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., $W \not\prec Q$). 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 $e_1 \rightarrow e_2$. We say two operations $O_1 \rightarrow O_2$ if $\forall i, j : finish_i(O_1) \rightarrow start_j(O_2)$. We use $RT(e)$ to represent the complete set $\{e_x : e_x \rightarrow e\}$, 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 $\prec$ to represent the "order before" requirements among requests. For instance, if O_2 reads a key written by O_1, we have $O_1 \prec O_2$, 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 $O_1 \rightarrow O_2$, we have $O_1 \prec O_2$.

We then formalize the two essential yet straightforward requirements for 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 $\alpha \rightarrow \beta$ by considering $\alpha \rightarrow send(m_1) \rightarrow recv(m_1) \rightarrow \beta$. 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 $\alpha' \rightarrow \beta$ 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 $\beta \rightarrow e$ 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 $e_i \rightarrow \beta \rightarrow e$. 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 $\beta \rightarrow send(m_2) \rightarrow e$), guaranteeing A only triggers e after β in real-time order, as the construction below: since $\alpha' \rightarrow \beta \rightarrow send(m_2) \rightarrow e \rightarrow recv(m_2)$, for any event $\alpha' \rightarrow \beta$, we have $\alpha' \rightarrow m_3$. 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 $R(A) \rightarrow Q(B)$ 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.

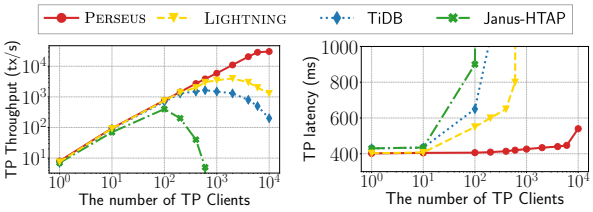
C. CASE STUDY ON SAFE TIMESTAMPS

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 site maintains a per-site watermark by tracking all intra-site nodes, meaning that this site will coordinate no new transaction with a smaller timestamp than the per-site watermark. Sites exchange their per-site watermarks periodically, and the minimum per-site watermark is the global watermark and transactions with timestamps smaller than the global watermark can be executed.

However, as watermarks are exchanged asynchronously among sites, each site's view of the global watermark is *not tight*, not containing all finished transactions. Therefore, to ensure read-finished-writes, even a site-local query has to be assigned a timestamp equivalent to its per-site watermark, larger than this site's view of the global watermark, and to wait for this site'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 sites differently. We leave these exciting developments as our future work.