mathcal{N}$ to update its local ADG (Line 4, Algo 4). In this step, the messages from TP nodes only carry transaction IDs that have been committed but not confirmed to be received by the DMs, as these transactions' ADGs can be later obtained asynchronously. PERSEUS only relies on these IDs to confirm the completeness of ADGs. Then, DMs send the updated ADGs to the coordinator.

After receiving the calibrated (updated) ADGs from DMs, the coordinator unions the received graphs by combining all the vertices and edges from the original graphs into a single graph. The unioned graph is still acyclic since PERSEUS's TP protocol prevents forming cycles. The graph unionized by the coordinator suggests a snapshot for the AP query Q. For causality, PERSEUS also combines system-detected causal dependency $\mathcal{D}_c$ in the unioned graph.

4.3 Correctness and Scalability Proof

We prove that PERSEUS achieves RSS and independent scalability (i.e., detaching the ordering of AP from TP's critical path) in supplementary material-§8.1 [83].

4.4 Optimize Freshness via Dynamic Snapshot

Next, we introduce a new dynamic snapshot algorithm to optimize the freshness of generated snapshots, which is not critical to the correctness of RSS but improves performance. Recall that after meeting the requirements of RSS, PERSEUS can still optimize the freshness of the snapshot

Algorithm 3: Code for AP coordinator (DM) in Perseus

```
1  state G_coord ← Local graph maintained by Dependency Managers.
2  function APCoord::DoAnalyticalQuery(Q):
3      N, C ← CaculateColumns(Q)
       // Phase 1: Partial Snapshot Generation – Collect the order requirements for RSS from relevant TP nodes (those
         containing the accessed data) and unify them.
4      send PrepareQuery(Q, C, N) to all n ∈ N
5      wait receive PrepareQueryReply()
6      G_RSS ← ∪_{n∈N} G_n      // G_n contains committed transactions at TP nodes
7      G_coord ← Union(G_coord, G_RSS)
8      Snap ← CaculateSnap(Q, C, G_RSS) ∪ D_causal
       // Phase 2: Query Execution – Executing Queries on AP Nodes
9      send CommitQuery(Q, Snap) to all n ∈ N
10     wait receive CommitReply(Result_n) from all n ∈ N
11     res ← AggregateResults(∪_{n∈N} Result_n)
12     return res
13 function APCoord::CaculateSnap(Q, C, G_RSS):
14     waitlist ← ∅
15     t_read ← GetTimeNow.Latest
       // Step 1: Form waitlist — Add all relevant transactions to the waitlist, ensuring that the waitlist contains a
         superset of the transactions necessary for RSS (G_RSS).
16     for each column in C do
17         for each T in G_coord do
18             if T.touchedColumns ∩ C ! = ∅ :
19                 waitlist ← waitlist ∪ {T}
       // Step 2: Optimize Freshness — Exclude concurrent transactions (those not in G_RSS) that have significant
         estimated shipping delays.
20     for each T in waitlist and T not in G_RSS do
21         if T.estimatedTime - t_read > threshold :
22             waitlist ← waitlist - {T}
23     return waitlist
```

by incorporating concurrent transactions. This is because RSS itself allows reads to return either the value written by the most recent completed transaction or a value written by a concurrent transaction that will eventually be committed (§2.3).

Such transactions can be common in Perseus due to two aspects: First, Perseus transfers orders ahead of updates (Figure 7). Thus, DMs can see concurrent transactions in their local ADG as soon as possible and detect them in the snapshot generation. Second, the latency of transaction execution (and the delivery of their updates) can differ in a geo-distributed deployment. Specifically, when a query is waiting for updates that are necessary for ensuring RSS from a far-away data center, Perseus can incorporate more transactions whose updates (and their predecessor's updates) are available from a local (or nearby) data center. Perseus can do so as long as the newly incorporated transactions will not increase the wait time of updates, and the derived snapshots still satisfy RSS.

To determine which concurrent transactions should be included in a snapshot at runtime, DMs estimate the shipping delay for each transaction's updates upon receiving their orders. The estimation is based on two factors: (1) the number of network rounds required for the updates to reach the AP node, (2) the estimated RTT between DMs and TP nodes. The estimated shipping time is then calculated as the product of these two factors. Note that the first factor is obtained and transmitted along with the ADG (depending on whether the transaction uses the fast or slow path), while the second factor is estimated statistically using NTP-synchronized clocks.

Given the estimated shipping delay, the DM will generate a list of transaction IDs for each query (called *waitlist*). This *waitlist* comprises the concurrent transactions that are intended for inclusion in the snapshot. In particular, the DM constructs a *waitlist* by traversing its local ADGs. In the first step (see Lines 16–19, Algo 3), the DM adds all relevant transactions to the waitlist. This

Algorithm 4: Code for non-coordinator DM in PERSEUS

```
1  state G_node ← local graph maintained by AP Node
2  function APNode::PrepareQueryRecv():
3      N' ← SubscribedTPNode()
4      send CalibrateGraph() to all n' ∈ N'
       // Check real-time Constraints from TP nodes
5      wait receive CalibrateReply(G_n') from all n' ∈ N'
6      G_node ← Union(G_node, ∪_{n'∈N'} G_n')
7      return G_node
8  function APNode::CommitQueryRecv(Q, Snap):
9      for each T in Snap do
10         for each column in T.touchedColumns do
11             if received T's updates for column :
12             merge updates to AP storage
13             else: block until receiving the updates
14     Result ← ExecutionQuery(Q)
15     return Result
```

includes both committed transactions (which are required by RSS) and concurrent transactions. In the second step (see Lines 21–22, Algo 3), the DM removes transactions from the waitlist if they are associated with significant shipping delays but are not required by RSS. Without loss of generality, we use a configurable parameter (*threshold*) to determine which transactions should be removed. Implementation-wise, we set the *threshold* as the largest estimated shipping delay of the essential ordering requirements of RSS (i.e., the largest estimated shipping delay of transactions in $\mathcal{G}_{RSS}$). By doing so, any newly incorporated concurrent transactions will not increase wait times while improving data freshness.

Note that PERSEUS does not rely on a precise estimation for the algorithm since PERSEUS can always stop waiting for the transactions with an under-estimated shipping delay as long as the necessary updates of RSS are received. In contrast, over-estimation may indeed influence the performance as the algorithm may ignore them and miss the optimization opportunity. However, overestimation is rare as we use routine statistics to estimate the delay and capture regular traffic. System exceptions (e.g., congestion) lead to underestimation. Thus, coarsely synchronized clocks (e.g., NTP clocks [62]) are sufficient for correctness and performance benefits. Even with overestimation, our approach can still consistently outperform the method without optimizations.

4.5 Fault-tolerance and Garbage Collection.

In PERSEUS, dependency graphs are transferred to DM after the transaction order has been finalized by its TP protocol (i.e., accepted by the majority). Since PERSEUS eliminates contention-induced aborts through pre-ordering and does not support user-level aborts (see §3.3), transactions are guaranteed to commit once their order is confirmed. Therefore, it's safe to transfer ADGs before executing the transaction. One potential concern is the possibility of aborts introduced by coordinator failures, which we address below.

Coordinator Failures. In case of a TP coordinator failure, the participating node for transaction T will detect that T has not progressed to the commit within a specified timeout period. Then, another healthy server will take up the role of the failed coordinator to commit any unfinished transactions. There are two possible scenarios for each unfinished transaction. If the old coordinator crashed after the accept phase, the new coordinator will learn the finalized order from the majority of participating nodes. Otherwise, the coordinator will retry T as a new transaction, going through the normal transaction protocol. Note that the second scenario happens before the dependency graph is transferred to DM, thus not affecting AP snapshot generation. On the other hand, PERSEUS handles AP coordinator failure by directly retrying the queries.

Epoch-based Garbage Collection. Perseus avoids transferring redundant ADGs by notifying each transaction only once (see implementation-level optimization in §4.1). To further reduce the storage cost of ADGs, Perseus uses a garbage collection mechanism similar to prior work [41, 63, 92], periodically removing outdated ADGs. Specifically, each TP node maintains an epoch number agreed upon via distributed consensus (e.g., Raft [68]) between TP nodes in several seconds. Then, a TP transaction is tagged with an epoch number when it starts. The epoch number on a TP node advances only after all transactions in the prior epoch commit. Thus, transaction orders from outdated epochs can be safely discarded.

5 Evaluation

5.1 Evaluation Setup

All experiments were conducted on our cluster with 25 machines, each having a 2.60GHz 24-core E5-2690 CPU, 40Gbps NIC, and 64GB RAM. We ran each node in a container and used tc [40] to control the RTT among nodes. We set the epoch for garbage collection as 3s.

Baselines. We compared Perseus to three notable decoupled HTAP systems (TiDB [37], F1 Lightning [97], and Janus-HTAP [9]) and two TP system (Spanner-RSS [35] and CockroachDB [60]). The three HTAP systems all provide weaker consistency than Perseus. We chose them because they cover different design patterns on snapshot generation: Lightning leverages safe timestamps, TiDB uses a centralized timestamp service, and Janus-HTAP uses a dependency graph over two-phase-locking. We did not evaluate other HTAP systems because they either cannot support our benchmark (e.g., Hologres [43] supports only batched writes) or requires special hardware that is not available in a geo-distributed environment.

Spanner-RSS [35] is the only database providing RSS. In its replication layer, Spanner-RSS employs a Paxos-style consensus protocol to synchronize data from leader nodes to replicas. We used it to evaluate a strawman approach, which naively treats AP nodes as replicas of TP nodes and synchronously transfers updates between the nodes using Paxos [50]. CockroachDB [89] is another notable geo-distributed database that implements Raft consensus. Unlike Spanner-RSS, CockroachDB provides single-key linearizability (SKL), which is incomparable to RSS [35]. SKL guarantees strict real-time order for both reads and writes at key-level granularity. SKL benefits CockroachDB for resolving transaction conflicts at the primary nodes (i.e., leader). It is important to note that SKL can only be delivered when requests are served at the primary nodes. This is because CockroachDB relies on read requests to update the timestamp cache (which records the most recent time a key was read) in order to maintain linearizability for each key.

In the context of HTAP, serving all AP queries on primary nodes is undesirable for two key reasons. First, it couples the storage layer design of TP and AP, making it impossible to optimize the two data copies independently. Second, it causes TP and AP workloads to share the same computational resources, resulting in poor performance isolation (see §5.8).

Alternatively, CockroachDB enables stale snapshot reads on follower replicas. By leveraging loosely synchronized clocks, CockroachDB can offer bounded staleness reads, allowing users to specify an upper bound on the staleness of the data relative to the current time. However, this best-effort approach does not guarantee a response [45] and does not provide strong consistency (e.g., SKL). It provides snapshot isolation without ***a logical contract*** on staleness (e.g., assurances like "read-the-latest-write" or linearizability). Moreover, CockroachDB relies on *closed timestamp* to implement this follower read. A closed timestamp ensures that no new writes will be created by transactions with commit timestamps less than the closed timestamp. To ensure the closed timestamp advances regularly, CockroachDB sets a fixed interval and must reject long-running read-write transactions at primaries that exceed this interval. The default interval is set to 3 seconds,

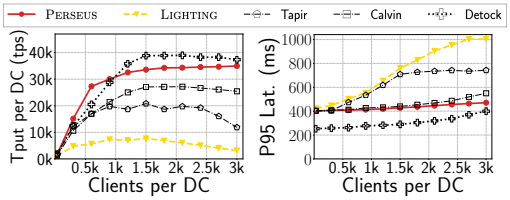


Fig. 8. TP perf. with increased clients.

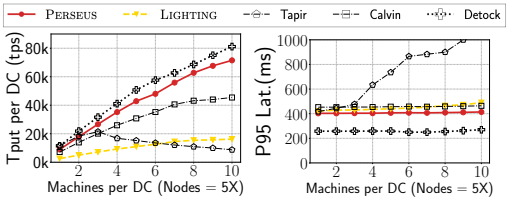


Fig. 9. TP perf. with increased nodes.

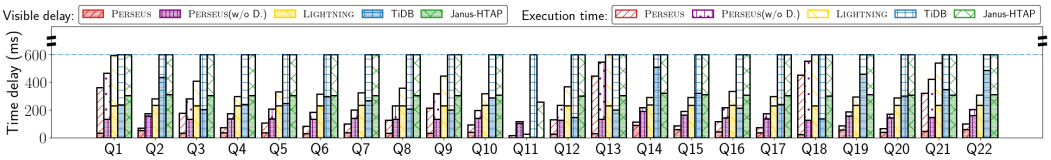


Fig. 10. AP performance on CH-benCHmark. We focus on visibility delay (VD) in our discussion.

which means that any bounded staleness reads with a constraint below this value will be rejected. We analyze the performance sensitivity of this interval in §5.10.

F1 LIGHTNING [97] is not open-source. We implemented its safe timestamps in our codebase using the same AP engine and updates transferring pipeline as PERSEUS. We set LIGHTNING's safe timestamp checking interval to 30*ms*: a lower value will not improve its freshness but degrade the performance. Janus-HTAP is also not open-sourced. We run Janus-HTAP by obtaining a code copy from its authors. We use the default parameters (e.g., 50*ms* for a batch interval) for experiments. We run TiDB using its open-source code [10] and tune the performance according to the official guidelines. We run Spanner-RSS and CockroachDB using their open-source code repos ([46, 71]).

All baselines and PERSEUS perform intra-data-center replication with a replication level of three. As discussed in §3.1, PERSEUS generally supports a flexible replication policy: TP nodes can be replicated either within a data center or across the data center. However, we do not perform geo-distributed replication in our evaluation since geo-distributed replication is orthogonal to our paper. Geo-distributed replication does not impact our freshness results, as visibility delay is evaluated after a transaction has been committed to all TP replicas. Furthermore, geo-replication's setup may favor the performance of PERSEUS and thus be considered unfair to other baselines. This is because geo-distributed replication lets PERSEUS hold a local TP replica of all TP nodes in each data center, and thus all AP queries in PERSEUS are local queries.

Workloads. We used three workloads in our evaluation.

CH-benCHmark is built from TPC-C [24] and TPC-H [23], widely used for evaluating HTAP [37, 81, 95]. We let AP clients query data based on their warehouse ID to emulate local queries.

HTAPBench contains all 6 types of read-write transactions in TPC-E and 2 new AP queries. TPC-E models a brokerage firm with customers issuing transactions related to trades. The 2 queries model business intelligence operations that access global data.

Microbench-Zipf. To test skewed workloads, we built microbench-Zipf. We populated each TP node with 10^6 key-value pairs and partitioned them into ten columns. We generate transactions based on YCSB-T. AP queries calculate the sum of a column.

Deployment. We set the intra-data-center RTT (between two nodes or from a client to a node) to 2*ms* and the cross-data-center RTT to 100*ms*. Our default deployment had 5 data centers and 100 TP shards. Each data center contained 20 TP shards distributed equally into 4 physical machines. Each data center also contained 5 DMs and 5 AP nodes allocated in one physical machine. Each DM and AP node subscribes to the TP nodes from one data center (i.e., 20 TP shards). As such, each data center has a full AP data copy. This eliminates geo-distributed join and task scheduling, which are orthogonal to our paper and do not impact the performance of snapshot generation.

5.2 Performance of TP Transactions

Although our paper primarily focuses on the snapshot generation mechanism of AP, the throughput and latency of TP directly affect the amount of data produced per second. This, in turn, squeezes the HTAP system's capacity to prepare snapshots. Therefore, we first compare the TP performance of Perseus with relevant HTAP baselines and other advanced TP protocols (i.e., Tapir [98], Calvin [91], and Detock [67]). These TP protocols also leverage deterministic ordering.

The results are shown in Figure 8 and Figure 9. We conducted experiments using CH-benCHmark with its default configuration. For HTAP systems, we used 40 AP clients per AP node to saturate AP throughput. For TP baselines, AP queries were disabled. Lightning, TiDB, and Janus-HTAP all use two-phase locking over two-phase commit for read-write transactions; thus, we report their performance with Lightning (implemented in our codebase) as a representative example. Notably, this concurrency control strategy is not specifically optimized for geo-distributed deployments, leading to relatively lower TP throughput in our experiments. In contrast, Perseus 's throughput scales with the number of TP clients and TP nodes. Compared to other advanced geo-distributed TP protocols, Perseus achieves comparable performance. Detock [67] achieves higher throughput than Perseus under the default setup (by up to 1.18×). However, in other configurations, we found Perseus can outperform Detock on read-intensive low-contention workloads by up to 1.7×. This is because Perseus induces more fast-path transactions under low-contention scenarios. A more comprehensive comparison of TP protocols is beyond the scope of this paper.

5.3 Performance of Local AP Queries

We evaluate local AP queries using CH-benCHmark. We spawned 1500 TP clients per data center and 40 AP clients per AP node to saturate both TP and AP throughput. Perseus achieves $\sim 35k$ transactions per second (tps) in each data center, which is comparable to other geo-distributed TP databases (e.g., [18, 67, 98], see §5.2). The AP performance (i.e., execution time) of Perseus is also better or comparable to baselines (Figure 10). The highlight is that Perseus provides a stronger consistency than baselines, while the visibility delay (VD) of Perseus on local queries was still 86.8%, 93.2%, and 90.1% lower. This is because Perseus's partial snapshots eliminated WAN messages, while baselines require global coordination.

We collect a breakdown to better understand the results (Table 3). Lightning has long staleness as its safe timestamp requires global communication to all TP nodes. Both TiDB and Janus-HTAP allow a centralized server to assign global orders to queries, resulting in long coordination time in a geo-distributed environment. TiDB's visibility delay came from two parts: (1) a WAN RTT to the global version manager ($127.7ms$); (2) waiting for all updates ordered before queries arrive ($103.6ms$). Janus-HTAP released a globally consistent snapshot by adding barriers ($121.8ms$) to transactions' orders and maintains a global map of the generated barriers. When a new query arrived, it went to the global server to fetch a consistent barrier cut ($127.5ms$), waiting for all transactions before the enforced barrier became visible ($53.7ms$). In Perseus, when a query arrived at the DM, the DM only calibrated the graph contacting with local TP nodes ($7.2ms$) and waited for the transactions in ADG to become visible locally ($14.9ms$). Overall, the latency overhead introduced by DM is minimal, as it is confined to the calibration of ADGs from local TP nodes. This is enabled by piggybacking geo-distributed indirect orders during transaction processing (§4.2).

Discussion. Perseus achieves much better freshness than baselines for local queries. Local queries are common in real workloads because TP nodes are partitioned by region (see §3.3). AP queries are considered local when a client conducts a regional analysis. With geo-distributed replication enabled, data is replicated globally. Updates originating from other regions can be replicated in the local data center, allowing more AP queries to be classified as local ones.

HTAP Sys.	Staleness	Snapshot Generation		Visibility Delay
		coordination	wait exe.	(Total)
LIGHTNING	237.2ms	0ms	0ms	237.2ms
TiDB	0ms	127.7ms	103.6ms	231.3ms
Janus-HTAP	121.8ms	127.5ms	53.7ms	303.0ms
PERSEUS	0ms	7.2ms	14.9ms	22.1ms

Table 3. Visibility delay breakdown of CH-benCHmark Q1. **"staleness"** is the time between updates of the snapshot are generated and the query arrives; **"coordination"** is the time spent to coordinate the query; **"wait exe."** is the blocking time waiting for updates.

HTAP Sys.	Staleness	Snapshot Generation		Visibility Delay
		coordination	wait exe.	(Total)
LIGHTNING	253.2 ms	0 ms	0 ms	253.2 ms
TiDB	0 ms	119.5 ms	221.8 ms	341.3 ms
Janus-HTAP	232.5 ms	148.1 ms	147.2 ms	527.8 ms
PERSEUS	0 ms	101.2 ms	101.2 ms	202.4 ms

Table 4. Visibility delay breakdown of HTAPBench MM.

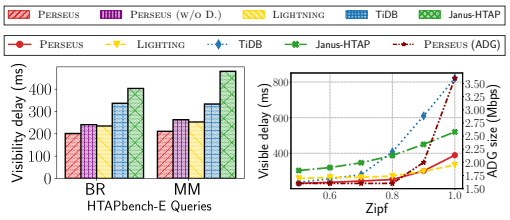
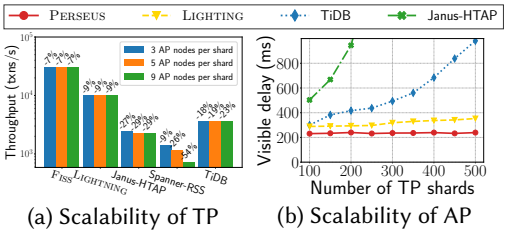


Fig. 11. HTAPBench

Fig. 12. Micro-zipf



(a) Scalability of TP (b) Scalability of AP

Fig. 13. Independent Scalability.

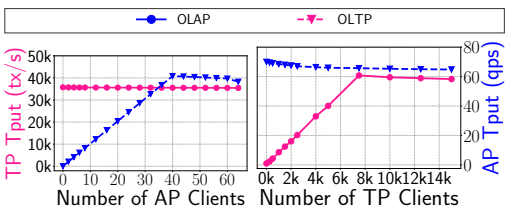
5.4 Performance of Geo-distributed AP Queries

We evaluated the performance of PERSEUS's geo-distributed queries using HTAPBench. The results are in Figure 11. The highlight is that PERSEUS ensured stronger consistency, while PERSEUS's visibility delay was still better than the baselines. The visibility delay of PERSEUS became larger than local queries because, in this experiment, AP queries required global snapshots instead of partial ones. In contrast, three baselines processed geo-distributed queries similarly to local ones. With more writes in the workload, the visibility delay of TiDB and Janus-HTAP increased slightly.

Table 4 shows the breakdown. For each query, PERSEUS took two cross-data-center network round-trips to serve it. When a query arrives at the coordinator, it sends the request to all relevant DMs. The DMs then calibrate their local ADG from TP nodes (cross-data-center). After receiving ADGs from DMs, the coordinator aggregates ADGs and generates a snapshot using the Dynamic Snapshot Algorithm. Then, PERSEUS sends ADGs to the AP nodes for execution. AP nodes wait for all needed data to become visible. Finally, the results were sent back to the coordinator. The latency overhead introduced by DM (for generating snapshot per query) is reflected in the snapshot generation time. LIGHTNING, which does not include additional DMs, has zero snapshot generation latency. However, this comes at the cost of significantly higher data staleness. When compared to others, PERSEUS 's snapshot generation latency is still low.

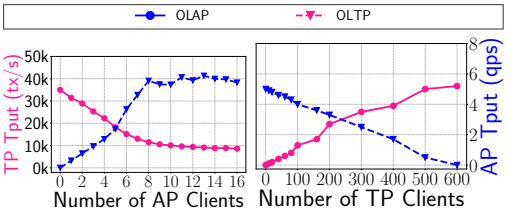
5.5 Performance on Skewed Workloads

In this experiment, we run Microbench-Zipf. As shown in Figure 12, TiDB's visibility delay increased dramatically with skewness because skewed updates are accumulated in the same Raft groups. The visibility delay of PERSEUS and Janus-HTAP also increased because the size of metadata (ADG in PERSEUS, batch dependency graph in Janus-HTAP) that was being maintained and exchanged grew with the amount of contention (the right y-axis in Figure 12). In contrast, LIGHTNING's visibility delay increased only slightly due to the overhead of merging more keys in the same AP nodes.



(a) Perf. isolation of TP (b) Perf. isolation of AP

Fig. 14. Perf. isolation of Perseus.



(a) Perf. isolation of TP (b) Perf. isolation of AP

Fig. 15. Perf. isolation of coupled design.

5.6 Cost Analysis of Perseus

Cost of transferring updates and ADGs. To investigate the overhead of ADGs, we first disabled the transmissions of updates and used typical dependency graphs for TP. Perseus achieved ~37k *tps* TP throughput. Then, we enabled the update transferring pipeline. The peak throughput of TP dropped to ~35k tps due to the cost of CPUs and networks. We then enabled ADGs, and the performance was maintained (~35k tps). This is as expected since the size of ADGs was small (i.e., ~32 bytes each transaction) in network packets. Moreover, Perseus's epoch-based tracing method (§4.5) prevents ADGs' size from becoming arbitrarily large. In our experiments, each node consumed at most 26.4 *Mbps* bandwidth for transferring ADGs. Furthermore, as shown Figure 12a, Perseus's total cost (7%) on TP was lower than TiDB's (18%) and Janus-HTAP's (27%) since Perseus did not add extra steps in TP. Perseus was comparable to Lightning (9%).

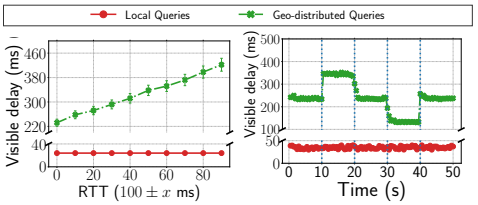
Cost of on-demand snapshots. We evaluate the overhead of assigning a snapshot per query in terms of query latency and computation cost on data nodes. The latency overhead is studied as the snapshot generation time in Table 3 and Table 4. Since snapshots only record the latest transaction IDs without data copying, the computation overhead is restricted to calculating ADGs and waitlist (Algo 3). Specifically, the computation cost on TP nodes has been majorly covered in transferring ADGs (i.e., 26.4*Mbps* on network bandwidth), along with a single process for responding to calibration requests in each node; On AP nodes, DMs spend roughly 7% of CPU cycles for unioning ADGs and forming the *waitlist*.

5.7 Effectiveness of Dynamic Snapshot.

We conducted an ablation study to evaluate the effectiveness of dynamic snapshots. In particular, we disabled the dynamic snapshot algorithm in our system variation: Perseus-w/o D. The variation lets AP queries observe updates in DM's local ADGs without pruning stragglers. The results are shown in Figure 10 and Figure 11. For local queries, the visibility delay of Perseus-w/o D. increased by 102.7*ms* on average, because the order of geo-distributed transactions is sent before their updates, which means it takes about a WAN round-trip time for these updates to show up in the local data center. Then, if Perseus-w/o D. collects their ADGs without pruning the formed *waitlist*, it causes significant wait time. In contrast, Perseus can incorporate data generated within the local data center (as long as it doesn't depend on geo-distributed transactions) and discard geo-distributed transactions that are not required by RSS, which helps reduce the waiting time. For geo-distributed queries, the visibility delay of Perseus-w/o D. increased by 40.8*ms* on average since it was limited by the slowest delivery of updates.

5.8 Performance Isolation and Scalability

Performance Isolation. Following [20, 81, 95], we let one kind of TP or AP clients saturate their throughput and see whether it sustains by adding the other kind of clients. We used CH-benCHmark for this experiment. First, we saturated TP and then added AP clients. As shown in Figure 14a,



(a) Fluctuation (b) Spike

Fig. 16. VD under network anomalies.

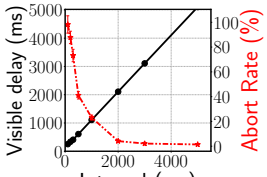


Fig. 17. Closed TS.

PERSEUS maintained a stable TP throughput. This is because adding AP workload only costs a few calibration messages (§4.2), which is off the critical path of TP. Second, we saturated AP and then added TP clients. As shown in Figure 14b, the AP throughput of PERSEUS dropped slightly (up to 12%) due to the cost of handling more updates. For comparison, we conduct a study on the strawm approach that issues both TP and AP requests to the primary TP nodes. This approach may naturally deliver strong consistency (e.g., strict serializability for PERSEUS or SKL for CockroachDB, see §5.1) by allowing TP and AP to share the same data copy, but it comes at the expense of sub-optimal performance and poor performance isolation. The results are shown in Figure 15.

Independent Scalability. To study the impact of scaling out AP on TP, we increased the number of AP nodes. We show the TP throughput drop in Figure 12a. PERSEUS's TP throughput downgrade was small and comparable to the other three HTAP systems. In contrast, the performance drop for Spanner-RSS was dramatic because it had to add more replicas in Paxos [48], affecting the critical path of transaction processing. To study the impact of scaling out TP on AP, we increased the number of TP nodes in each data center from 20 to 100 and added more AP nodes to keep each AP node subscribing to a fixed number of TP nodes (i.e., 20). Figure 12b shows the results. PERSEUS's visibility delay was stable. LIGHTNING's visibility delay increased slightly since it included more TP nodes to agree on a safe timestamp; TiDB's and Janus-HTAP's visibility delay increased proportionally because they relied on the global order to serve queries.

5.9 Robustness to Network Anomalies

We first set the cross-data-center RTTs to $100 \pm x$ ms with a uniform distribution and spawned TP and AP clients to reach peak throughput. As shown in Figure 16a, PERSEUS's visibility delay for local queries was stable because PERSEUS's ADG enabled PERSEUS to coordinate queries locally without being blocked by WAN messages. The visibility delay of geo-distributed queries increased roughly proportional to x. We then evaluated PERSEUS on abrupt changes in cross-data-center RTT. As shown in Figure 16b, we changed the RTT among data centers every 10s: we increased the cross-data-center RTT from 100ms to 150ms at 10s and back to 100ms at 20s; then, we changed it to 50ms at 30s and back to 100ms at 40s. The visibility delay of local queries was stable. The visibility delay of geo-distributed queries changed with network spikes.

5.10 A Case Study on Closed Timestamp

CockroachDB uses a closed timestamp mechanism to read data from replicas (§5.1). This approach lets users set the maximum time window for making snapshots on the primary nodes. A closed timestamp ensures that any transaction whose expected commit time is earlier than this timestamp **cannot** create new writes. As a result, the visibility delay (VD) of CockroachDB is highly related to this value. CockroachDB recommends 3s as the default value [89], which is significantly larger than the VD observed in §5.3 and §5.4. However, theoretically, a user can choose a smaller value if they want replicas to be fresher at the cost of TP performance (e.g., higher abort rate for unfinished writes). Because of such a trade-off, we don't consider CockroachDB as a regular baseline but evaluate it separately.

We run Ch-benCHmark. The results with different intervals are shown in Figure 17. If a significant short interval was used (i.e., frequently taking snapshots at primaries), the VD can be low. When the interval is set to 100*ms*, the visibility delay is around 150*ms*, with 50*ms* attributed to cross-region data transfer. However, the abort rate in this configuration is very high (98.26%), making the setup impractical. Overall, to avoid a significant impact on TP performance, the epoch length should be greater than 2000*ms*. Thus, the visibility delay of CockroachDB is higher than Perseus in practice.

6 Related Works

Order-before-Execution. Order-before-execution (a.k.a deterministic concurrency control) is widely-used for optimizing TP protocols [18, 30, 55, 65, 67, 72, 73, 78, 91, 98]. Conceptually, Perseus extends deterministic concurrency control from TP to HTAP, offering two key benefits. First, it allows AP nodes to learn the order of updates beforehand, enabling DMs and AP nodes to prepare snapshots in advance. Second, it facilitates ADG piggybacking by collecting essential snapshot information during the ordering phase without additional steps.

Note that several deterministic TP protocols also employ a component similar to DM to maintain serializability between transactions (e.g., scheduler in Calvin [90], batch metadata in Q-Store [72], and sequence managers in Caerus [36]). These protocols rely on exchanging orders between DMs to establish a global transaction order. In contrast, the DM in Perseus is specifically tailored for RSS and read-only AP queries. Since read-only requests do not generate data and RSS relaxes real-time ordering between readers, Perseus eliminates the need to maintain consistency among DMs (§4.2). Naively extending DMs in existing works to support AP queries may result in a tightly coupled concurrency control between TP and AP (discussed in the first category in §2.4).

Using Dependency Graph for Ordering. Dependency graphs are widely used in concurrency control. For example, BOHM [27] and PWV [29] construct dependency graphs from batches of input transactions, enabling parallel execution. Rococo [63], Janus [65], and Aria [55] use dependency graphs to track conflicts at runtime and reorder transactions to solve conflicts. However, these works have the limitation discussed in §3.2 when supporting AP queries. In contrast, Perseus introduces ADGs to achieve partial snapshots and detaches AP ordering from TP's critical path.

Snapshots Generation in TP. Some previous works [21, 30, 85, 86, 89] have studied how to efficiently generate snapshots for TP replicas. We classify them into two categories. The first category [21, 30] performs consensus or atomic broadcast among all TP replicas to prepare globally synchronized states. These methods have evolved to epoch-based or watermark-based approaches in HTAP (discussed in §2.4). The second category requires an additional voting stage between TP replicas to generate snapshots [85]. Perseus does not require voting rounds between AP replicas.

7 Conclusion

We present Perseus, a decoupled HTAP system with regular sequential serializability. Perseus generates on-demand partial snapshots for AP queries, decoupling concurrency control between TP and AP to achieve independent scalability. In addition, Perseus enhances data freshness for AP queries through dynamic snapshots.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and feedback. The work is supported in part by the National Key R&D Program of China (2022ZD0160201), HK RGC RIF (R7030-22), HK RGC GRF (Ref: 17208223 & 17204424), a Huawei flagship research grant in 2023, SupernetAI, and the HKU-CAS Joint Laboratory for Intelligent System Software.

References

[1] [n. d.]. Serializable ACID Transactions on Streaming Data. https://www.ververica.com/blog/serializable-acid-transactions-on-streaming-data.

[2] [n. d.]. Unlocking Real-time Data Synchronisation: A Guide to Change Data Capture (CDC) with Kafka and Debezium. https://medium.com/@sami.alashabi/unlocking-real-time-data-synchronisation-a-guide-to-change-data-capture-cdc-with-kafka-and-356c0ba083b7.

[3] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88.

[4] 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.

[5] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Proteus: Autonomous Adaptive Storage for Mixed Workloads. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. New York, NY, USA, 700–714. doi:10.1145/3514221.3517834

[6] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record* 24, 2 (1995), 23–34.

[7] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2017. Flowdb: Integrating stream processing and consistent state management. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 134–145.

[8] Vaibhav Arora, Ravi Kumar Suresh Babu, Sujaya Maiyya, Divyakant Agrawal, Amr El Abbadi, Zhiya Xue, Xun and, Zhujian Nan, and Feng. 2018. Dynamic Timestamp Allocation for Reducing Transaction Aborts. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 269–276. doi:10.1109/CLOUD.2018.00041 ISSN: 2159-6190.

[9] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2017. Janus: A hybrid scalable multi-representation cloud datastore. *IEEE Transactions on Knowledge and Data Engineering* 30, 4 (2017), 689–702.

[10] TiDB authors. [n. d.]. TiDB's github. https://github.com/pingcap/tidb.

[11] AWS. [n. d.]. Regions, Availability Zones, and Local Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[12] Azure. [n. d.]. Regions and availability zones. https://docs.microsoft.com/en-us/azure/availability-zones/az-overview.

[13] Ronald Barber, Matt Huras, Guy Lohman, C Mohan, Rene Mueller, Fatma Özcan, Hamid Pirahesh, Vijayshankar Raman, Richard Sidle, Oleg Sidorkin, et al. 2016. Wildfire: Concurrent blazing data ingest and analytics. In *Proceedings of the 2016 International Conference on Management of Data*. 2077–2080.

[14] Dennis Butterstein, Daniel Martin, Knut Stolze, Felix Beier, Jia Zhong, and Lingyun Wang. 2020. Replication at the speed of change: a fast, scalable replication solution for near real-time HTAP processing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3245–3257.

[15] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.

[16] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. Titant: Online real-time transaction fraud detection in ant financial. *arXiv preprint arXiv:1906.07407* (2019).

[17] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[18] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 210–227.

[19] Huawei Cloud. [n. d.]. Global Layout. https://www.huaweicloud.com/intl/en-us/about/global-infrastructure.html.

[20] 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* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 293–304. doi:10.1145/3030207.3030228

[21] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*.

[22] THE TRANSACTION PROCESSING COUNCIL. [n. d.]. TPC-E. http://www.tpc.org/tpce/.

[23] THE TRANSACTION PROCESSING COUNCIL. 2012. TPC-H. http://www.tpc.org/tpch/.

[24] THE TRANSACTION PROCESSING COUNCIL. 2014. TPC-C. http://www.tpc.org/tpcc/.

[25] Lei Deng, Jerry Gao, and Chandrasekar Vuppalapati. 2015. Building a Big Data Analytics Service Framework for Mobile Advertising and Marketing. In *First IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2015, Redwood City, CA, USA, March 30 - April 2, 2015*. IEEE Computer Society, 256–266.

[26] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+ T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 223–230.

[27] Jose M Faleiro and Daniel J Abadi. 2014. Rethinking serializable multiversion concurrency control. *arXiv preprint arXiv:1412.2324* (2014).

[28] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 613–624. doi:10.14778/3055540.3055553

[29] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[30] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment* 12 (2019), 1471–1484.

[31] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 492–528.

[32] Bryan Ford. 2019. Threshold logical clocks for asynchronous distributed coordination and consensus. *arXiv preprint arXiv:1907.07010* (2019).

[33] Gartner. [n. d.]. Real-time Insights and Decision Making using Hybrid Streaming, In-Memory Computing Analytics and Transaction Processing. https://www.gartner.com/imagesrv/media-products/pdf/Kx/KX-1-3CZ44RH.pdf.

[34] Google. 2022. AlloyDB for PostgreSQL under the hood: Columnar engine. https://cloud.google.com/blog/products/databases/alloydb-for-postgresql-columnar-engine.

[35] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 163–179.

[36] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2023. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proceedings of the VLDB Endowment* 17, 3 (2023), 469–482.

[37] 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.

[38] Yihe Huang, Hao Bai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2018. The Impact of Timestamp Granularity in Optimistic Concurrency Control. *arXiv preprint arXiv:1811.04967* (2018).

[39] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2022. Opportunities for optimism in contended main-memory multicore transactions. *The VLDB Journal* (2022), 1–23.

[40] Bert Hubert. [n. d.]. tc(8), Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[41] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'10)*.

[42] Jiaxin Jiang, Yuan Li, Bingsheng He, Bryan Hooi, Jia Chen, and Johan Kok Zhi Kang. 2022. Spade: A real-time fraud detection framework on evolving graphs. *Proceedings of the VLDB Endowment* 16, 3 (2022), 461–469.

[43] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, et al. 2020. Alibaba hologres: a cloud-native service for hybrid serving/analytical processing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3272–3284.

[44] Guoxin Kang, Lei Wang, Wanling Gao, Fei Tang, and Jianfeng Zhan. 2022. OLxPBench: Real-time, Semantically Consistent, and Domain-specific are Essential in Benchmarking, Designing, and Implementing HTAP Systems. *arXiv preprint arXiv:2203.16095* (2022).

[45] Cockroach Lab. [n. d.]. Follower Read. https://www.cockroachlabs.com/docs/stable/follower-reads.

[46] CockroachDB Lab. [n. d.]. Implementation of CockroachDB. https://github.com/cockroachdb/cockroach.

[47] Cockroach Lab. [n. d.]. Regional Tables. https://www.cockroachlabs.com/docs/stable/regional-tables.

[48] Leslie Lamport. [n. d.]. Paxos made simple. http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf.

[49] Leslie Lamport. 1986. On Interprocess Communication. *ACM Transactions on Programming Languages and Systems* 8, 1 (1986), 1–23. doi:10.1145/49602.49604

[50] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[51] 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.

[52] 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.

[53] 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.
[54] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 401–416.
[55] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. (2020).
[56] 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.
[57] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.* 7, 5 (Jan. 2014), 329–340. doi:10.14778/2732269.2732270
[58] 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.
[59] Yancan Mao, Jianjun Zhao, Shuhao Zhang, Haikun Liu, and Volker Markl. 2023. Morphstream: Adaptive scheduling for scalable transactional stream processing on multicores. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[60] Andrei Matei. 2019. CockroachDB's Consistency Model. https://www.cockroachlabs.com/blog/consistency-model/.
[61] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. 2015. S-store: Streaming meets transaction processing. *arXiv preprint arXiv:1503.01143* (2015).
[62] D.L. Mills. 1981. IEN173:Time Synchronization in DCNET Hosts. https://web.archive.org/web/19961230073104/http://www.cis.ohio-state.edu/htbin/ien/ien173.html.
[63] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 479–494. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/mu
[64] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. [n. d.]. Github: a stable commit of NYU-NEWS/Janus. https://github.com/NYU-NEWS/janus/tree/3c1b60cd965551b4bd3b1f3c475e16fdde2e4c2b.
[65] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 517–532.
[66] MySQL. 2022. MySQL Heatwave. https://dev.mysql.com/doc/heatwave/en/heatwave-introduction.html.
[67] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[68] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC '14)*.
[69] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.
[70] M Pezzini, D Feinberg, N Rayner, and R Edjlali. 2016. Real-time insights and decision making using hybrid streaming, in-memory computing analytics and transaction processing.
[71] Princeton-SNS. [n. d.]. Implementation of Spanner-RSS. https://github.com/princeton-sns/spanner-rss.
[72] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication.. In *EDBT*. 73–84.
[73] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.
[74] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.
[75] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.
[76] Jon TS Quah and M Sriganesh. 2008. Real-time credit card fraud detection using computational intelligence. *Expert systems with applications* 35, 4 (2008), 1721–1732.
[77] 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.

[78] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12 (2019), 1747–1761.

[79] 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).

[80] Harvard Bussiness School. [n. d.]. Dynamic Pricing: What It Is & Why It's Important. https://online.hbs.edu/blog/post/what-is-dynamic-pricing.

[81] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)(July 2021), USENIX Association.*

[82] Takamitsu Shioi, Takashi Kambayashi, Suguru Arakawa, Ryoji Kurosawa, Satoshi Hikida, and Haruo Yokota. 2022. Serializable HTAP with Abort-/Wait-free Snapshot Read. *arXiv preprint arXiv:2201.07993* (2022).

[83] Haoze Song, Xusheng Chen, Ruijie Gong, Zekai Sun, Tianxiang Shen, Cheng Li, Hao Feng, Sen Wang, and Heming Cui. [n. d.]. Perseus: Achieving Strong Consistency and High Data Freshness for Scalable Geo-distributed HTAP (Appendix). Supplemental Materials in ACM.

[84] Haoze Song, Wenchao Zhou, Feifei Li, Xiang Peng, and Heming Cui. 2023. Rethink query optimization in htap databases. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[85] A Sousa, A Correia, Francisco Moura, José Pereira, and Rui Oliveira. 2006. Evaluating certification protocols in the partial database state machine. In *First International Conference on Availability, Reliability and Security (ARES'06).* IEEE, 8–pp.

[86] António Sousa, Fernando Pedone, Rui Oliveira, and Francisco Moura. 2001. Partial replication in the database state machine. In *Proceedings IEEE International Symposium on Network Computing and Applications. NCA 2001.* IEEE, 298–309.

[87] Cloud Spanner. [n. d.]. New Spanner geo-partitioning improves performance and lowers costs. https://cloud.google.com/blog/products/databases/spanner-gets-geo-partitioning.

[88] Adriana Szekeres and Irene Zhang. 2018. Making consistency more consistent: A unified model for coherence, consistency and isolation. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data.* 1–8.

[89] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1493–1509.

[90] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12).* Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2213836.2213838

[91] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2014. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. In *SIGMOD '12: Proceedings of the 2012 ACM SIGMOD international conference on Management of data.*

[92] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* 18–32.

[93] Alejandro Vera-Baquero, Ricardo Colomo-Palacios, and Owen Molloy. 2016. Real-time business activity monitoring and analysis of process performance on big-data domains. *Telematics and Informatics* 33, 3 (2016), 793–807.

[94] Paolo Viotti and Marko Vukolić. 2016. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–34.

[95] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[96] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zhang. [n. d.]. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2021).*

[97] 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.

[98] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4 (Dec. 2018), 1–37. doi:10.1145/3269981

[99] Shuhao Zhang, Yingjun Wu, Feng Zhang, and Bingsheng He. 2020. Towards concurrent stateful stream processing on multicore processors. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 1537–1548.

[100] Jianjun Zhao, Yancan Mao, Zhonghao Yang, Haikun Liu, and Shuhao Zhang. 2025. Scalable Transactional Stream Processing on Multicore Processors. *IEEE Transactions on Knowledge and Data Engineering* (2025).

Received January 2025; revised April 2025; accepted May 2025