PERSEUS: Achieving Strong Consistency and High Scalability for Geo-distributed HTAP

Haoze Song[†], Xusheng Chen[†§], Ruijie Gong[†], Tianxiang Shen[†], Ji Qi[†], Cheng Li[‡],
Hao Feng[§], Sen Wang[§], Gong Zhang[§] and Heming Cui[†]
[†]The University of Hong Kong, [‡]University of Science and Technology of China, [§]Huawei Technologies Co., Ltd.
[†]{hzsong, xschen, rjgong, txshen2, jqi, heming}@cs.hku.hk,
[‡]{chengli7}@ustc.edu.cn, [§]{feng.hao1, wangsen31, nicholas.zhang}@huawei.com

***Abstract*—The emergence of global data-driven applications (e.g., business intelligence) makes geo-distributed hybrid transactional and analytical processing (HTAP) databases highly desirable. Existing distributed HTAP systems provide users with good performance on both transactions and analytical queries, and this good performance is scalable to the number of storage nodes. Unfortunately, these systems provide only weak consistency: some analytical queries may work on stale data generated by transactions, impairing applications' quality and correctness.**

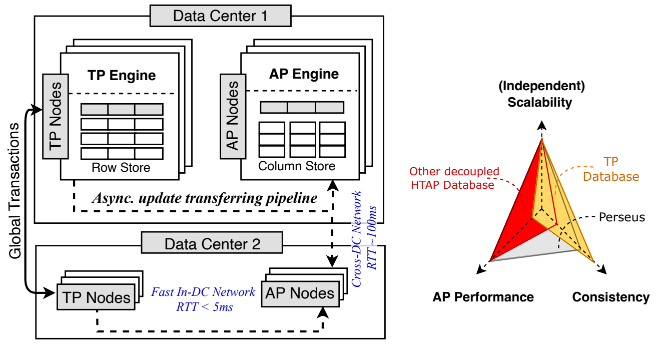
In this paper, we present PERSEUS, a scalable HTAP database that enforces strong consistency for both transactions and analytical queries. To handle consistency efficiently, PERSEUS features a new data structure (called augmented dependency graph) that explicitly records the versions of data, implying which data need to be read together in a consistent snapshot. In addition, PERSEUS presents a new dynamic snapshot algorithm to minimize data staleness on geo-distributed analytical queries.

Extensive evaluation results show that, compared to the latest HTAP databases with even weaker consistency (e.g., snapshot isolation), PERSEUS achieves up to 86.8%~93.2% **lower visibility delay, which is an important metric of data freshness, capturing the time interval during which transactional updates to the database can be visible to analytical queries; PERSEUS is scalable to many nodes and robust to network instability.**

I. INTRODUCTION

Cloud providers make it easier to deploy online transaction processing (TP) applications across geo-distributed near-client data centers for low data access latency, elastic scalability, and service localization [12], [13], [21], [22]. Recently, the emergence of real-time decision-making applications (e.g., business intelligence [29], [89] and fraud detection [17], [77]) desires 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.

Existing HTAP databases typically consist of a TP engine executing global clients' transactions (continuously updating the database's state) and an AP engine executing analytical queries. To handle mixed workloads efficiently, a popular category of distributed HTAP databases (e.g., [10], [19], [40], [42], [56], [59], [72], [91], [93]) independently optimizes two data copies for TP and AP (e.g., using row-oriented storage for TP and column-oriented storage for AP), use an *update transferring pipeline* to capture TP's updates asynchronously and timely apply them to AP's individual storage nodes. We show an example in Figure 1a.



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

Fig. 1: (a) shows a deployment model of geo-distributed HTAP databases (b) compares the performance of TP, PERSEUS, and other decoupled HTAP: PERSEUS strikes a practical design point, which should be desirable for global decision-making applications.

This decoupled design has unique strengths in achieving isolation between TP and AP, and such isolation is especially desirable due to two aspects. First, it empowers the HTAP systems to provide independent computation resources for TP and AP. Systems can isolate the performance between different types of workloads, which is studied as *performance isolation* in the previous papers [16], [42], [63], [82]. Second, it enables TP and AP to scale out independently: good TP and AP performance can be both maintained with an increasing number of either TP or AP nodes, which we term as ***independent scalability***. It is important because cloud providers make TP and AP nodes scale in and scale out independently as a decent practice: customers can independently scale out TP or AP nodes to cater to workload increments and scale down to save costs. As a result, decoupled HTAP systems have become popular, and our paper will focus on such a design.

We define strong consistency in HTAP as the whole system operating as a single system for providing consistent external views [25], [84] for both transactions and AP queries. Specifically, the system should ensure not only strong isolation (e.g., serializability) but also provide regularity [90] for both TP and AP: AP queries can see all TP writes that are finished before the AP query starts in real-time order.

Strong consistency has been advocated by many trending HTAP applications (e.g., [17], [75], [76]) because these applications may use the results of analytical queries to conduct real-time decisions: clients may use the results of AP queries as the payloads of TP transactions [38], [39], [49]. Instead of submitting these analytical queries to a stand-alone TP system, HTAP makes it possible to process such analytical queries on independently optimized columnar storage and AP engine in favor of performance (e.g., ×7 ~ ×10 speed-ups [4], [5]). We show a comparison of TP and HTAP in Figure 1b.

Unfortunately, to the best of our knowledge, ensuring ***strong consistency*** is still an open problem for any decoupled HTAP databases. Therefore, our paper aims to build a geo-distributed scalable HTAP system with as strong as possible consistency, which should be powerful enough to support these trending global decision-making applications.

We present PERSEUS[1], an HTAP system that provides regular sequential serializability (RSS). RSS is a strong consistency model originally tailored for read-only transactions [41]. We extend its definition from TP to HTAP by incorporating AP queries as read-only transactions. Specifically, RSS requires all read operations to observe a consistent snapshot that reflects the most recently finished updates and preserves causality. RSS enforces all but only a tiny subset of ordering requirements in a strictly serializable manner. The tiny subset of ordering requirements forgo by PERSEUS contains only those operations caused by the order transitivity (through a read-only transaction seeing writes of a concurrent transaction). We formally define RSS in Section II-B and discuss its desirability in Section II-A.

PERSEUS tackles two unique research challenges on 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 takes a novel data structure in the concurrency control layer to transmit transaction order with minimal performance perturbation on TP, which we term Augmented Dependency Graph (ADG). Briefly, ADG stitches each transaction update with metadata, indicating which updates must be read together. ADG contains fine-grained global dependency information, enabling the tracing of dependent transactions efficiently. Such a design also enables PERSEUS to ensure independent scalability, as now the task for ordering AP query is detached from the critical path of ordering transactions (to be illustrated in Section II-C).

The second challenge is to achieve low staleness for AP queries, where the data transfer delays for different updates can be large and different (e.g., either cross-data-center or in-data-center). To address this challenge, PERSEUS carries a new dynamic snapshot algorithm with two features: (1) it estimates the delivery of updates according to the recent network statistics (2) it selects the near-optimal snapshot that ensures both RSS and low data staleness. The tricky is that almost all consistency models (including RSS) let a system freely determine whether a *concurrent transaction* should be observed in the snapshot, as long as the external view of the HTAP databases is monotonic. As such, PERSEUS proactively incorporates concurrent transactions committed after the query starts for better freshness (Section IV-B).

Implementation. We implemented PERSEUS based on a TP codebase [70]. We prove that PERSEUS satisfies RSS and independent scalability in our technical report [32]. Leveraging four standard benchmarks (TPC-C [28], TPC-H [27], TPC-E [26], and YCSB-T [30]), we implemented three HTAP workloads that highly desire HTAP, including the widely used CH-benCHmark [24] and two new ones. We compared PERSEUS to the three latest decoupled HTAP systems (TiDB [42], Lightning [93], Janus-HTAP [10]) and a TP system (Spanner-RSS [41]). Among these HTAP systems, only PERSEUS provides strong consistency (RSS), while others provide at most serializable snapshot isolation [35].

Contribution. Our main contribution is PERSEUS, a geo-distributed decoupled HTAP database that achieves as strong as possible consistency. Our evaluation shows:

- PERSEUS achieves high TP and AP performance.
- PERSEUS achieves low staleness for AP queries. PERSEUS's visibility delay (a metric of staleness, which captures the time interval during which transactional updates to the database can be visible to queries) is up to 86.8%, 93.2%, 90.1% lower than LIGHTNING, TiDB, and Janus-HTAP.
- PERSEUS's TP and AP performance are independently scalable. PERSEUS's good TP performance was maintained with an increasing number of AP nodes; in turn, AP performance was maintained with an increasing number of TP nodes.

The rest of the paper is organized as follows. Section II discusses the background of RSS, decoupled HTAP, and data freshness. Section III provides an overview of system components. Section IV details the design of how ADG and dynamic snapshot algorithm work. Section V evaluates the performance of PERSEUS. Section VI concludes the paper.

II. BACKGROUND AND MOTIVATION

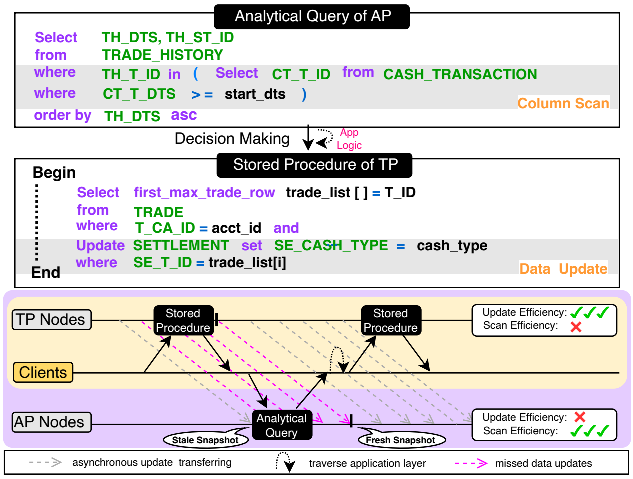
In this section, we first provide a background of consistency models and illustrate the rationale for using regular sequential serializability in PERSEUS. We then discuss the concurrency control mechanism of the existing works. Finally, we introduce data freshness and its performance indicator: visibility delay.

A. Consistency Models

Consistency Models in HTAP. Consistency models play a crucial role in determining the correctness and performance of an HTAP application. Essentially, a consistency model is a contract between databases and their clients, which specifies the acceptable return values for a set of operations. Applications prioritize stronger consistency so that the databases have limited possible return values, which makes it easier for programmers to build an accurate application on top of them.

In HTAP, enterprises utilize vast amounts of data to support their organizational needs, weaving transactions and analytical queries together. Practically, almost all workloads can be split into transactions and analytical queries. For example, TPC-E [26] (which is known as a TP workload) has a mix of row-oriented transactions and column-oriented AP queries. See the

[1] PERSEUS was a valiant, intelligent, and dexterous hero of Greek mythology. We use it to signify that geo-distributed HTAP is difficult but can be efficient and consistent by careful designs.



```
Select    TH_DTS, TH_ST_ID
from      TRADE_HISTORY
where     TH_T_ID in  (  Select  CT_T_ID  from  CASH_TRANSACTION
where     CT_T_DTS   > =   start_dts   )
order by  TH_DTS  asc
```

```
Begin
      Select  first_max_trade_row  trade_list [ ] = T_ID
      from    TRADE
      where   T_CA_ID = acct_id  and
      Update  SETTLEMENT  set  SE_CASH_TYPE  =  cash_type
      where   SE_T_ID = trade_list[i]
End
```

Fig. 2: This diagram shows a simplified AP query and TP transaction (i.e., a stored procedure) from TPC-E [26]. Executing transactions on TP nodes and executing analytical queries on AP nodes can maximize the execution performance. However, sn HTAP system with weak consistency (e.g., snapshot isolation) may read stale snapshots and then determine the wrong payload of the stored procedure.

example in Figure 2: the query operation fetches the latest trade history from the TRADE_HISTORY table and guides the decisions on security checks. Compared to submitting these analytical queries to a TP system, HTAP can speed up analytical queries, favoring performance (see Figure 1b).

As TP transactions and AP queries are integral aspects of workloads, our paper considers the HTAP database as a whole system that provides consistency guarantees.

Why decoupled HTAP? The contradiction between the poor AP performance of TP databases and the poor consistency of AP databases (e.g., using stale data) motivates us to design a decoupled HTAP database with as strong as possible consistency. First, the decoupled architecture provides AP nodes the flexibility to optimize AP performance and lets the integrated HTAP database scale out independently. Second, strong consistency satisfies the needs and simplifies the development of HTAP applications. We refer to HTAP applications as those applications where transactions and AP queries ***cannot*** be simply separated into two non-intersecting workloads. Otherwise, the workloads can be efficiently handled by traditional AP databases with ETL [1], [3], and thus are not specific in HTAP.

Why RSS? There are many legacy consistency models in the market. We introduce the popular ones. It should be noted that isolation levels and consistency models have been blurred for decades. For example, snapshot isolation, which essentially provides a constraint on isolation (i.e., the isolation in ACID), is frequently referred to as a consistency model. For simplicity, we use the term "snapshot isolation" for the consistency models without any time constraint. Same as the previous papers [2], [90], we say a consistency model is strong if it provides regularity (i.e., single order and real-time writes).

Figure 2 shows an anomaly when using snapshot isolation. The AP query may miss important updates from TP transactions (i.e., the pink arrows in the diagram). Thus, the clients may use the wrong payload for the next TP transaction, which is issued based on the results of the AP query. Such anomalies can be common in fraud detection [17] and real-time pricing applications [37]. Using snapshot isolation, they require a complex application-layer logic for verifying the results.

	Notations	Description
Operations in HTAP	R	A read-only transaction.
	W	A read-write transaction.
	Q	An read-only AP query.
	O	Any operations among R, W, and Q.
Order Relations	$O_1 \otimes O_2$	O_1 *conflicts* with O_2: they access the same data and one of them is W.
	$O_1 \rightarrow O_2$	O_1 happens before O_2 according to a global wall clock.
	$O_1 \rightsquigarrow O_2$	O_2 casually depends on O_1: e.g., the order posted by the client session.
	$O_1 \lessdot O_2$	O_1 orders before O_2 in the global historical log.

TABLE I: Notations for consistency models.

Another popular category of consistency models is derived from serializability (i.e., an isolation model), and thus, all provide serializable isolations. For example, strict serializability (SS) provides an abstract that appears to execute transactions sequentially, one at a time, in an order consistent with the transactions' real-time order; Serializable snapshot isolation (SSI) requires transaction executions that are equivalent to *some* serial schedule on a single data copy (a snapshot), while SSI does not essentially restrict the set of allowed serial schedules (i.e., without "real-time order"); Regular sequential serializability (RSS) further enhances the requirement on the available schedules of SSI. It preserves casualty and regularity for all operations, as well as the real-time order for writes.

Table II shows the comparison of the three representative serializable consistency models. We list four ordering invariants for comparison. We define the needed notations in Table I. Intuitively, I_1 indicates serializability; I_2 indicates the "read-the-latest-writes" property; I_3 indicates causality; I_4 indicates real-time order between read operations and queries.

Overall, SS is the strongest model. However, we do not use SS since it is costly to achieve in geo-distributed HTAP [41], [62]. We discuss the problem of SS in our technical report [32]. 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 transactions (e.g., by synchronously contacting all TP nodes).

Compared to SS, the invariant that RSS forgoes is that a read-only operation from a client may see the updates that are temporarily not visible to other read-only operations from other clients (i.e., I_4). It can be compliant with the correctness of decision-making applications because the anomalies only happen between read-only operations without involving updates (e.g., read-write transactions).

B. Regular Sequential Serializability

We now formally define RSS in HTAP using the notations in Table I. Intuitively, we extend RSS from [41] to HTAP systems by regarding queries as read-only operations.

Ordering Invariants	**SS** [73]	**RSS** [41]	**SSI** [48]	**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 II: Representative ordering requirements of different consistency levels. SSI is weaker than RSS and provides no real-time properties.

Definition (**View Equivalence**). *Two schedules S_1 and S_2 are view-equivalent if they produce the same set of results when executed against the same database state. S1 is view-equivalent to S2 if and only if: The order of any two conflicting operations in S1 is the same as the order of those operations in S2.*

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

C. Decoupled HTAP Databases

Many decoupled HTAP databases have been proposed [10], [15], [42], [59], [67], [78], [80], [82], [83], [93]. However, they either cannot ensure strong consistency or have to sacrifice performance and scalability when deployed in a geo-distributed environment. We classify them into two categories based on their concurrency control mechanisms for AP queries.

The first category uses the centralized timestamp to coordinate TP transactions and AP queries. These systems have the potential to provide strong consistency at the cost of scalability and freshness. For instance, TiDB [42] and HANA ATR [59] use global timestamps (counters) for all transactions and queries. Suppose they want to achieve regularity for AP queries (i.e., promising AP queries can see all TP writes that are finished before the query starts). In that case, a query must travel to the global counter to retrieve the latest timestamp, which is a single-node bottleneck. Worse still, AP queries should 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 TP transactions, even if the AP query does not read the data generated by the transaction, leading to bad data freshness. See our evaluation in Section V-B.

Advanced timestamp mechanisms [6], [9], [20], [34], [36], [43], [44], [66], [92] (e.g., hybrid logical clock) may overcome the limitation of centralized order assignment. However, to the best of our knowledge, they are not well-customized for HTAP databases and can incur performance problems (e.g., tagging timestamps for each key in the column store can be costly because compaction and single instruction multiple data (SIMD) are common optimizations in AP nodes).

The second category periodically generates consistent snapshots for AP queries and can not provide strong consistency (e.g., regularity) for AP queries. For instance, Google F1 lightning [93] 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. F1 lightning provides snapshot isolation for AP queries. The snapshot can be consistent but stale. Vegito [82] adopts a gossip-style scheme to allocate consistent epoch numbers for dependent transactions. Each AP query is assigned an epoch ID and queries data on a stale snapshot with the same epoch ID. When deployed in a geo-distributed environment, besides the loss of regularity for AP, it also incurs bad data freshness since the time for gossiping a globally legal epoch ID can be costly. Janus-HTAP [10] features a batch graph for solving consistency issues between TP and AP. Transactions are firstly grouped into batches, and AP queries are required to find a set of cuts in the batch graph to retrieve data in AP nodes.

Different from these works, PERSEUS generates an *on-demand partial snapshot*: a snapshot contains only relevant keys accessed by the AP queries but captures all ordering requirements. Compared to global snapshots that contain the state of the whole database (e.g., the mechanisms in the first category), partial snapshots discard those unrelated slow ongoing TP transactions, thus reducing the time for snapshot preparation. Compared to periodical snapshots (e.g., the mechanisms in the second category), on-demand snapshots provide regularity by capturing the latest writes in real-time order.

To do so, PERSEUS proactively records transaction dependency during TP and attaches each update in an augmented dependency graph (ADG), indicating which updates of other keys must be read together. ADG contains fine-grained global dependency information and enables the on-demand tracing of transactions (to be illustrated in Section III-B).

Besides these decoupled HTAP databases, there are many coupled HTAP databases. They are designed for a single node or in-data-center deployments [16], [18], [50]–[52], [60], [61], [74], [81], [83], and thus orthogonal to our paper.

D. Data Freshness and Visibility Delay

Information is most valuable when it first appears [7], [57]. For that reason, data freshness is commonly mentioned as an essential design goal of HTAP databases [49], [63].

To gauge freshness, a metric should quantify how recent the snapshot of TP is seen by each AP query. We use visibility delay for the performance indicator, which is also studied in the previous papers [19], [42], [82], [91]. The previous works measured visibility delay for each update. They used the average number for the visibility delay of the system, which is suitable for comparing the performance of the data transferring service. However, our paper targets the on-demand freshness optimization for each AP query. Thus, we define the visibility delay from the AP queries' perspective, which is slightly different.

Visibility delay of an AP query is defined as the time interval during which transactional updates to the database can be visible to the query. Specifically, we assume TP nodes have generated a series of data versions by performing new read-write transactions (each transaction for a new data version): V_0, V_1, ..., V_t. Thus, V_t should be the data version that includes the most recent update. We then assum AP nodes see the versions V_0, V_1, ..., V_a, where $V_a \leq V_t$. When a query Q arrives at AP nodes, it uses V_q for execution, where $V_0 \leq V_q \leq V_t$. Therefore, 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).

In practice, we evaluate the visibility delay of an AP query by calculating the difference between the timestamp of the query that begins to execute and the timestamp of the used snapshot (i.e., the commit timestamp of the latest transaction in the snapshot). It consists of the time for determining V_q (i.e., the coordination time for AP queries) and the time for V_q to become visible at the AP nodes.

Freshness Optimization for Local Queries. The demand for querying on local or regional data is typical in geo-distributed HTAP applications. For instance, the regional managers of a multinational company (e.g., [8], [31]) mostly query local data inside their region to provide better-localized services (e.g., pricing [68], trading tracking [85]). However, we find that most of the HTAP systems neglect the data freshness optimization for local queries. As discussed in Section II-C, they rely on global timestamps or global snapshots to provide a consistent view for even local queries. We make freshness optimizations for local queries in PERSEUS by generating partial snapshots for local queries using ADG, which eliminates the cost of preparing global snapshots. PERSEUS is a unified system for both queries with and without locality. PERSEUS provides stronger consistency than baselines for both two types of queries and achieves better data freshness for local queries (see our evaluation in Section V-B).

III. PERSEUS OVERVIEW

Recall that PERSEUS is designed for geo-distributed HTAP in which cross-datacenter communication can be costly. Our goal is to optimize the cost for an AP query to obtain a globally consistent snapshot and achieve as strong as possible consistency without sacrificing (independent) scalability.

We first clarify the system, deployment, transaction, and query models of PERSEUS. We then introduce its dependency managers and ADGs. Finally, we provide an overview of system designs.

A. Architecture

System Model. PERSEUS consists of a group of *TP nodes* and a group of *AP nodes*. As shown in Figure 1a, TP nodes have a row-oriented data copy of the database in which data are assigned into partitions. Each TP node holds a specific database partition, and the partition can be replicated to provide high availability. We omit TP replicas for simplicity.

To serve client requests, TP nodes run a distributed concurrency control protocol and update the database. Meanwhile, updates are asynchronously shipped to the AP nodes. AP nodes have a column-oriented data copy, and the data partition in AP nodes can be different from TP nodes (a.k.a. asymmetric partition [65]). Each AP node can subscribe to updates from a set of TP nodes, grouping frequently accessed data together.

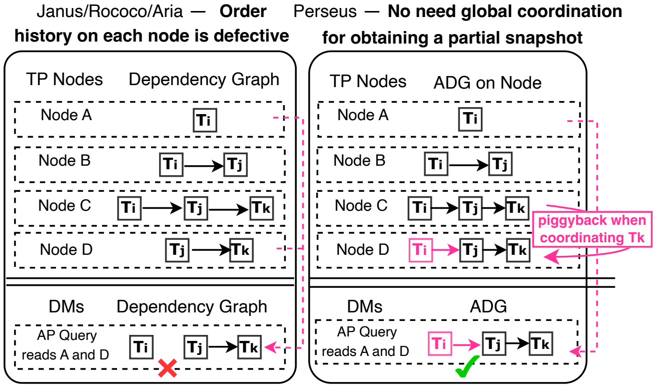


Fig. 3: We assume three transactions in the example: T_i access nodes A and B; T_j access nodes B and C; T_k access nodes C and D. TP nodes execute transactions in the serializable order $T_i \prec T_j \prec T_k$. Typical dependency graph records only direct dependencies between transactions, and thus, a DM for nodes A and D cannot obtain a complete order history based on the records on TP nodes A and D. 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.

Nodes don't have synchronized clocks. Networks between nodes are asynchronous: nodes should communicate with each other using asynchronous messages. Messages can be either dropped or reordered but promise an eventual delivery.

Deployment Model. We consider a typical geo-distributed deployment (see Figure 1a). TP and AP nodes are divided into multiple data centers. AP nodes can subscribe to TP nodes, either in-datacenter or cross-datacenter. Cross-datacenter communication is costly and unstable (e.g., $\sim 100ms$ RTT), while the in-datacenter network is fast (e.g., $< 5ms$ RTT) and has a high bandwidth (e.g., 10 Gbps [12]).

Transaction Model & Query Model. PERSEUS supports geo-distributed transactions. In our prototype, transactions are written as stored procedures for performance and maintainability.

PERSEUS supports AP queries with and without locality. Formally, we define *local query* as the AP query access data belonging to the TP nodes from a single datacenter; otherwise, we term the AP query as *geo-distributed query*. To optimize the performance of local query (i.e., obtaining a partial snapshot without global coordination), PERSEUS requires prior knowledge of which data that is *potentially* touched by Q. PERSEUS requires such information only in the granularity of columns rather than specific keys, which is easy to detect in SQL and can be automatically achieved with existing tools [33], [86]. For interactive or extra-complex queries, PERSEUS can mark all columns as potentially touched and generate a global snapshot for such queries (i.e., falling back to the baseline methods).

B. Dependency Managers and Augmented Dependency Graph

In addition to TP and AP nodes, PERSEUS has another crucial component: *Dependency managers (**DMs**)*. In PERSEUS, DMs are responsible for collecting order history and creating consistent snapshots. We deploy AP nodes and DMs in pairs.

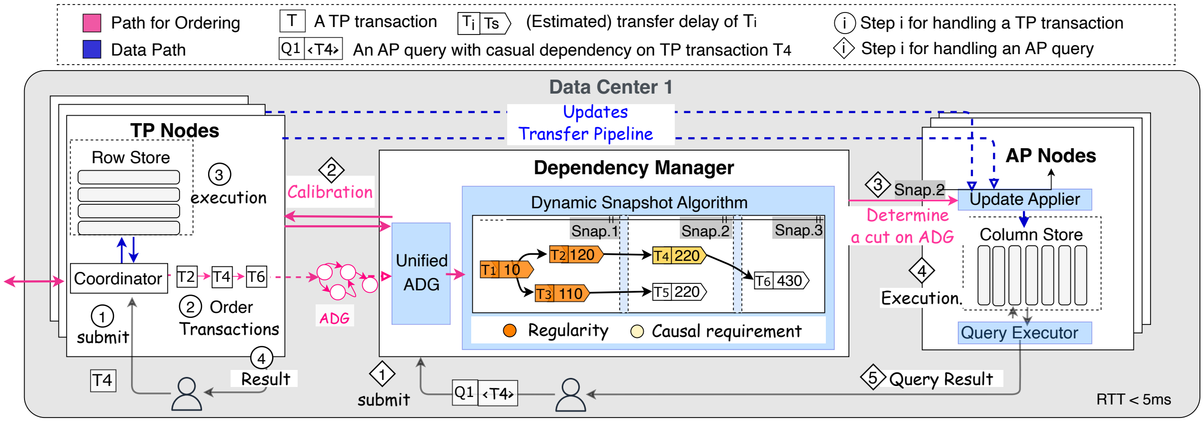


Fig. 4: PERSEUS has three key components: TP nodes serve transactions and generate ADG to transfer order information from TP to AP; Dependency Managers (DMs) track ADG from TP nodes and generate RSS snapshot (a consistent cut on the combined ADG) with high freshness; AP nodes timely absorb asynchronously transferred updates from TP nodes and serve queries based on the given snapshot.

Each AP node has a DM colocated with it. Each AP node and DM pair handles a subset of TP nodes (partitions).

Specifically, DMs use an augmented dependency graph (**ADG**) for carrying order information and managing partial snapshots. ADG is a serialization graph based on the transaction execution history. Each vertex in the graph corresponds to a transaction. Each edge represents a dependency between two transactions due to conflicting data access. Compared to typical dependency graphs that are commonly used in TP databases (e.g., Janus [71], Rococo [69], and Aria [64]), ADGs are designed for HTAP: it carries sufficient order information for each node to construct a serializable history (e.g., the history for a partial snapshot) without global coordination.

Technically, typical dependency graphs only record each transaction's direct dependent information, while ADG additionally collects indirect but essential ordering information by piggybacking them during the transaction execution.

TP databases do not need such order information because all transactions in TP share the same data copies (including replicas). In this way, when a new transaction is issued and wants to find a valid snapshot, it contacts all relevant TP nodes (i.e., the TP nodes have the data accessed). All data conflicts and defective orders can be solved and detected on the contacted TP nodes (as proved in [64], [71]).

However, in HTAP, AP queries read data directly from AP nodes (i.e., an asynchronous data copy) and do not participate in the concurrency control of TP. Instead of letting AP nodes actively ask all TP nodes to obtain a complete order history (i.e., generating a global snapshot by global coordination), PERSEUS augments its dependency graph to proactively carry and spread order history among nodes. Thus, ADG can enable DMs to observe all RSS ordering requirements at a low cost.

We show a comparison between typical dependency graphs and ADGs in Figure 3. In this example, we assume TP nodes execute transactions in the serializable order $T_i \prec T_j \prec T_k$. Using the typical dependency graph, a DM handles TP nodes (partitions) A and D and cannot capture the ordering requirement between T_i and T_k based on dependency from TP partitions A and D. Therefore, when an AP query wants to read data from A and D at AP nodes, the order history is defective, leading to observing an in-consistent partial snapshot. ADG handles such an issue by recording indirect dependency and piggybacked the dependency when coordinating T_k.

We prove ADG is sufficient for consistent partial snapshots and capturing all ordering requirements of RSS in our technical report [32]. Intuitively, a partial snapshot should know the latest transaction in the dependency graph (e.g., T_k), and its dependency should be complete since all predecessors are ordered by TP nodes, and the orders are piggybacked.

We show that ADG only introduces small and acceptable overhead for piggybacking ordering information in our evaluation (see Section V-B).

C. System Overview

Using DMs and ADG, PERSEUS provides partial snapshots for AP queries (both local and geo-distributed). We now provide a system overview of how PERSEUS enforces RSS and optimizes data freshness based on these generated partial snapshots.

System Workflow. Figure 4 shows the system architecture of PERSEUS, as well as the workflow of TP and AP inside a data center. For TP transactions, clients submit the payloads to a nearby TP node and treat it as the coordinator. Then, the coordinator solves the conflicts between transactions by reordering them and guarantees that the ADGs on each TP node are acyclic to enforce isolation and consistency (i.e., the same way used in [64], [69], [71]). After that, an execution order is determined among TP nodes, and transactions are executed according to the given order.

As long as the order is determined, order information (i.e., ADG) is also asynchronously transferred from TP nodes to DMs. After execution, new data is asynchronously transferred from TP nodes to AP nodes using the updates transferring pipeline. DMs proactively track ADG from TP nodes and generate *partial snapshot*, which is essentially a consistent cut on the combined ADG without actual data. AP nodes timely absorb updates from TP nodes and merge them into the AP-optimized column stores.

For an AP query, clients submit it to a DM (colocated with

AP nodes). The DM calibrates ADG with TP nodes to enforce the "read-the-latest-write" property of RSS. To optimize data freshness, the DM runs a dynamic snapshot algorithm to cut the ADGs for on-demand snapshots while keeping RSS intact (to be illustrated later). Given the snapshots, AP nodes can now execute the AP query and return the results to the clients.

On-demand Partial Snapshot and RSS (Section IV-A). For a query Q, PERSEUS enforces RSS on the snapshot recorded by ADG. Basically, the partial snapshot that guarantees RSS should obey all RSS's ordering requirements, including those requirements by transitivity through un-accessed data. There should exist a global snapshot of the whole database that satisfies RSS and is view-equivalent to this partial snapshot.

PERSEUS achieves this in three aspects. First, ADGs guarantee the partial snapshots are serializable. Second, PERSEUS captures the latest reads by calibration. Third, PERSEUS carries casual dependencies (e.g., through message passing) during snapshot generation. We formally prove that PERSEUS satisfies RSS in [32]. As an optimization for HTAP, we let the client mark the dependency between TP transactions and AP queries from different client sessions, and thus, the client can issue transactions and queries in an open loop (i.e., the client does not need to wait for the return values of the transaction before issues the next dependent AP query).

Dynamic Snapshot Algorithm (Section IV-B). As introduced in Section I, PERSEUS proposes a new dynamic snapshot algorithm to optimize data freshness. Specifically, RSS requires a query to see all transactions finished before the query starts. After ensuring RSS, PERSEUS can still freely choose which concurrent transactions (and their updates) should be included in the snapshot. However, there is a dilemma: incorporating more updates tends to improve data freshness, but it may also involve long waits if these transactions' or their predecessors' updates are shipped from faraway sites or blocked by the network. In this way, the data freshness also decays as time goes on.

To address this dilemma, PERSEUS annotates each transaction on the ADG with its estimated available time and prunes stragglers branches that are not necessarily required by RSS. We show a simple example in Figure 4 and detail the algorithm in Section IV-B. In this example, Q_1 is submitted with a client-claimed causal dependency T_4. DM generates a fresh partial snapshot that satisfies all RSS requirements (i.e., $Snap.2$) for it. $Snap.2$ involves T_1 to T_3 for regularity (including serializability and "read-the-latest-writes") and involves T_4 for causality. As T_5 has a similar available time as T_4, PERSEUS proactively involves it but discards T_6 as it tends to arrive late.

IV. PROTOCOL AND IMPLEMENTATION

In the next, we detail the TP protocol of PERSEUS, illustrate how RSS is enforced, and show how the dynamic snapshot algorithm works. We also introduce an epoch-based garbage collection mechanism to discard outdated ADGs.

A. On-demand Partial Snapshot

We show how ADGs help with obtaining an on-demand partial snapshot for AP queries by illustrating how ADG is generated and provides an intact order history on AP nodes.

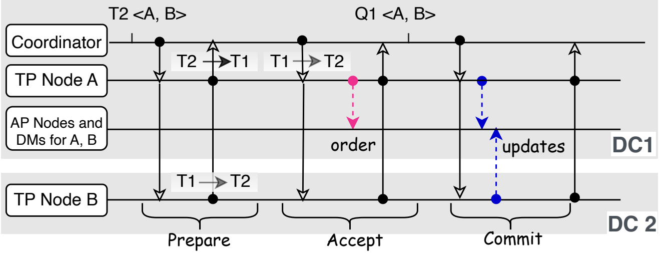


Fig. 5: This diagram shows PERSEUS's message flow to coordinate transaction T_2. We assume T_2 access TP nodes A and B, while there is another ongoing transaction T_1 that also accesses TP nodes A and B and is in conflict with T_2. Our key insight is that: the order (in the pink arrow) between T_1 and T_2 can be obtained before updates (in the blue arrow) are available on AP nodes.

Generation of ADG. We derive the TP protocol of PERSEUS from Janus-TP[2] [71]. For simplicity, we do not consider TP replicas (i.e., the replicas for TP nodes to achieve high availability), which are orthogonal to our paper. Similar to other dependency-graph-based protocols (e.g., [64], [69], [71]), Janus-TP solves conflicts between transactions by letting coordinators determine a single executive order. Specifically, PERSEUS's TP protocol consists of three phases: *Prepare Phase*, *Accept Phase*, and *Commit Phase*.

Algorithm 1 shows the protocol. When a client submits a TP transaction to the coordinator, the coordinator handles the transaction by evoking *DoTransaction(T)*. The coordinator first initially ADG and calculates the read/write set of the transaction. Then, in the *Prepare Phase*, the coordinator sends a prepare message to all relevant TP nodes (i.e., the nodes have the data accessed by T). Each TP node adds the transaction to its local ADGs and sends the combined ADG back to the coordinator. When adding conflict transactions into the local ADG, TP nodes order them with the received order.

As such, the order determined by different TP nodes may be contradictory. To ensure a single order, the coordinator collects the ADGs from TP nodes and 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 immediately. Otherwise, in the *Accept Phase*, the coordinator notifies all relevant TP nodes about the determined order using ADGs, and TP nodes modify their local ADGs according to the given order. As long as the order has been determined, TP nodes asynchronously pass the finalized ADGs to AP nodes (and DMs) who have subscribed to the TP nodes.

After that, the TP nodes send back an acknowledgment to the coordination. Finally, in the *Commit Phase*, the transaction can be finalized and executed. Updates are transferred to corresponding AP nodes using the updates transferring pipeline.

Figure 5 shows a minimal example to coordinate a transaction T_2. Firstly, the coordinator of T_2 sends `prepare` messages to TP nodes A and B. Each node replies with its local dependency graph reflecting dependency between T_2 and

[2] There are two database systems called Janus. To distinguish them, we call the HTAP system Janus-HTAP [10] and the TP system Janus-TP [71].

Algorithm 1: Code for TP coordinator in PERSEUS

```
state $\mathcal{D}_T$ ← Dependency of transaction $T$
state $\mathcal{F}_T$ ← Set of nodes have received $T$ and $\mathcal{D}_T$
function TPCoord::DoTransaction($T$):
    $\mathcal{G}_{coord}$ ⟨$\mathcal{D}_\star, \mathcal{F}_\star$⟩ ← ∅
    $\mathcal{N}$ ← CaculateNodes($T$)
    // Phase.1 Prepare:
    send PreAccept($T$) to $n \in \mathcal{N}$
    wait receive PreAcceptRep($\mathcal{G}_n$) from $n \in \mathcal{N}$
    if ∀ $n \in \mathcal{N}$, $\mathcal{G}_s$ ⟨$\mathcal{D}_T, \mathcal{F}_T$⟩ are identical :
        $\mathcal{G}_{coord}$ ← Union($\mathcal{G}_{coord}$, $\cup_{n \in \mathcal{N}} \mathcal{G}_n$)
        goto Commit Phase
    else:
        $\mathcal{N}'$ ← { n ∈ Majority($\mathcal{D}_T$) }
        $\mathcal{G}_{coord}$ ← Union($\mathcal{G}_{coord}$, $\cup_{n' \in \mathcal{N}'} \mathcal{G}_{n'}$)
        goto Accept Phase
    // Phase.2 Accept:
    send Accept($T$, $\mathcal{G}_{coord}$) to $n \in \mathcal{N}$
    wait receive AcceptReply($Ack.$) from $n \in \mathcal{N}$
    goto Commit Phase
    // Phase.3 Commit:
    send Commit($T$) to each $n \in \mathcal{N}$
    return to client after receiving execution results
```

Algorithm 2: Code for AP coordinator (DM) in PERSEUS

```
1  state $\mathcal{G}_{coord}$ ← Local dependency graph maintained by Coord.
2  state $\mathcal{D}_{causal}$ ← Casual dependency observed in client sessions
3  function APCoord::DoAnalyticalQuery($Q$):
4      $\mathcal{N}$, $\mathcal{C}$ ← CaculateColumns($Q$)
       // Phase.1 Snapshot Generation
5      send PrepareQuery($Q$, $\mathcal{C}$, $\mathcal{N}$) to all $n \in \mathcal{N}$
6      wait receive PrepareQueryReply($\mathcal{G}_n$) from all $n \in \mathcal{N}$
7      $\mathcal{G}_{RSS}$ ← $\cup_{n \in \mathcal{N}} \mathcal{G}_n$
8      $\mathcal{G}_{coord}$ ← Union($\mathcal{G}_{coord}$, $\mathcal{G}_{RSS}$)
9      Snap ← CaculateSnap($Q$, $\mathcal{C}$, $\mathcal{G}_{RSS}$) ∪ $\mathcal{D}_{causal}$
       // Phase.2 Query Execution
10     send CommitQuery($Q$, Snap) to all $n \in \mathcal{N}$
11     wait receive CommitReply($Result_n$) from all $n \in \mathcal{N}$
12     res ← AggregateResults($\cup_{n \in \mathcal{N}} Result_n$)
13     return res

14 function APCoord::CaculateSnap($Q$, $\mathcal{C}$, $\mathcal{G}_{RSS}$):
15     waitlist ← ∅
16     $t_{read}$ ← GetTimeNow.Latest
       // Step.1 Build RSS-Dependency
17     for each column in $\mathcal{C}$ do
18         for each $T$ in $\mathcal{G}_{coord}$ do
19             if $T$.touchedColumns ∩ $\mathcal{C}$ ! = ∅ :
20                 waitlist ← waitlist ∪ {$T$}
       // Step.2 Optimize Freshness
21     for each $T$ in waitlist and $T$ not in $\mathcal{G}_{RSS}$ do
22         if $T$.estimatedTime - $t_{read}$ > threshold :
23             waitlist ← waitlist - {$T$}
24     return waitlist
```

concurrent transactions T_1. According to the received order, node A implies $T_2 \to T_1$, and node B implies $T_1 \to T_2$. Secondly, T_2's coordinator detects the conflict between T_1 and T_2 by combining the received ADGs and then determines $T_1 \to T_2$. After that, the coordinator notifies the TP nodes about the order and can commit T_1 sequentially.

From the diagram, our observation is that the TP protocol of PERSEUS always determines a serializable order before execution (which is common in deterministic databases [34], [69], [79], [87]). This property lets DMs and AP nodes prepare snapshots before data becomes visible to AP queries, hiding the time cost for generating snapshots. Moreover, it gives PERSEUS the opportunity to determine whether to involve specific transactions according to the dynamic snapshot algorithm.

Partially Intact Order History on Nodes. Based on the TP protocol above, we now further explain how ADGs help with obtaining a partially intact order on TP and AP nodes. Recall the example in Figure 3. Compared to a typical dependency graph, ADGs piggyback indirect transaction dependencies among TP and AP nodes. Specifically, in the *Prepare Phase*, nodes send the ADG consisting of all recorded dependencies of *T* to the coordinator instead of the direct dependencies. For instance, when coordinating T_k, node C replies the complete ADG ($T_i \to T_j \to T_k$). Then, when the coordinator combines the ADGs and determines the order, the dependency of $T_i \to T_j$ are piggybacked to node D in the *Accept Phase* (or the *Commit Phase* when the graph is originally acyclical).

Transferring complete ADGs from TP nodes may incur additional overhead on networks and CPU. PERSEUS addresses this challenge based on the invariant that it is sufficient for each TP node to be notified of the existence and the dependency of each transaction only once. In our implementation, PERSEUS binds a set $\mathcal{F}_T$ with each transaction in the ADG. 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 locally recorded ADG $\mathcal{G}_l$, and only piggyback the transaction order in $\mathcal{G}_l$ that are unknown to other involved TP nodes. When processing on the coordinator, PERSEUS merges the received ADG, as well as the metadata ($\mathcal{F}_T$). Thus, TP shards can learn the piggybacking work done by others and avoid redundant work. Finally, the coordinator only sends the minimal ADGs to each TP node based on whether the dependencies in ADG should be piggybacked.

Our evaluation shows the overhead introduced by ADG is small and acceptable (see Section V-B).

B. RSS and Dynamic Snapshot

As presented in Section I and Section III, PERSEUS enforces RSS and optimizes data freshness while keeping RSS invariant. We now detail how to get an RSS-consistent snapshot.

PERSEUS let the first visited AP node (DM) be the coordinator of the query. The coordinator aligns the snapshot, combines the execution results, and sends the execution results back to clients. Algorithm 2 shows the code of AP coordinators. Algorithm 3 shows the code of non-coordinator AP nodes.

As shown in Algorithm 2, 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. In the rest, we focus on the first phase (i.e., the snapshot generation).

Snapshot Generation. Assume a client issues an AP query Q to a DM (i.e., the *coordinator*) with causal dependency $\mathcal{D}_c$. The DM calculates the set of columns $\mathcal{C}$ and the set of TP nodes (partitions) $\mathcal{N}$ that will be accessed by Q (Algorithm 2, line 4). After that, the coordinator sends a message to the relevant AP node's DM to ask for the latest ADGs.

When the AP node's DM receives the message, each DM

Algorithm 3: Code for non-coordinator DM in PERSEUS

```
1  state $\mathcal{G}_{node}$ ← local dependency graph maintained by AP Node
2  function APNODE::PrepareQueryRecv($Q$, $\mathcal{C}$, $\mathcal{N}$):
3      $\mathcal{N}'$ ← SubscribedTPNode()
4      send CalibrateGraph() to all $n' \in \mathcal{N}'$
       // Check real-time Constraints from TP nodes
5      wait receive CalibrateReply($\mathcal{G}_{n'}$) from all $n' \in \mathcal{N}'$
6      $\mathcal{G}_{node}$ ← Union($\mathcal{G}_{node}$, $\cup_{n' \in \mathcal{N}'} \mathcal{G}_{n'}$)
7      return $\mathcal{G}_{node}$

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 $Result$
```

sends a calibrating request to the subscribed TP nodes to align its local ADG (Algorithm 3, line 4) to ensure that the query can observe all the committed transactions before the Q starts in real-time. Even though it may incur an additional round of network communication, such a calibration is critical for achieving the "read-the-latest-write" property. For workloads with good locality, most queries should access data owned by a single data center so that the AP nodes can calibrate the order locally. This is achieved by the partial snapshots (and ADGs) of PERSEUS. In contrast, without ADG, a local TP node can not guarantee the order history in a single data center is intact, thus relying on global coordination even for local queries.

In the calibration step, the messages from TP nodes only carry the transaction IDs that have been committed but not confirmed to be received by the DMs. This is because the dependencies between these transactions (i.e., ADGs) can be later obtained asynchronously in the previous order transferring pipelines. PERSEUS only relies on these IDs to confirm the completeness of received ADGs. Then, DMs send the calibrated ADGs to the coordinator.

After receiving the calibrated ADGs from DMs, the coordinator unifies the dependency. For causality, PERSEUS also combines $\mathcal{D}_c$ to form the finalized dependency set. To optimize freshness, PERSEUS further uses a dynamic snapshot algorithm to prune the ADGs (Algorithm 2, lines 25-27).

Dynamic Snapshot Algorithm. For each query, RSS has a set of essential ordering requirements (i.e., the calibrated ADGs and $\mathcal{D}_c$). However, after meeting these requirements, PERSEUS can still adjust the snapshot by incorporating transactions that are not essential (e.g., concurrent transactions).

Such transactions can be common in PERSEUS due to two aspects: First, PERSEUS transfers orders ahead of updates (Figure 5). 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 largely differ in a geo-distributed deployment, which opens up the space for adjustment. 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.

In this way, PERSEUS can improve freshness as long as the newly incorporated transactions will not increase the wait time of the AP query and the derived snapshots still satisfy RSS. To do so, DMs estimate a shipping delay for the updates of each transaction when receiving their orders from TP nodes.

Recall the TP protocol of PERSEUS in Section IV-A; the order of transactions can be available in either *Accept Phase* (for conflict transactions) or *Commit Phase* (for non-conflict transactions). Thus, the updates can be available in either one or two rounds of network delays. As such, DMs estimate the shipping delay according to two aspects: (1) the number of network rounds before the updates can be transferred to the AP node (2) the estimated RTT between DMs and TP nodes, which determines the time cost to transfer updates. (1) is obtained and transferred together with ADG, while (2) is estimated according to statistics. We do not estimate other costs (e.g., the time for updates merge) since they are difficult to capture and can be constant for different transactions.

Given the estimated shipping delay, the coordinator generates a list (called *waitlist*) for each query. A *waitlist* of an AP query consists of the transaction IDs that need to be visible to the query. DMs then form the *waitlists* from locally calibrated ADG and rule out the transactions that cost a long time to become visible. Without loss of generality, in the algorithm, we use a configurable parameter (*threshold*) to determine which transactions should be ignored. However, in the implementation of PERSEUS, by default, we set the *threshold* as the largest estimated shipping delay of the essential ordering requirements of RSS (i.e., the transactions in the calibrated ADGs and $\mathcal{D}_c$). Thus, newly incorporated transactions will not cause a longer query latency.

It should be noted that PERSEUS does not rely on a precise estimation for the algorithm since PERSEUS can always stop waiting for the transactions with 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) usually lead to underestimating. Moreover, even with overestimation, our approach can consistently outperform the method without optimizations. Our evaluation (Figure 7) shows our method is sufficient for the dynamic snapshot algorithm.

Epoch-based Garbage Collection. To garbage collect the ADGs on TP nodes and DMs, PERSEUS uses an epoch-based garbage collection mechanism similar to previous works [46], [69], [88]. Overall, it removes ADGs from previous epochs periodically to alleviate the storage cost on TP and AP nodes.

Specifically, each TP node keeps an epoch number that increases monotonically. A TP transaction is tagged with an epoch number when it starts. The epoch number on a TP node increases only after all transactions in the last epoch are committed. In this way, the graph vertices of the transactions

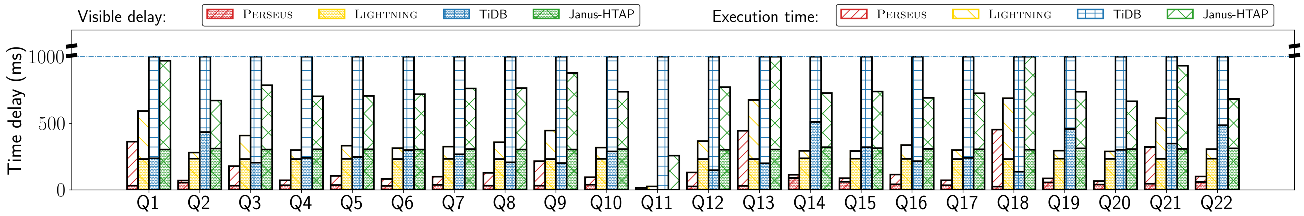


Fig. 6: AP performance on CH-benCHmark. Due to the implementation of different AP engines, the execution time of PERSEUS and three HTAP baselines differs. We present the comparison of the execution time for interested readers and focus on the bottom parts (visibility delay) in our discussion, which is one of our major contributions. Bars with more than $1000ms$ are compressed for readability.

(including all the predecessors) from the outdated epoch can be safely discarded by both TP nodes and DMs.

V. EVALUATION

In this section, we study the performance of PERSEUS and compare it with state-of-the-art works in various aspects.

A. 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 Docker container and used tc [45] to control the RTT among nodes.

Implementation. We implemented PERSEUS's TP nodes, ADG, and updates transferring pipeline with 3062 lines of C++ code. All protocol messages are implemented with asynchronous RPC calls. Each TP node has three threads: one for executing transactions, one for handling I/O requests, and the other for transferring updates and ADGs to AP nodes. We implemented DM and AP nodes with 3872 lines of C++ code based on an in-memory query engine (PTree [58]). We set the epoch interval for garbage collection as $3s$.

Baselines. We compared PERSEUS to three latest decoupled HTAP systems (TiDB [42], F1 LIGHTNING [93], and Janus-HTAP [10]) and a TP system (Spanner-RSS [41]). The three HTAP systems all provide weaker consistency guarantees 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 decoupled HTAP systems because they either cannot support our benchmark (e.g., Hologres [47] supports only batched writes) or requires special hardware (e.g., VEGITO [82] require RDMA) that is not available in a geo-distributed environment.

Spanner-RSS [41] is the only database providing RSS. 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 [55].

As F1 LIGHTNING [93] is not open-source, we implemented its safe timestamps mechanism on our codebase using the same AP engine and updates transferring pipeline as PERSEUS. Therefore, the performance of our implementation may not directly reflect the industrial production. In contrast, we provide an insightful analysis of the snapshot mechanism, which is the focus of our paper. We set LIGHTNING's safe timestamp checking interval to $30ms$: a lower value will not further improve its freshness but will 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., $50ms$ for a batch interval) for experiments.

We run TiDB using its open-source code [11] and tune the performance according to the official guidelines.

Workloads. We used three workloads to emulate diverse application scenarios of geo-distributed HTAP.

CH-benCHmark is built from a unified schema of TPC-C [28] and TPC-H [27]. It has been widely used for evaluating HTAP databases (e.g., [42], [82], [91]). In the workload, each client had a warehouse ID. To emulate local queries, we let the AP client query data based on its own warehouse ID. Thus, the queries access only local TP nodes (participants).

HTAPBench-E. To evaluate geo-distributed AP queries, we built a new workload (HTAPBench-E). It 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. Specifically, Broker Report (BR) generates a real-time report on the potential of brokers; Market Monitor (MM) reports the performance of the market by watching all customers' current holdings. Each query accesses global data.

Microbench-Zipf. To study the performance under a skewed workload, we built microbench-Zipf. We populated each TP node with one million 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 $2ms$ and the cross-data-center RTT to $100ms$ (as suggested in [12], [14]).

Our default deployment had 5 data centers and 100 TP nodes. Each data center contained 20 TP nodes, 5 DMs, and 5 AP nodes (Section III-A). Each DM and AP node subscribes to the TP nodes from one data center (i.e., 20 TP nodes). As such, each data center manages a full columnar data copy. This helps us eliminate geo-distributed join and task scheduling, which are orthogonal to our paper.

B. 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 ~ $35k$ transactions per second (tps), which is comparable to other geo-distributed TP databases (e.g., [20], [25]).

HTAP Sys.	Staleness	Snapshot Generation		visibility delay
		coordination	wait exe.	(total)
LIGHTNING	237.2ms	0ms	0ms	237.2ms
TiDB	6.2ms	121.5ms	103.6ms	231.3ms
Janus-HTAP	121.8ms	127.5ms	53.7ms	303.0ms
PERSEUS	7.1ms	7.2ms	14.9ms	29.2ms

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

The AP performance (i.e., execution time) of PERSEUS is also better or comparable to baselines (Figure 6).

The highlight of this experiment is that PERSEUS provides a stronger consistency than baselines, which should be desirable for real-time decision-making applications, while the visibility delay of PERSEUS on local queries was still 86.8%, 93.2%, and 90.1% lower (Figure 6). This is because PERSEUS’s partial snapshots eliminated cross-data-center communication, while baselines require global coordination to find an even weaker snapshot (see Section II-C).

We collected the breakdown data to further understand the visibility delay (Table III). LIGHTNING has long staleness as its safe timestamp requires global communication to all TP nodes and is straggled by the slowest clock. 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. Specifically, TiDB’s visibility delay came from two parts: (1) a WAN RTT to the global version manager (121.5ms); (2) waiting for all updates ordered before queries arrive (103.6ms). Janus-HTAP took 121.8ms to release a consistent snapshot (i.e., batch) by adding barriers to transactions’ orders, and it 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 (52.7ms). In PERSEUS, when a query arrived at the DM, the DM calibrated the graph contacting with local TP nodes (7.2ms) and waited for the transactions in ADG to become visible (Section IV-B). Unlike TiDB, which needs to wait for *all* transactions ordered before queries in a global order, PERSEUS only waits for updates from local nodes.

C. *Performance of Geo-distributed AP Queries*

We evaluated the performance of PERSEUS’s geo-distributed queries using HTAPBench-E. The results are in Figure 7. The highlight is that PERSEUS ensured stronger consistency, while PERSEUS’s visibility delay was comparable to baselines.

The visibility delay of PERSEUS became larger than local queries because, in this experiment, AP queries required global snapshots instead of partial ones. Thus, PERSEUS did cross-data-center coordination (Section IV-B) to meet the RSS guarantees (i.e., for “read-the-latest-writes”).

In contrast, three baselines processed geo-distributed queries similarly to local ones. Due to the workload properties (i.e., more transaction writes per second), the visibility delay of TiDB and Janus-HTAP increased a bit since TiDB waited for more transactions to become visible, and Janus-HTAP needs more time to allocate consistent barrier cuts.

HTAP Sys.	Staleness	Snapshot Generation		visibility delay
		coordination	wait exe.	(total)
LIGHTNING	235.2 ms	0 ms	0 ms	235.2 ms
TiDB	6.2 ms	132.5 ms	204.8 ms	337.3 ms
Janus-HTAP	160.8 ms	115.1 ms	127.5 ms	403.4 ms
PERSEUS	5.9 ms	103.4 ms	123.2 ms	232.5 ms

TABLE IV: Visibility delay breakdown of HTAPBench-E BR.

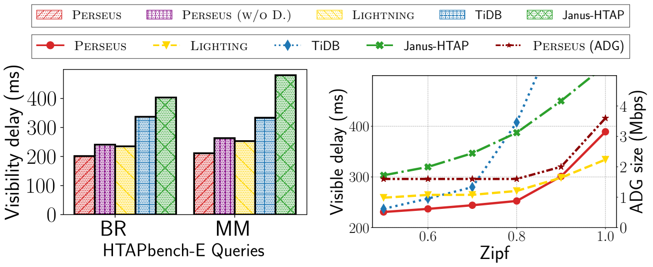


Fig. 7: HTAPBench-E

Fig. 8: Microbench-zipf

Table IV 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 (cross-data-center). Finally, the results were sent back to the coordinator for aggregation. We do not reiterate the breakdown of the baselines since they behaved similarly to local ones.

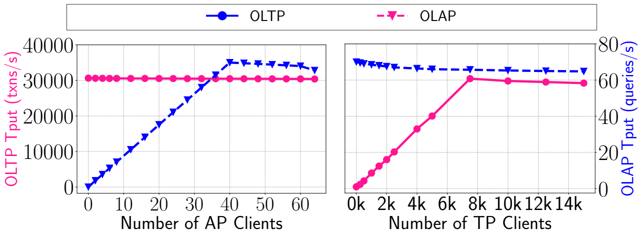
D. *Performance on Skewed Workloads*

In this experiment, we run Microbench-Zipf. As shown in Figure 8, TiDB’s visibility delay increased dramatically with skewness because skewed updates 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 8). In contrast, LIGHTNING’s visibility delay increased only slightly due to the overhead of merging more keys in the same AP nodes.

E. *Cost of transferring updates and ADGs.*

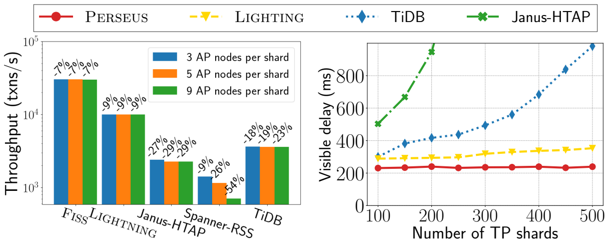
To investigate the overhead caused by transferring ADGs, we ran a TP-only workload (TPC-C). We disabled the transmissions of updates and ADGs (using typical dependency graphs for TP). PERSEUS achieved ~37k tps TP throughput with 1.5k clients in each data-center. Then, we enabled the update transferring 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 very small (i.e., ~32 bytes each transaction) in network packets. In addition, PERSEUS’s epoch-based tracing method (Section IV-B) prevents the ADG size from becoming arbitrarily large. In our experiments, each node consumed at most 26.4 $Mbps$ bandwidth for transferring ADGs.

Moreover, as shown in the later experiments (Figure 9a),



(a) Perf. isolation of OLTP (b) Perf. isolation of OLAP

Fig. 9: Performance isolation of PERSEUS.



(a) Scalability of OLTP (b) Scalability of OLAP

Fig. 10: Scalability compared to HTAP and Spanner-RSS.

PERSEUS's total cost (7%) on TP was lower than TiDB's (18%) and Janus-HTAP's (27%) because PERSEUS did not add extra steps in TP. PERSEUS was comparable to LIGHTNING (9%) but ensured stronger consistency.

F. Effectiveness of Dynamic Snapshot.

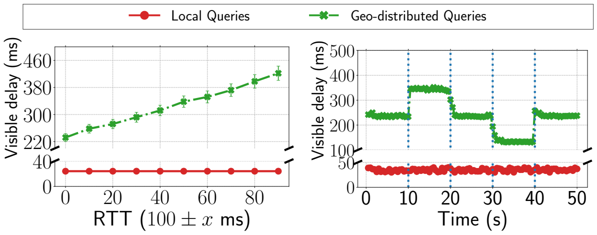
We conducted an ablation study to evaluate the effectiveness of dynamic snapshots on geo-distributed queries (using HTAPBench-E). In particular, we disabled the dynamic snapshot algorithm in our system variation: PERSEUS-w/o D. The variation lets AP queries only observe essential updates required by RSS. As shown in Figure 7, the visibility delay of PERSEUS-w/o D. increased by $40.8ms$ on average. This is because, compared to PERSEUS, PERSEUS-w/o D. ignored all new concurrent transactions, and data staleness was bounded to the slowest delivery of essential updates required by RSS.

G. Performance Isolation and Independent Scalability

Recall the unique strength of the decoupled HTAP database in achieving performance isolation and independent scalability (see Section I). In this experiment, we study the performance of PERSEUS on these two properties.

Performance Isolation. Following the methods used in [23], [82], [91], 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 ran 1800 TP clients per data center for a peak TP throughput; then, we added AP clients until AP throughput is saturated. As shown in Figure 9a, PERSEUS maintained a stable TP throughput. This is because adding AP workload only costs a few calibration messages (Section IV-B), which is off the critical path of TP. Second, we used 40 AP clients for a peak AP throughput and then added TP clients. As shown in Figure 9b, the AP throughput of PERSEUS dropped slightly (up to 12%) due to the cost of handling more updates. Note that 40 AP clients had already saturated the CPU resources of our server since AP is a computation-intensive task.



(a) Network Fluctuation (b) Network Spike

Fig. 11: PERSEUS's visibility delay under network anomalies.

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 10a by comparing them to the TP-only setting (i.e., without AP nodes). 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 the Paxos [53] group, affecting the critical path of transaction processing. This validates our motivation to build a decoupled HTAP instead of naively using TP backups.

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 10b 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 visibility delay increased proportionally because it relied on the global order to serve queries. Janus-HTAP's visibility delay increased because the cost for finalizing a consistent cut in batch graph increased with more concurrency in TP.

H. 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 11a, 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 11b, 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 as they were coordinated locally. The visibility delay of geo-distributed queries changed with network spikes.

VI. CONCLUSION

We presented the design, implementation, and evaluation of PERSEUS, a decoupled HTAP system with regular sequential serializability. PERSEUS generates on-demand partial snapshots for AP queries using ADG and, thus, decouples the concurrency control between TP and AP, achieving independent scalability. In addition, PERSEUS optimizes data freshness for geo-distributed AP queries using dynamic snapshots.

References

[1] Aws glue: Discover, prepare, and integrate all your data at any scale. https://aws.amazon.com/glue/.

[2] Consistency model. https://en.wikipedia.org/wiki/Consistency_model#Strong_consistency_models.

[3] Google cloud: Performing etl from a relational database into bigquery using dataflow. https://cloud.google.com/architecture/performing-etl-from-relational-database\-into-bigquery.

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

[5] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. Proteus: Autonomous adaptive storage for mixed workloads. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 700–714, New York, NY, USA, 2022.

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

[7] Nitin Agrawal and Ashish Vulimiri. Low-latency analytics on colossal data streams with summarystore. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 647–664, 2017.

[8] Amazon. Amazon Shop. https://www.amazon.com/.

[9] Vaibhav Arora, Ravi Kumar Suresh Babu, Sujaya Maiyya, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Zhiyanan ., and Zhujianfeng . Dynamic Timestamp Allocation for Reducing Transaction Aborts. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 269–276, July 2018. ISSN: 2159-6190.

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

[11] TiDB authors. Tidb's github. https://github.com/pingcap/tidb.

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

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

[14] Microsoft Azure. Azure network round-trip latency statistics. https://docs.microsoft.com/en-us/azure/networking/azure-network-latency.

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

[16] Amirali Boroumand, Saugata Ghose, Geraldo F Oliveira, and Onur Mutlu. Polynesia: Enabling high-performance and energy-efficient hybrid transactional/analytical databases with hardware/software co-design. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2997–3011. IEEE, 2022.

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

[18] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment*, 9(13):1401–1412, 2016.

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

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

[21] Alibaba Cloud. Alibaba Cloud's Global Infrastructure. https://www.alibabacloud.com/global-locations.

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

[23] Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, and Rui Oliveira. Htapbench: Hybrid transactional and analytical processing benchmark. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 293–304, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, pages 1–6, 2011.

[25] 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. Spanner: Google's globally-distributed database. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, October 2012.

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

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

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

[29] Lei Deng, Jerry Gao, and Chandrasekar Vuppalapati. 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*, pages 256–266. IEEE Computer Society, 2015.

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

[31] ebay inc. Shop eBay. https://www.ebay.com/.

[32] Song et al. PERSEUS: Achieving Strong Consistency and High Scalability for Geo-distributed HTAP (Technical Report). https://github.com/icde24p590/perseus/blob/main/tr.pdf, 2023.

[33] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, January 2017.

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

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

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

[37] Yupeng Fu and Chinmay Soman. Real-time data infrastructure at uber. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2503–2516, 2021.

[38] Gartner. In-memory dbms vs in-memory marketing. https://blogs.gartner.com/donald-feinberg/2014/09/28/in-memory-dbms-vs-in-memory-marketing/?_ga=2.172848676.641528062.1639488442-337805139.1639488442.

[39] Gartner. 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.

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

[41] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular sequential serializability and regular sequential consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 163–179, 2021.

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

[43] Yihe Huang, Hao Bai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. The impact of timestamp granularity in optimistic concurrency control. *arXiv preprint arXiv:1811.04967*, 2018.

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

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

[46] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, 2010.

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

[48] Hyungsoo Jung, Hyuck Han, Alan Fekete, and Uwe Röhm. Serializable snapshot isolation for replicated databases in high-update scenarios. *Proceedings of the VLDB Endowment*, 4(11):783–794, 2011.

[49] Guoxin Kang, Lei Wang, Wanling Gao, Fei Tang, and Jianfeng Zhan. Olxpbench: Real-time, semantically consistent, and domain-specific are essential in benchmarking, designing, and implementing htap systems. *arXiv preprint arXiv:2203.16095*, 2022.

[50] Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. Rethink the scan in mvcc databases. SIGMOD '21, page 938–950, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. Diva: Making mvcc systems htap-friendly. In *Proceedings of the 2022 International Conference on Management of Data*, pages 49–64, 2022.

[52] Tae-Kyung Kim, Seung-Hyun Jung, Kyoung-Ran Kim, Jae-Soo Yoo, and Wan-Sup Cho. Hyperdb: a pc-based database cluster system for efficient olap query processing. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 288–293, 2007.

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

[54] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

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

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

[57] Steve LaValle, Eric Lesser, Rebecca Shockley, Michael S Hopkins, and Nina Kruschwitz. Big data, analytics and the path from insights to value. *MIT sloan management review*, 52(2):21–32, 2011.

[58] Guanling Lee, Yi-Chun Chen, and Kuo-Che Hung. Ptree: Mining sequential patterns efficiently in multiple data streams environment. *Journal of Information Science & Engineering*, 29(6), 2013.

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

[60] Kitaek Lee, Insoon Jo, Jaechan Ahn, Hyuk Lee, Hwang Lee, Woong Sul, and Hyungsoo Jung. Deploying computational storage for htap dbmss takes more than just computation offloading. *Proceedings of the VLDB Endowment*, 16(6):1480–1493, 2023.

[61] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell+: Snapshot isolation without snapshots. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 425–441, 2020.

[62] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making {Geo-Replicated} systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

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

[64] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. 2020.

[65] Yi Lu, Xiangyao Yu, and Samuel Madden. Star: Scaling transactions through asymmetric replication. *arXiv preprint arXiv:1811.02059*, 2018.

[66] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. MaaT: effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, January 2014.

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

[68] Pratap Chandra Mandal. Pricing strategies of multinationals for global markets–considerations and initiatives: Pricing strategies for global markets. *International Journal of Business Strategy and Automation (IJBSA)*, 1(1):24–36, 2020.

[69] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, October 2014. USENIX Association.

[70] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Github: a stable commit of NYU-NEWS/Janus. https://github.com/NYU-NEWS/janus/tree/3c1b60cd965551b4bd3b1f3c475e16fdde2e4c2b.

[71] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 517–532, 2016.

[72] MySQL. Mysql heatwave. https://dev.mysql.com/doc/heatwave/en/heatwave-introduction.html, 2022.

[73] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[74] Tarikul Islam Papon, Ju Hyoung Mun, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. Relational fabric: Transparent data transformation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3688–3698. IEEE, 2023.

[75] M Pezzini, D Feinberg, N Rayner, and R Edjlali. Real-time insights and decision making using hybrid streaming, in-memory computing analytics and transaction processing, 2016.

[76] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[77] Jon TS Quah and M Sriganesh. Real-time credit card fraud detection using computational intelligence. *Expert systems with applications*, 35(4):1721–1732, 2008.

[78] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

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

[80] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-store: A real-time oltp and olap system. *arXiv preprint arXiv:1601.04084*, 2016.

[81] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F Ilyas. Real-time lsm-trees for htap workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 1208–1220. IEEE, 2023.

[82] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 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*, 2021.

[83] Takamitsu Shioi, Takashi Kambayashi, Suguru Arakawa, Ryoji Kurosawa, Satoshi Hikida, and Haruo Yokota. Serializable htap with abort-/wait-free snapshot read. *arXiv preprint arXiv:2201.07993*, 2022.

[84] Adriana Szekeres and Irene Zhang. 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*, pages 1–8, 2018.

[85] Michael B Teitz. Neighborhood economics: Local communities and regional markets. *Economic Development Quarterly*, 3(2):111–122, 1989.

[86] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[87] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 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*, May 2014.

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

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

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

[91] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. Polardb-imci: A cloud-native htap database system at alibaba. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

[92] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zhang. Unifying timestamp with transaction ordering for mvcc with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2021*.

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

Technical Report

This report provides a technical proof sketch for our paper "PERSEUS: Achieving Strong Consistency and High Scalability for Geo-distributed HTAP".

It consists of five parts: Section A presents our proof for RSS and individual scalability; Section B shows the problem of SS; Section C provides a case study on existing timestamp protocols; Section D shows our supplemental results. Section E discusses impacts and future work.

A. CORRECTNESS AND PERFORMANCE ANALYSIS

In this section, we show that PERSEUS is correct for ensuring RSS. We first show that ADGs can efficiently capture the complete ordering requirements of RSS. We then formally prove that PERSEUS achieves RSS (including serializability, casual dependencies, and read the latest write property).

Besides RSS, we also provide an analysis of how PERSEUS detaches the ordering of AP from the critical path of TP, implying independent scalability of TP and AP.

A. ADGs capture all ordering requirements of RSS.

We prove that the dependencies captured by PERSEUS's ADG is sufficient to support partial snapshots under RSS.

Lemma 1. *If $node_i$ is a participating shard of transaction T_n, $node_i$ should note T_n and all its predecessor dependencies after the pre-accept phase.*

Proof. We prove the property in the scope of a dependency chain by induction: $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$. A similar example is shown in Section IV-A. In PERSEUS, the dependency $T_1 \rightarrow T_2$ can be captured by all participants of T_2, and the dependency $T_1 \rightarrow T_2 \rightarrow T_3$ should be captured by all participants of T_3 by piggyback mechanism (Algorithm 1), even when the participants of T_3 do not take participation in T_1 directly. By induction, the whole dependency chain is noted by all participants of T_n.

Then, we extend our proof to the graph scope; a transaction in a weekly connected component (*WCC*) can reside in multiple dependency chains. Each dependent transaction should always appear in one of the chains. Since we have proved that each dependency chain of the transaction is completed, then the *WCC* of the transaction should be completed.

To maintain an RSS-consistent snapshot, a safe guarantee is to let AP nodes observe global dependency graphs with all transactions. We refine this requirement by only tracing the dependency in *WCC* of the transactions that modified the data item held by the AP node. Assuming an AP node held a partial set of arbitrary TP shards, if a transaction T_i is contained in a local ADG $\mathcal{G}_l$ on TP shard, then with Lemma 1, its dependency must be complete. Then, DMs can merge ADGs from relevant AP shards and build a partial snapshot to serve the query. □

B. PERSEUS *ensures RSS.*

We prove that PERSEUS ensures RSS with three steps. First, we demonstrate that PERSEUS ensures serializability: the execution of all operations in PERSEUS is equivalent to being executed in a serial schedule **S**. We prove PERSEUS ensures serializability for all transactions and then prove PERSEUS preserves serializability after adding AP queries. In PERSEUS, all transactions are coordinated by a TP protocol derived from Janus-TP, which is proven to ensure serializability because the union of the dependency graph among all transactions is acyclic. PERSEUS's TP protocol augments Janus-TP by replacing the dependency graph with ADG (Section IV-A), which *preserves serializability* because the *WCC* of dependent transactions in ADG is a *superset* of the directly dependent transactions in Janus-TP's dependency graphs, and the ADG is still acyclic.

Intuitively, Queries' serializability is considered as the relationships with read-write transactions, i.e., each query observes a consistent snapshot of the system that (1) preserves transactions' atomicity and (2) preserves the order in the equivalent serial order **S**. These two requirements are enforced by ADG (Section IV-A) as it ensures the completeness of each transactions' dependencies (Lemma 1) and all updates.

Second, we prove that PERSEUS preserves causal dependencies. For transactions, PERSEUS's TP protocol provides strict serializability (SS), and the set of causal dependencies **must** be a subset of all real-time requirements of SS. Specifically, two operations are causally dependent ($O_1 \rightsquigarrow O_2$) if either a client issuing O_2 remembers O_1, or O_2 reads a value from the execution result of O_1. In either case, O_1 must finish before O_2 starts and thus is before O_2 in real-time. PERSEUS enforces causal dependencies from transactions to queries because PERSEUS carries them with ADG and waits until all these dependent transactions and their predecessors are met (Section IV-B).

For any causal dependency $Q \rightsquigarrow T$, we need to verify $Q \prec T$ (see the definition of notations in Section II-B). We consider the writes and reads in T separately. T's writes must be ordered after Q, because Q has finished and cannot see unhappened writes. We prove that for all keys accessed by Q, T sees at least a new version as Q. This can be confirmed by transitivity. In PERSEUS, when Q observes updates of some read-write transaction W, the ADG of W must be committed before Q finishes because PERSEUS only makes W's updates visible after W's ADG is committed. Since Q finishes before T starts, T must not be in the ADG (i.e., $W \nprec Q$). As PERSEUS's transactions ensure serializability, we have $Q \prec W$, and the causality is ensured.

Third, PERSEUS ensures that a read-only operation observes all write operations finished before it starts. PERSEUS ensures read-only transactions see all finished writes because PERSEUS's TP protocol ensures SS. For queries, PERSEUS uses a calibration step (Section IV-A) to fetch the finished read-write transactions on relevant TP shards. The fetched set of the read-write transactions is guaranteed to be a superset of transactions having finished before the query starts because any such transactions must also finish before the calibration message arrives (**Lemma 1**).

C. Perseus *decouples the concurrency control of TP and AP*

Perseus decouples the concurrency control of TP and AP in two aspects. First, the message complexity on TP for ensuring AP consistency is $O(1)$ to the number of AP nodes and concurrent AP queries. Perseus ensures this by not involving the AP query in the TP protocol. Second, Perseus prepares a consistent state for a query that only involves actively contacting relevant nodes. This is ensured by Perseus's ADG (Lemma 1), and the calibration process of Perseus's protocol only involves relevant TP nodes (Algorithm 2).

B. A Study on Strict Serializability

We show that strict serializability (SS) is fundamentally costly to realize in distributed HTAP databases. Specifically, by theoretical analysis, we find that the strawman approach to enforcing SS can easily impose heavy performance overhead on coordination, sacrificing performance and scalability.

A. System Model and Preliminaries

To begin with, we first reiterate our system model of Perseus, along with the preliminaries that are used in Section II-B.

System Model. The system consists of a group of *TP nodes* and a group of *AP nodes*. Nodes don't have synchronized clocks (e.g., atomic clocks). Networks between nodes are also asynchronous: nodes should communicate with each other using asynchronous messages. Each TP node holds a database shard, and each AP node subscribes to updates from a set of TP nodes. The TP nodes run a distributed concurrency control protocol for client requests updating the database and asynchronously ship the updates to the AP nodes in streams.

Operations and Events. We consider three operation types: R for a read-only transaction, W for a read-write transaction, and Q for an AP query. We use O for operations of any type. We say O_1 and O_2 *conflict* if they access the same key and one of the accesses is a write operation, denoted as $O_1 \otimes O_2$. We use $send_i(m)$ and $recv_j(m)$ to denote the events for sending and receiving the message m.

Real-time order. We say event e_1 precedes e_2 if e_1 happens before e_2 according to a hypothetical global wall clock, denoted as $e_1 \to e_2$. We say two operations $O_1 \to O_2$ if $\forall i, j : finish_i(O_1) \to start_j(O_2)$. We use $RT(e)$ to represent the complete set $\{e_x : e_x \to e\}$, and $RT(O)$ for the complete set of operations preceding O. Additionally, we use $RT_A(O)$ for the complete set of operations preceding O on node A.

Transaction ordering. We use $<$ to represent the "order before" requirements among requests. For instance, if O_2 reads a key written by O_1, we have $O_1 < O_2$, and the equivalent order for serializability should preserve this ordering requirement.

Given the terminologies above, we have the definition of strict serializability (for short, SS):

Strict Serializability (SS). All operations are equivalent to being executed in a serial schedule S that preserves real-time ordering, i.e., for any $O_1 \to O_2$, we have $O_1 < O_2$.

We then formalize the two essential yet straightforward requirements for scalability in decoupled HTAP systems. **R1**. For an analytical query Q, an AP node should contact only TP nodes containing the data accessed by Q. This property ensures the cost of coordinating an AP query is irrelevant to the number of TP nodes. **R2**. The complexity of the messages used to achieve consistency for AP queries on the *critical path* of TP should be $O(1)$ to the number of concurrent AP queries and AP nodes. The critical path of TP refers to the life circle of transactions inside the databases from when the transaction is issued by the user to when the transaction results are returned to the user. This property ensures the span of AP nodes will not degrade the performance of TP. This is because, by definition, the cost of TP for AP will not increase with the number of AP nodes.

B. Completeness in Distributed Systems

Before we discuss the inefficiency of ensuring strict serializability in Perseus, we present a lemma, which should be general to all distributed systems. Specifically, the lemma demonstrates how to capture the complete set of real-time events in an asynchronously networked system.

Lemma A. *Consider any two nodes A and B with event β on B, and A has events that trigger independently from B. If B wants to acquire a complete superset of $RT_A(\beta)$, B must actively send a message to A after event β triggers.*

Proof Intuition. In an asynchronous network, system nodes infer real-time order among events through the transitivity of "happened before" relation [54]. For instance, in the following construction, we have $\alpha \to \beta$ by considering $\alpha \to send(m_1) \to recv(m_1) \to \beta$. However, single-sided messages from A to B cannot ensure the completeness of $RT(\beta)_A$. For any such message m_k, there can always be an event α' that succeeds $send(m_k)$ but precedes β due to asynchrony because the order of $send(m_k)$ and β is undetermined. For example, in the following construction, B cannot guarantee to capture the real-time dependency $\alpha' \to \beta$ through the message m_1.

A: $\alpha, send(m_1), \alpha', \cdots$
B: $recv(m_1), \beta$

To ensure completeness, one always needs an event e on A working as an anchor point with $\beta \to e$ as the right boundary. Thus, node A can send the complete set of $RT_A(\beta)$, since for all $e_i \in RT_A(\beta)$, we have $e_i \to \beta \to e$. As A's events (e) can trigger independently from B's event (β), B should always guarantee the order between e and β by sending an active message from B to A (thus we have $\beta \to send(m_2) \to e$), guaranteeing A only triggers e after β in real-time order, as the construction below: since $\alpha' \to \beta \to send(m_2) \to e \to recv(m_2)$, for any event $\alpha' \to \beta$, we have $\alpha' \to m_3$. Therefore, m_3 can carry a guaranteed complete superset of $RT_A(\beta)$.

A: $\alpha, \alpha', \cdots$ $recv(m_2), \mathbf{e}, send(m_3)$
B: $\beta, send(m_2)$ $recv(m_3)$

C. Strict Serializability Imposes Performance Costs

We insist strict serializability (SS) in HTAP is overly strong and expensive: it can easily lead to bad performance at the cost of scalability. According to the definition, SS requires each query Q to capture the completeness of $RT_{\#}(Q)$ on all

TP nodes, including those real-time orders caused by transitivity between read-only queries. Specifically, SS guarantees a snapshot visible to AP queries should always be more up-to-date than any previously visible snapshot. For example, as a similar example discussed in [41], if a user Alice sees the latest write made by a user Bob and notifies another user Allen (even using an out-of-box message, e.g., a telephone call) about the writes, given SS, Allen must be able to read the writes made by Bob in the database. To ensure the real-time order between read-only operations, it usually relies on lumping the concurrency control of TP and AP together. In contrast, RSS does not enforce the real-time order between AP queries, thus favoring performance and scalability. As analytical queries always have a much longer execution time than read-only transactions and decision-making queries are usually interleaved with transactions, the enforced real-time order of AP queries in SS is overly strong.

We illustrate the difference between SS and RSS by analyzing protocols in PERSEUS below:

A: $W(A=1)$ $R(A)$
B: $W(B=1)$ $recv(m_1)$
AP: $Q(B)$, $send(m_1)$

Consider the above construction in PERSEUS when using ADGs for TP protocol. PERSEUS executes the transactions according to the resolved topology order in the dependency graph; thus, transactions can be executed in a one-shot manner for better performance. Similar designs are generally used in multiple deterministic TP databases. We show that, in PERSEUS, an AP node cannot collect a set $\mathbb{O}_Q$ guaranteed to contain all operations preceding Q in real-time ($RT(Q) \subset \mathbb{O}_Q$) while guaranteeing independent scalability.

We assume $R(A) \to Q(B)$ in real-time order. Thus, if the database provides SS, the database should always guarantee $R \prec Q$; for example, assume $R(A)$ is issued by Alice and Q(B) is issued by Allen, and the real-time order can be revealed by an out-of-box message. W is a transaction (for example, the writes made by Bob) that includes $W(A=1)$ and $W(B=1)$; W is concurrent to both $R(A)$ and $Q(B)$, which means the orders between $R(A)$ and W, $Q(B)$ and W are not specific according to causality. However, in our construction, $R(A)$ reads W's update (i.e., $W(A=1)$), thus we should have $W \prec R(A)$. Therefore, the only possible SS schedule is $W \prec R(A) \prec Q(B)$ (i.e., in our previous example, Allen has to be guaranteed to observe the writes made by Bob, $W \prec Q(B)$).

However, how can the AP node know the existence of $R(A)$ and the order of $W \prec Q(B)$ by transitivity? Due to the requirement **R2**, $Q(B)$ and the AP nodes cannot participate in the critical path of concurrency control protocol for transactions. Therefore, the AP node must observe $R(A)$ by sending active messages to the TP nodes. Due to the requirement **R1**, the AP node can communicate only with node B, hoping that B knows the existence of R. Due to Lemma B-B, to get $RT_B(start(Q))$, the AP node must actively send a message (m_1) to B after $start(Q)$: any transaction finishing before Q in must finish before $recv(m_1)$. If B wants to acquire $RT_A(recv(m_1))$ that guarantees to include R, B has to send out another message after $recv(m_1)$ due to Lemma B-B, which violates **R1**.

Takeaways. Strict serializability in decoupled HTAP is expensive and can be at the cost of scalability.

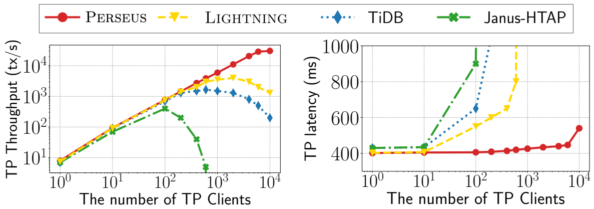
C. Case study on Safe Timestamps

One may consider extending safe timestamp mechanisms used for read-only transactions in geo-distributed OLTP systems to support queries in HTAP. We take Ocean Vista [34], one of the latest OLTP systems that use timestamps to order transactions, as an example. In Ocean Vista (OV), each transaction is assigned a unique timestamp. For serializability, each node executes relevant transactions in ascending timestamp order. To achieve this, each node must ensure that no new transactions with smaller timestamps will be inserted when executing a transaction with timestamp ts. OV uses a watermark (i.e., safe timestamp) mechanism. Specifically, each data center maintains a per-data-center watermark by tracking all intra-data-center nodes, meaning that this data-center will coordinate no new transaction with a smaller timestamp than the per-data-center watermark. Data-centers exchange their per-data-center watermarks periodically, and the minimum per-data-center watermark is the global watermark and transactions with timestamps smaller than the global watermark can be executed.

However, as watermarks are exchanged asynchronously among data-centers, each data-center's view of the global watermark is *not tight*, not containing all finished transactions. Therefore, to ensure read-finished-writes, even a data-center-local query has to be assigned a timestamp equivalent to its per-data-center watermark, larger than this data-center's view of the global watermark, and to wait for this data-center's view of the global watermark to pass its timestamp, with rounds of WAN RTT waiting.

D. Supplemental Evaluation Results

We provide additional evaluation results of PERSEUS for interested readers.



(a) TP Cluster Throughput (b) TP 90% latency

Fig. 12: TP throughput and latency of PERSEUS and baselines with TP-only workloads (CH-benCHmarks with AP parts disabled).

The throughput and latency of TP imply the volume of generated data per second, which squeezes the ability of HTAP systems to ship changed data and merge data into AP storage. Hence, we also compare TP performance of PERSEUS, LIGHTNING, TiDB and Janus-HTAP using CH-benCHmark with OLTP-only workload in geo-distributed environments.

The results are shown in Figure 12a and Figure 12b. TiDB and Janus-HTAP adopt 2PC OLTP protocol with 2PL and are not designed for geo-distribute deployment, so they achieve much lower TP throughput (i.e., ~1500 tps and ~700 tps respectively) in our experiment. Their 90% latency increased dramatically when a large number of TP clients were configured. PERSEUS's throughput showed a positive correlation to the number of TP clients at the beginning and achieved a peak throughput (~35k tps) with the client's number of 10k.

E. IMPATS AND FUTURE WORKS

A. *Trending Applications*

Social trading (e.g., eToro, Ameritrade) leverages TP to do transactions and AP to analyze real-time market trends. In a social trading network, opinion leaders can significantly impact capital markets as their transactions can be automatically followed by many users and cause stock volatility in several seconds. Thus, AP queries with strong consistency and data freshness to reflect real-time stock trends are highly desirable for all users to make timely decisions.

Data-driven real-time pricing (e.g., online taxi, auction, and supply chain) is prevalent to maximize revenue. TP processes user transactions to update the supply and demand continuously; AP makes swift decisions based on real-time results and generates new TP transactions to adjust prices automatically. Therefore, the "(AP) read latest (TP) write" of strong consistency is desirable for diverse revenue-oriented scenarios: AP queries always see the latest price update.

B. *Future Work*

PERSEUS can be a practical template for developing new decoupled HTAP systems that demand diverse trade-offs between consistency and scalability. For instance, one may consider building an HTAP system with a weaker consistency (e.g., process-ordered serializable) for edge computing applications; she can still use PERSEUS's ADG to achieve a consistent partial snapshot with both scalability and serializability while enjoying even better data freshness than PERSEUS. One may also build an HTAP system with strict serializability but give up independent scalability when deploying strongly coupled applications within a single data center at a small scale. Moreover, PERSEUS can also inspire the community to design new consistency levels specific to HTAP systems, such as a consistency level with *guaranteed* bounded staleness or a consistency level that treats transactions from different datacenters differently. We leave these exciting developments as our future work.