K2: On Optimizing Distributed Transactions in a Multi-region Data Store with TrueTime Clocks

Haoze Song[#]
The University of Hong Kong
Hong Kong SAR, China
hzsong@cs.hku.hk

Yongqi Wang[#]
Shanghai Jiao Tong University
Shanghai, China
wangyongqi7@sjtu.edu.cn

Xusheng Chen[*]
Hao Feng
Huawei Cloud, China
{chenxusheng6,feng.hao1}@huawei.com

Yazhi Feng
Xieyun Fang
Huawei Cloud, China
{fengyazhi,fangxieyun}@huawei.com

Heming Cui[*]
The University of Hong Kong
Shanghai AI Laboratory
heming@cs.hku.hk

Linghe Kong
Shanghai Jiao Tong University
Shanghai, China
linghe.kong@sjtu.edu.cn

ABSTRACT

TrueTime clocks (TTCs) that offer accurate and reliable time within limited uncertainty bounds have been increasingly implemented in many clouds. Multi-region data stores that seek decentralized synchronization for high performance represent an ideal application of TTC. However, the co-designs between the two often failed to realize their full potential. This paper proposes K2, a multi-region data store that explores the opportunity of using TTC for distributed transactions. Compared to its pioneer, Google Spanner, K2 augments TTC's semantics in three core design pillars. First, K2 carries a new timestamp-generating scheme that is capable of providing a small time uncertainty bound at scale. Second, K2 revitalizes existing multi-version timestamp-ordered concurrency control to realize multi-version properties for read-write transactions. Third, K2 introduces a new TTC-based visibility control protocol that provides efficient reads at replicas. Our evaluation shows that, K2 achieves an order of magnitude higher transaction throughput relative to other geo-distributed transaction protocols while ensuring a lower visibility delay at asynchronous replicas.

PVLDB Reference Format:
Haoze Song, Yongqi Wang, Xusheng Chen, Hao Feng, Yazhi Feng, Xieyun Fang, Heming Cui, and Linghe Kong. K2: On Optimizing Distributed Transactions in a Multi-region Data Store with TrueTime Clocks. PVLDB, 18(6): 1756 - 1769, 2025. doi:10.14778/3725688.3725704

1 INTRODUCTION

Recent years have witnessed the growing popularity of multi-region data stores. Distributed transactions and strong consistency (e.g., strict serializability) have been advocated in these stores for their powerful *ACID* semantics and their ability to provide an excellent abstraction of programming in a single-threaded model. However, their performance is often criticized due to the high cost of transaction coordination across geographic distances. Fortunately, TrueTime clocks (TTC) have emerged and been deployed by multiple cloud providers [6, 19, 30, 36]. TTC offers precise timestamps for distributed events with a constrained uncertainty bound. Thus, we can identify the orders of two independent events without coordination if their timestamps (and uncertainty bound) do not overlap.

Our paper tries to answer a natural question: can we achieve high performance for strictly serializable transactions (i.e., providing ACID semantics with the strongest consistency guarantees) in a multi-region data store with TTCs? To answer this question, we present the design and implementation of K2, a multi-region data store that leverages TTC to run transactions efficiently.

To our knowledge, prior to K2, Google Spanner [6] was the first global-scale data store that leverages TTC to optimize geo-distributed transactions. Specifically, Spanner uses TTC to implement its multi-version data store, providing non-blocking snapshot reads based on TTC timestamps. Technically, Spanner coordinates read-write transactions via a classical two-phase locking protocol and commits transactions using two-phase commits. To eliminate the coordination for read-only transactions, Spanner implements *TrueTime API* to assign a commit timestamp for each read-write transaction and uses this timestamp to index generated data versions. By doing so, Spanner can execute each read-only transaction on a specific snapshot by directly comparing its start timestamp to the timestamps of data versions. As the comparison is fully decentralized and the order between transactions strictly follows the order posed by *TrueTime API*, Spanner achieves strict serializability (a.k.a external consistency or linearizability) with moderate cost.

Compared to Spanner, K2 takes a step further and augments the use of TTC in three core design pillars. The first innovation (❶) is that K2 introduces a new batch scheme for timestamp generation. A strawman approach (used in Spanner) for implementing *TrueTime API* lets each machine regularly calibrate its local clock with an accurate time source (called time master). Thus, each machine can generate high-precision timestamps and calculate their corresponding time uncertainty bounds based on local clocks. However, maintaining TTC on each machine can be inefficient and expensive since ordinary servers (equipped with quartz clocks) can have a big clock drift (e.g., 200ppm), and the clock calibration frequency should be configured in a low setting (e.g., 30s in Spanner) due to limited synchronous channels of time master. As a result, it can contribute to a large uncertainty bound (e.g., 7ms in Spanner).

Instead of equipping TTC to all machines, K2 implements a TTC Oracle, which consists of a cluster of PTP servers [11] in each data

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725704

[#]The authors contribute equally to this paper. [*]Corresponding authors.

center. A PTP server can have a small time uncertainty bound by frequently calibrating its clock with the time master, resulting in microseconds (μs) clock accuracy [28]. This is enabled by two factors: first, maintaining a small TTC Oracle cluster size mitigates the synchronous channel bottleneck; second, a new hardware clock is deployed on each PTP server's network card so that most latency-sensitive paths for clock calibration are in hardware.

K2 features a new timestamp batching algorithm (called K2-TB) to allocate timestamps for ordinary servers from TTC Oracle. Specifically, with K2-TB, once an ordinary server receives a timestamp from remote TTC Oracle servers, it derives a batch of timestamps and acts as a proxy for issuing timestamps to transactions. K2-TB introduces two benefits: (1) it prevents most requests from getting timestamps remotely, reducing timestamp requesting latency; (2) it relieves the workloads of TTC Oracle. A small TTC Oracle cluster is sufficient to support large-scale workloads, benefiting clock synchronization and timestamps' scalability.

For correctness, each batch has a limited time to live (TTL) to avoid generating significantly old timestamps. Since multiple ordinary servers can hold batches with overlapped windows and issue timestamps concurrently, K2 uses a new timestamp synthesizing method and restricts minimal transaction latency by commit wait. We formally analyze the relationship between TTL and commit wait time (CWT) in §4.3. In particular, CWT is commonly used in TTC-based transaction systems [6, 38] to provide consistency guarantees by enforcing transactions to wait out time-uncertainty bound before committing. We revitalize this mechanism in K2-TB. K2-TB does not significantly amplify CWT. A small batch size is sufficient for realizing high performance. On the contrary, thanks to the design of TTC Oracle, K2-TB can provide efficient timestamping service with small uncertainty bounds at scale.

The second innovation of K2 (❷) is leveraging TTC to realize multi-version features for read-write transactions. K2 redesigns the existing multi-version timestamp ordering (MVTO) protocol to fully leverage TTC's capabilities and further optimizes the protocols for multi-regions. Compared to Spanner, K2-MVTO tolerates more concurrency on read-write transactions by allowing readers to make progress regardless of concurrent writers. Specifically, in K2-MVTO, a transaction gets its timestamp at the beginning, uses it for ordering, and applies it as the index of committed data versions. Since start timestamps have already established a total order between transactions, K2-MVTO eliminates the need to use commit timestamps. By doing so, K2-MVTO fundamentally avoids write-write conflicts and carries on a deadlock-free design (§5.2).

The third design innovation (❸) is a new TTC-guided visibility control protocol (called K2-VCR). Multi-region data stores typically deploy cross-region replicas asynchronously to facilitate near-client reads. For instance, Spanner implements both stale and strong (linearizable) reads on its read-only replicas [5]. To ensure consistency, Spanner introduces a safe timestamp mechanism, which guarantees all transactions with a commit timestamp smaller than the safe timestamp have been committed and replicated. Using safe timestamps, a straightforward method for serving reads at replicas is to have a read-only transaction wait until the safe timestamp exceeds the read timestamp. However, increasing safe timestamps becomes inefficient if a transaction is prepared on the primary nodes but not committed, because Spanner cannot determine if the transaction will eventually be committed or not and does not know its specific commit timestamp, either. As a result, Spanner can introduce a large visibility delay at replicas. Generally, Spanner suggests users use a minimum staleness of $10s$ to benefit from non-blocking reads [9].

K2-VCR addresses this issue by co-designing with K2-MVTO. In particular, K2-VCR lets safe timestamps grow in a staggered approach (called epochs). Epochs are divided on each node independently by TTC time. An epoch's interval is manually configurable, TTC-based, and will not be dragged by long-running transactions. The tricky is that, as K2 has eliminated commit timestamps and does not use commit timestamps to index data versions, a slow commit transaction (e.g., prepared but not committed) can always calculate its epoch collaboratively between the data nodes and coordinator as long as its provisional results are not visible before commit. Based on such an observation, K2-VCR conducts a fine-grained visibility control to consistently assign epochs to read-write transactions and then use the generated epochs to serve reads at replica (§6.5).

Contribution. Our main contribution is K2, a multi-region data store that supports efficient distributed transactions over TTC. We deeply exploit the opportunity and performance benefits of using TTC in distributed transactions. We implement K2 and compare it with state-of-the-art designs that are used in advanced multi-region data stores. Our evaluation shows:

- K2-TB is efficient and scalable. K2-TB has the capacity to issue 10^5 timestamps per second per core while ensuring a relatively small uncertainty bound (e.g., $400\mu s$).
- K2-MVTO achieves an order of magnitude better performance than other TTC-based transaction protocols (i.e., D2PL-TTC) and outperforms other MVTO-based works (e.g., MVTO-HLC) by 2.32× in TPC-C while ensuring stronger consistency.
- K2-VCR can achieve a low visibility delay under a typical deployment without artificially aborting transactions.

2 BACKGROUND AND MOTIVATION

2.1 Why we need TrueTime Clocks?

Versioning is one of the fundamental techniques for concurrency control. In OLTP, multi-versioning enables read-only transactions to access older tuple versions, while read-write transactions can simultaneously create new ones. Time is widely used for versioning. We classify existing approaches into three categories.

Using TrueTime for Versioning. TrueTime provides strict monotonicity across a whole system. Existing works use commit timestamps for versioning. To preserve real-time order over uncertainty bounds, transaction protocols should introduce a commit wait to distinguish two overlapped transactions. For instance, Spanner [6] uses *TrueTime API* to assign a commit timestamp to each transaction and wait out uncertainty-bound in the commit phase so that a committed data version is always indexed with a timestamp that has gone into the past. Compared to Spanner, K2 improves the use of TTC by proposing a new timestamp generation scheme (§4) and optimizing its transaction (§5) and snapshot protocols (§6).

Using Logical Time for Versioning. A straightforward solution for generating logical timestamps is through a timestamp oracle (TSO). TSO is widely adopted for simplicity and manageability [2, 17, 33, 37, 47]. With TSO, time is determined through a centralized clock cluster. Each transaction has a start timestamp

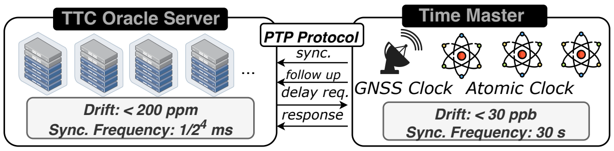


Figure 1: K2's TTC Oracle Architecture.

as its transaction ID to sort all transactions in a globally consistent order. After that, transactions are executed in an order-before-execution manner. Nevertheless, TSO incurs two significant flaws, which are extremely severe in multi-region deployments: first, obtaining timestamps from a remote region will cause large network latency (e.g., 50ms); second, centralized clocks face fault-tolerant challenges. To achieve high availability, a typical design backs up its TSO with consensus (e.g., Paxos [20]). Then, the TSO cluster cannot serve any timestamps during the re-election period when failures are suspected, which can be long due to the cross-region communication being involved.

Distributed logical clocks are proposed to overcome the problem of centralized TSO. For instance, Lamport Clock [22] lets each node maintain a counter that is incremented whenever a local event occurs. When a message is sent, the counter value is included in the message. The receiving node then updates its counter to be the maximum of its current counter and the counter received in the message. This ensures that the order of events is consistent across all nodes. Implementing Lamport Clock (or its variants, e.g., vector clocks [33]) for distributed transactions provide causality but not strict serializability (or linearizability), which can be a limitation for time-sensitive applications [28].

Deterministic concurrency control [3, 12, 14, 16, 26, 27, 29, 34, 44, 50], which orders transactions before executions and lets transactions execute in a deterministic order, can be regarded as a special subcategory that uses logical time for versioning. For instance, Calvin [44] implements a sequencing layer to intercept transactional inputs and places them into a global sequence. Detock [29] deterministically constructs a dependency graph to track partial order between transactions [27]. However, all recent proposals have limited or no support for interactive transactions, thereby preventing their use in many industrial deployments.

Using Hybrid Time for Versioning. A Hybrid Logical Clock (HLC) combines physical and logical clocks, providing a timestamping mechanism that captures both the ordering of events (like logical clocks) and a coarse-grained actual time (like physical clocks). Technically, HLC provides causality tracking through its logical component to provide strict monotonicity and can provide snapshot reads (with bounded staleness) through its physical component.

In recent years, HLC has been considered in various data stores. For instance, CockroachDB [41] uses HLC to generate both start and commit transaction timestamps and uses the commit timestamps to index data versions. DST [46] proposes a hybrid timestamp mechanism to support snapshot reads with bounded staleness. However, as HLC still relies on logical components to track causality (i.e., the same as distributed logical clock), it's difficult for HLC-based designs [3, 10, 41, 43, 46, 48, 49] to achieve linearizability at a moderate cost. As a result, many existing works opt to support weaker consistency. For instance, CockroachDB implements single-key linearizability, i.e., a consistency model weaker than linearizability.

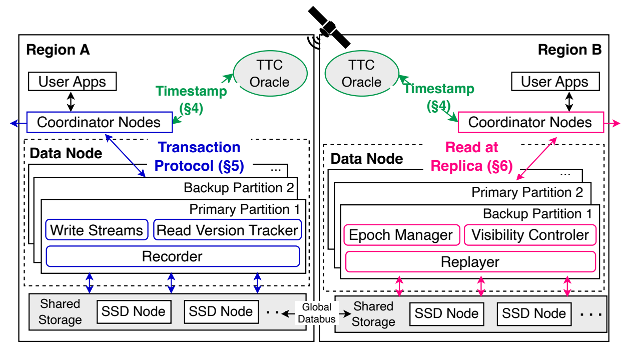


Figure 2: K2's system architecture.

CockroachDB does not guarantee that the ordering of transactions touching disjoint key sets will match their ordering in real-time.

Takeaways. Both logical and hybrid time have some limitations in supporting multi-versioning, especially when linearizability is required. TrueTime offers a promising solution for strongly-consistent versioning in geo-distributed transactions at a moderate cost.

2.2 How to deploy TrueTime Clocks?

To synchronize with physical time, TrueTime Clocks (TTCs) typically rely on a global navigation satellite system (GNSS) to provide the time synchronization source and use atomic clocks for high availability. We call such a fault-tolerance time source a time master. Each data center has a time master to synchronize ordinary clocks inside the data center. As clocks on servers always drift over time, two major factors impact the size of the time uncertainty bound (ϵ): synchronization frequency ($\mathcal{F}$) and maximum drift rate ($\mathcal{D}$). An ϵ can be calculated as follows, where ϵ_{base} is a small constant (a few nanoseconds) that accounts for noise [24]: $\epsilon = \epsilon_{base} + \mathcal{F} \times \mathcal{D}$.

Spanner [6] sets the synchronization frequency as 30s and assumes a maximum drift rate as $200\mu s/s$ (i.e., 200ppm), contributing to 7ms in ϵ. After Spanner, many works have been proposed to optimize ϵ by either improving synchronization methods or squeezing the maximum drift rate assumption. For instance, the precision time protocol (PTP) clock [11] introduces a new hardware clock on the network card. The network card can capture synchronization packets upon arrival to align the PTP clock with the server. It eliminates the inaccuracy introduced by software jitter and improves standard synchronization frequency. DTP [23] further modifies the Ethernet physical layer to facilitate message exchanges at the microsecond frequency. Sundial [24] leverages specialized hardware that synchronizes every 100μs and performs fast failure detection. Graham [28] uses a learning method to characterize computer clocks automatically and uses the characterization to reduce the maximum assumed drift from $200\mu s/s$ to $100ns/s$.

Our Solution. In K2, we choose to deploy a cluster of PTP servers as TTC Oracle in each data center and let ordinary servers request timestamps from TTC Oracle. We do not directly equip each server with TTC for three practical reasons. First, while PTP servers have been available in major clouds [19, 30, 36], ordinary servers (that do not have a hardware clock on the network card) remain. Upgrading all NICs for existing machines is neither easy nor cost-efficient. Requesting a small size of PTP cluster reduces the deployment cost and improves the applicability of K2. Second, a small PTP cluster mitigates the bottleneck caused by the limited synchronization channels on the time master. Generally, a time master with fewer active channels can support a higher synchronization frequency. Typically, the synchronization frequency ($\mathcal{F}$) of a time master is inversely proportional to the number of active channels [18]. Third, a stand-alone PTP cluster simplifies fault tolerance and further reduces software and network jitters in clock synchronization.

We show an overview of our solution in Figure 1. In particular, we use the BeiDou Navigation Satellite System (BDS) as a reliable and accurate timing resource. TTCs are synchronized with time master (a BITS clock [18]) using PTP to deliver precise timestamps with a small uncertainty bound. Like Spanner, we assume a conservative value for the maximum clock drift rate (i.e., $200\mu s/s$) to tolerate frequency variations across a wide range of temperatures. TTC Oracle servers are synchronized with the BITS clocks at a high frequency ($1/2^4$). Consequently, we have an uncertainty bound (ϵ) for timestamps generated by TTC Oracle as $100\mu s$, which tolerates eight loss of synchronization message (i.e., $8 \times 1/2^4 s \times 200\mu s/s$). If more synchronization messages are lost, the affected TTC server becomes temporarily unavailable via a timeout mechanism.

Takeaways. Many advanced clock synchronization methods can achieve a small uncertainty bound (e.g., sub-microseconds). TTC Oracle is a cost-efficient approach for deploying TrueTime clocks.

3 SYSTEM MODEL AND ARCHITECTURE

Figure 2 shows K2's architecture over two regions A and B. Basically, K2 shares a similar high-level architecture with other state-of-the-art multi-region data store systems (e.g., Spanner and CockroachDB). The system is deployed across multiple geographic regions. A region comprises servers linked through a low-latency network, generally located within one data center or across several data centers situated near one another. The network between different servers (either intra-region or cross-region) is asynchronous: packets can be dropped, reordered, or delayed. Ordinary servers (i.e., coordinator nodes and data nodes) do not have synchronized clocks. A stand-alone server cluster (i.e., TTC Oracle, Figure 2) is equipped with a TrueTime clock by frequently calibrating time with timing sources (i.e., a BITS clock, §2.2).

As a distributed database, K2 divides data into multiple partitions, with each partition having a primary copy assigned to one region. Partitions are handled by data nodes. We call a data node holding primary partitions as a primary data node; otherwise, we call it a replica node. K2 adopts a cloud-native architecture. An underlying shared storage layer handles data persistence and intra-region fault tolerance. Cross-region replica data nodes are materialized by replaying logs from primary data nodes.

To execute a transaction, a user can send the transaction to its closest region. The first server that receives a transaction becomes its *coordinator*. Then, the coordinator requests a start timestamp using our timestamp algorithm (called K2-TB, see §4) and uses this timestamp to resolve transaction conflicts and versioning data. K2 executes read-write transactions over primary partitions using our new transaction protocol (called K2-MVTO, see §5), while read-only transactions can be executed over both primary and replica nodes. When reading from replica nodes, K2 uses a new visibility control algorithm for efficiency and correctness (called K2-VCR, see §6).

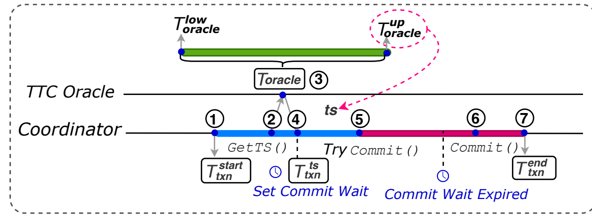


Figure 3: A strawman approach for timestamp generation.

Consistency and Isolation Model. K2 supports general transactions with strict serializability [6, 15, 32], which implies serializability for isolation and linearizability for real-time order. Intuitively, strict serializability provides a total order for all transactions with respect to the wall-clock time order. Therefore, if a transaction txn_1 ends (i.e., the wall-clock time for notifying the results of txn_1 to user apps) before txn_2 starts (i.e., the wall-clock time for notifying the recipient of txn_2 to user apps), then txn_1 must appear before txn_2 in the total order. Consequently, transactions seem to be processed individually in the sequence they are received [25].

4 TIMESTAMP DESIGN

This section first identifies the properties required by correctness (§4.1). Next, we present a strawman approach that ensures these properties (§4.2). Then, we analyze the problem of the strawman approach and motivate K2-TB (§4.3). Finally, we show K2-TB still satisfies the correctness of the invariants (§4.4).

4.1 Required Correctness Invariants

As motivated in §2.1, K2 uses TTC timestamps for transaction ordering and data versioning. Like Spanner, to achieve strict serializability, K2 needs to assign globally meaningful timestamps to transactions. These timestamps should represent a serialization order of transactions, and the serialization order must correspond to their real-time order. Formally, we should ensure an invariant:

INVARIANT 4.1 ($txn_1 \xrightarrow{rto} txn_2 \Rightarrow ts_1 < ts_2$). *If a transaction txn_1 commits before another transaction txn_2 starts in wall-clock time, denoted as $txn_1 \xrightarrow{rto} txn_2$. Then, we have the timestamp of txn_1 smaller than that of txn_2, denoted as $ts_1 < ts_2$.*

This invariant can be safeguarded by enforcing that the given timestamp of a transaction is always within its lifetime so that the given timestamp reserves the order relationship between transactions. We take the notation from [6] and use the function $abs(e)$ to denote the wall-clock time of an event e. The function is used throughout our paper. Formally, considering the property below:

PROPERTY 4.1. *For any transaction that has a timestamp ts, we must $abs(T^{start}_{txn}) < ts < abs(T^{end}_{txn})$.*

PROOF. According to the definition of real-time order, given $txn_1 \xrightarrow{rto} txn_2$, we have $abs(T^{end}_{txn_1}) < abs(T^{start}_{txn_2})$. Then, Property 4.1 suggests $ts_1 < abs(T^{end}_{txn_1}) <$ and $abs(T^{start}_{txn_2}) < ts_2$. According to transitivity, finally, we have $ts_1 < ts_2$. □

Takeaways. When assigning timestamps to transactions, Property 4.1 should be preserved, and Property 4.1 is sufficient to achieve strict serializability based on these timestamps.

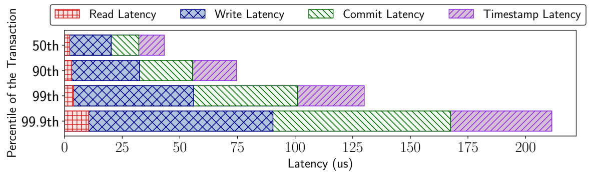


Figure 4: Latency breakdown of YCSB-T (Strawman approach).

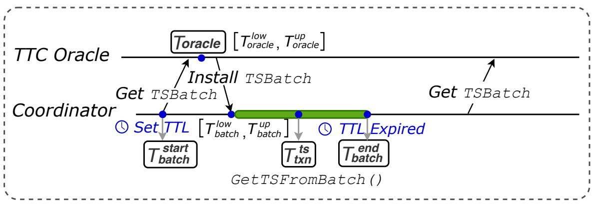


Figure 5: K2's timestamp batching workflow.

4.2 A Strawman Approach

When using TTC Oracle for assigning timestamps (§2.2), ordinary servers (e.g., coordinators, Figure 2) should request timestamps from TTC Oracle server through an intra-data-center network. A simple approach is for each coordinator to request a timestamp as needed. We show a message flow of such an approach in Figure 3.

In Figure 3, the coordinator initiates a new transaction at ① and obtains a new timestamp for it at ②. When a TTC Oracle server receives the request at ③, the server generates a timestamp with an error bound. It guarantees $T^{low}_{oracle} \leq abs(T_{oracle}) \leq T^{up}_{oracle}$, where $abs(T_{oracle})$ is the wall-clock time for generating timestamps. To ensure the transaction's timestamp is larger than ① (i.e., $abs(T^{start}_{txn}) < ts$, the left part of Property 4.1), the strawman approach picks the uncertainty upper bound as the timestamp (i.e., $ts = T^{up}_{oracle}$). Then, as ① must happen before the time for generating timestamps, we have $abs(T^{start}_{txn}) < abs(T_{oracle}) \leq T^{up}_{oracle} = ts$.

To guarantee the wall-clock commit time is larger than the transaction's timestamp (i.e., $ts < abs(T^{end}_{txn})$, the right part of Property 4.1), we should set a minimal transaction latency and let the transaction wait for a specific period to enforce $T^{up}_{oracle} = ts < abs(T^{end}_{txn})$. In Spanner, commit wait is achieved by comparing the ordinary server's local time with the transactions' timestamps. If the difference is smaller than the uncertainty bound, Spanner enforces the transaction to wait. However, K2 does not equip ordinary servers with TTC. The tricky is that even though clocks on ordinary servers can not tell a precise time, they are good enough to count a small time interval (even taking clock drift into consideration). Therefore, as shown in Figure 3, we let a local coordinator set a commit wait time (CWT) period after getting a timestamp at ④ and check whether the period has expired before committing the transaction at ⑤. CWT is calculated as $2\epsilon \times (1 + \mathcal{D})$, considering clock drift on ordinary servers. If the check fails, we block the transaction until CWT expires. Finally, the coordinator can commit the transaction at ⑥ and end the transaction at ⑦.

Using commit wait, we have $abs(T^{ts}_{txn}) + CWT < abs(T^{end}_{txn})$. According to causality, the time for generating a timestamp must happen before the time for receiving the timestamp. Thus, we have $abs(T_{oracle}) < abs(T^{ts}_{txn})$. Finally, $ts = T^{up}_{oracle} \leq abs(T_{oracle}) + 2\epsilon < abs(T^{ts}_{txn}) + CWT < abs(T^{end}_{txn})$, satisfying Property 4.1.

K2 Timestamp Batch

Data Structure:

```
struct TsBatch {
  uint64_t TBBase          ▶ Batch base, assign TBBase = T^up_oracle+TTL
  uint16_t OracleID        ▶ TTC Oracle Server ID
  uint16_t TSCount         ▶ Number of ts can be generated
  uint8_t  TBNanoSecStep   ▶ Step to skip between two ts
  uint8_t  TTLus           ▶ TTL of batch (pre-configured)
  bool     TTLExpired }    ▶ Verification Flag
```

Algorithm:

```
GetTsFromBatch(batch):
  Assert (Not TTLExpired && usedCount < batch.TSCount)
  NanoSecAdjust = usedCount * batch.TBENanoSecStep
  ts(batch.TBEBase + NanoSecAdjust, batch.OracleID)
  return ts
```

Figure 6: K2's timestamp batching algorithm.

4.3 Timestamp Batching Algorithm

Optimization Opportunity. Compared to Spanner's approach (i.e., deploying TTC on all ordinary nodes), our strawman approach incurs additional overhead for timestamp assignment, requiring each transaction to make an extra intra-data-center round trip. In many real-world workloads, most transactions operate data in the same data center or even on the same local partitions [1, 7, 46]. Additional network round trips will notably lengthen the critical section of transactions and further ***increase the chance of conflicts***, causing extra transaction aborts or blocking time. It is reported in [46] that the overhead for getting a timestamp can lead to 10%∼30% throughput drop, depending on transactions' execution time.

In Figure 4, we show a latency breakdown for YCSB-T transactions that access data in a single region. Timestamp latency contributes 20.1%∼25.4% to the total latency. Therefore, reducing latency for assigning timestamps is non-trivial [12]. It helps minimize the contention footprint and thus benefits end-to-end throughput.

Furthermore, a timestamp server has a limited throughput. Our experiments show a TTC Oracle server can stably generate ∼60k timestamps per second. To support large-scale workloads (e.g., 300k transactions per second in Figure 14), TTC Oracle would need to scale out, incurring additional costs in clock synchronization.

Timestamp Batching. To minimize remote requests and improve the throughput of timestamp generation, K2 introduces a timestamp batching algorithm. Figure 5 shows K2's timestamp batching scheme. Before getting a timestamped batch from TTC Oracle, the coordinator set a time-to-live (TTL). Typically, TTL is set in hundreds of microseconds (e.g., 100μs). For availability, TTL should be larger than the latency for obtaining a single timestamp from TTC Oracle (18μs on average in our experiments, §7.2). Note that violating this requirement does not harm correctness but results in getting an expired batch. Then, the coordinator requests a timestamp from TTC Oracle via TSBatch. After receiving the timestamp, the coordinator works as a proxy to assign timestamps for the following timestamp requests before the TTL expires. The coordination calculates a series of timestamps from the base timestamp. After TTL expires, the timestamp batch becomes invalid, and the coordinator should request a timestamp from the TTC Oracle again.

In particular, the timestamp batch in K2 is structured as Figure 6. We set the base timestamp for the batch as $T^{up}_{oracle}+TTL$ to guarantee the given timestamp is always bigger than the wall-clock start time of the transaction. The step between two timestamps in a batch is suggested by TBNanoSecStep, which is in nanoseconds. For simplicity, we set the allowed batch size to the same as the

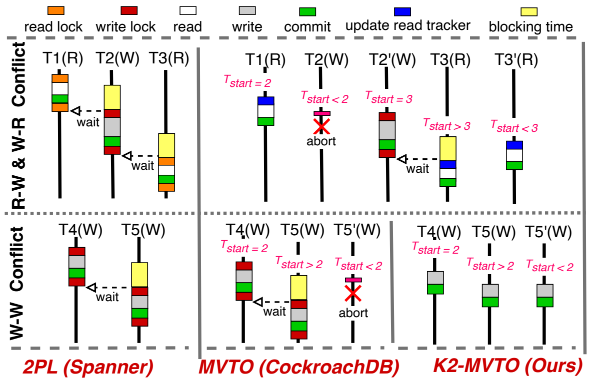


Figure 7: A comparison of 2PL, MVTO, and K2-MVTO. 2PL causes blocks in all cases. MVTO and K2-MVTO optimize R-W and W-R conflict by allowing the reader to access the old version. K2-MVTO further avoids W-W conflict by allowing multiple temporary writes.

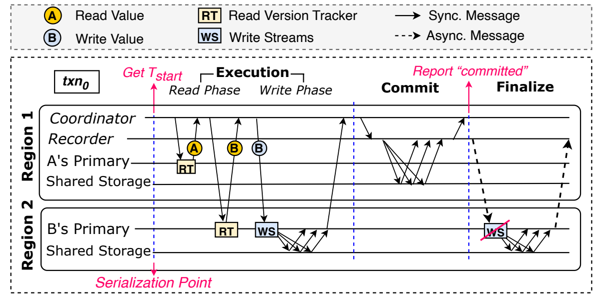


Figure 8: K2-MVTO's transaction procedure.

value of TTL, which can be configured in practice. Then, we have $T^{low}_{batch} = T^{up}_{oracle} + TTL$ and $T^{up}_{batch} = T^{up}_{oracle} + 2 * TTL$. Thus, the number of timestamps that can be generated from the batch (i.e., TSCount) can be calculated by TTL/TBNanoSecStep.

To use a timestamp batch, we first check whether the TTL has not expired and whether the timestamps in the batch have not been used up. If the check succeeds, the coordinator generates a timestamp by adjusting nanoseconds (Line 2-3, Figure 6).

In a rare case, two different coordinators may obtain the same timestamp batches from different TTC Oracles. To eliminate the exactly same timestamps, K2 pairs each timestamp with the Oracle server ID that generates it. If two timestamps are identical, their Oracle server IDs are compared. These server IDs do not influence correctness (since we do not use them to ensure Property 4.1) but facilitate timestamp comparison in our transaction protocol.

4.4 Correctness and Performance Discussion

The correctness intuition is similar to our strawman approach: timestamps generated from the batch are synthesized with time upper bounds ($T^{up}_{oracle} + TTL$), and each transaction waits for its batch's time uncertainty to elapse. In particular, the coordinator should set CWT as $2 \times (TTL + \epsilon) \times (1 + \mathcal{D})$ for the transaction after getting its timestamp from the batch, where T^{end}_{batch} is the expired time of the batch. One may note that CWT can be amplified by batch size (TTL). However, we find a small batch size is sufficient for improving performance since we adjust timestamps from a batch in nanoseconds. For example, if we use $10ns$ as a step and $100\mu s$ for TTL, we can generate a maximum of $10,000$ timestamps from a single batch. Moreover, CWT does not extend a transaction's conflict window, as conflicts are resolved prior to the waiting phase. This distinguishes it from the latency penalty incurred when obtaining timestamps from a TTC Oracle server. We formally prove Property 4.1 still holds with K2-TB in our technical report [40].

5 TRANSACTION PROTOCOL

K2's transaction protocol (K2-MVTO) is a variant of multi-version timestamp-ordering transaction protocol. We choose MVTO as a basic protocol since it imposes the minimal execution constraints required to ensure serializability [4]. Similar to CockroachDB, K2-MVTO implements one-phase commits to finalize temporary writes asynchronously, reducing the commit latency in multi-region environments. In addition to using TrueTime in CockroachDB's protocol, K2-MVTO introduces two optimizations to realize its full potential. First, K2-MVTO assigns each transaction a single timestamp, combining the start timestamp and commit timestamp together. It helps reduce the chance of commit wait time (to be explained in §5.4). Second, based on the first optimization, K2-MVTO implements a write stream that allows multiple temporary writes on a single data tuple, resulting in a write-write conflict-free design.

We compare the methods to solve conflicts among two-phase locking (2PL), MVTO, and K2-MVTO in Figure 7. A lower blocking time and fewer abort cases indicate better performance. 2PL blocks transactions to solve conflicts in all scenarios. Existing MVTO design blocks transactions in write-write (W-W) conflicts since a commit timestamp needs to be determined after the write. Otherwise, an early write may be given a larger timestamp by its coordinator, mismatching the serializable order of writes. K2-MVTO overcomes this restriction by the two aforementioned innovations.

5.1 K2-MVTO Overview

Figure 8 illustrates the procedure of handling a read-write transaction (txn_0). The read set is {A, B}, and the write set is {B}. Like existing MVTO protocol, K2-MVTO assigns each transaction a unique start timestamp that reflects its position in the serialization order. To ensure serializability between transactions, K2-MVTO implements read version trackers (RT) and write streams (WS) to monitor transaction conflicts. RT and WS track information at the key level. RT stores information about the reads being performed on a key, whereas WS records all written intents of ongoing transactions. If a transaction attempts to write with a timestamp smaller than the highest read timestamp on a key, it is aborted to ensure that no read misses a write from an earlier transaction.

To execute txn_0, K2-MVTO consists of three phases: execution, commit, and finalization. During the execution phase, txn_0 reads the value of A and B, updates their RTs, and creates a write intent for B in its WS, according to the start timestamp acquired from TTC. In the commit phase, K2-MVTO commits and persists transaction status in a single round trip. The write intent on B is finalized asynchronously in the next phase by refreshing WS.

Persistence and Cross-region Replication. K2-MVTO stores data in shared storage, consisting of a group of SSD servers. The shared storage appears as an intra-region replication layer of K2. When a node crashes, K2 recovers its states on a new node by duplicating data from the shared storage. Thus far, we ignore cross-region replicas. Existing synchronous or asynchronous protocols

```
K2 Transaction Coordinator Protocol
At Coordinator_i (C_i):
ReceiveNewRWTxn(txn):
  txn.startTS ← GetTSfromBatch
  for each read operation:                      ▶ parallel execution
      Node_i.Read(txn, key)
  for each write operation:                     ▶ parallel execution
      Node_i.Write(txn, key, value)
  Commit(txn)
Commit(txn):
  Recorder.commit(txn, <epoch_i>)
ReceiveNewROTx(tx):
  ReadType:
  | LinearizableRead:
  |    txn.startTS ←GetTSfromLocalBatch
  |    NodeType:
  |    | PrimaryRead:
  |    |     for each read operation:
  |    |         Node_i.Read(txn, key)
  |    | ReplicaRead:
  |    |     for each read operation:
  |    |__       Node_i.RepRead(txn, key)
  | SnapshotRead:
  |    txn.startTS ← A Given Timestamp
  |    for each read operation:
  |__      Node_i.RepRead(txn, key)
```

Figure 9: Coordinator Protocol.

```
K2 Transaction Data Node Protocol
At Primary Data Node_i:
Read(txn, key):
 1: if exist WS[key].ts <= txn.startTS:
 2:     PushWS(key, startTS)                    ▶ resolve undetermined write
 3: else:                                         (Part of Write-Read Conflict)
 4:     update txn.startTS to Read Version Tracker (RT)
 5:     return the latest value version whose ts <= txn.startTS
Write(txn, key, value):
 6: if exist RT[key].ts > txn.startTS:
 7:     Abort(txn)                              ▶ resolve Read-Write conflict
 8: else:
 9:     add txn.startTS, value to Write Streams (WS)
10:     return success, Epoch_i ←GetEpoch()
PushWS(key, txn.startTS):
11: for each w in WS[key]:
12:     ask w's recorder for decision
13:     if abort:
14:         clean and ignore w
15: return the latest value version whose ts <= txn.startTS
Finalize(txn, key, txn.Epoch):
16: persist(txn.startTS,WS[key], Epoch_C)
17: garbageCollect(txn.startTS,WS[key])
At Recorder_i:
Commit(txn, <epoch_i>):
18: Epoch_R ← GetEpoch()
19: Epoch_C ← Max(Epoch_i, Epoch_R)             ▶ Epoch_i is advised by
20: Persist(txn, Epcoh_C)                         Data Node_i in write()
21: for each write operation:
22:     Node_i.Finalize(txn, key, Epoch_C)      ▶ async. finalize
At Replica Data Node_i:
RepRead(txn, key):
23: hold until Replayed Epochs' End Timestamp >= txn.startTS
24: use the Ceiling Epoch of txn.startTS as a read view
25: if exist WS[key].ts <= txn.startTS:
26:     PushWS(key, startTS)
27: else:
28:     return the latest value version whose ts <= txn.startTS
```

Figure 10: Data node Protocol.

(e.g., Paxos [21] or other replication protocols [31, 51]) can be directly applied to the shared storage writing stage of K2-MVTO.

5.2 Processing Phases

We show the pseudocode of K2-MVTO in Figure 9 and Figure 10. Thus far, we can ignore epoch-related code (colored in orange).

Phase 1. Execution. Upon receiving a new read-write transaction, the coordinator first obtains a start timestamp using the algorithm detailed in §4.3. When data nodes receive read and write requests from the coordinator, they perform conflict checks. Without loss of generality, we consider three types of transaction conflicts: write-read, read-write, and write-write conflicts.

A write-read conflict occurs when a transaction reads data that has been modified by another transaction that has not yet been committed. In K2-MVTO, all undetermined writes are organized in a *write stream* (i.e., a series of write intents with data version specified). If the reader's timestamp is smaller than the writer's, the reader can simply skip the writer's data version (Line 5, Figure 10). Conversely, if the reader's timestamp is larger than the writer's, it encounters an undetermined data version in the write stream. In such a case, K2-MVTO allows the reader to actively consult the writer's transaction recorder for the final decision of the write.

In K2-MVTO, transaction recorders are functional modules for recording and persisting the status of transactions (e.g., in progress, aborted, or committed). K2 detaches this module from coordinators to make coordinators stateless. Typically, transaction recorders are deployed on data nodes, and a transaction record is created when the first write of a transaction occurs. To serve the requests PUSHWS, recorders hold the response until the decision of the writes is determined (abort or commit). Finally, when a data node receives a response from the recorder, it can proceed by either reading or skipping the write version (Lines 13-15, Figure 10).

A read-write conflict occurs when a transaction attempts to write data that another transaction has already read. Similar to existing designs, whenever an operation reads a value, K2-MVTO records the reader's timestamp in a read version tracker, indicating the high watermark of values being read. If the writer's timestamp is larger than the reader's, K2-MVTO always creates a new write version in the write streams (Line 9, Figure 10). Conversely, if the writer's timestamp is smaller than the reader's, K2-MVTO aborts the writer's transaction to ensure serializability (Line 7, Figure 10).

A write-write conflict arises when a transaction attempts to write data that another transaction has already written. By implementing write streams and designating write versions with immutable start timestamps, K2-MVTO fundamentally eliminates write-write conflicts by always generating a new write version in the write streams. Creating an old version in the write streams does not compromise serializability if the version is not being read. In K2-MVTO, start timestamps are sufficient to establish a serializable order. Read-write and write-read conflict checks ensure linearizability.

Phase 2. Commit. K2-MVTO commits a transaction in a single round trip if all read/write operations of the transaction succeed. In the commit phase, K2-MVTO persists the determined status of transactions (i.e., commit or abort) via transaction recorders while cleaning up undetermined write streams (that are resolved by commit) later. This approach allows K2-MVTO to commit a transaction without waiting for acknowledgments from all participants, thereby improving user-perceived latency. In particular, in a multi-region deployment, this enables K2-MVTO to commit a transaction with only an intra-region round-trip when the recorders are not synchronously replicated to other regions or cost a single WAN RTT latency when the recorders are synchronously replicated.

Phase 3. Finalize. K2-MVTO issues finalization requests after a decision (commit or abort) is reached for a transaction (see Lines

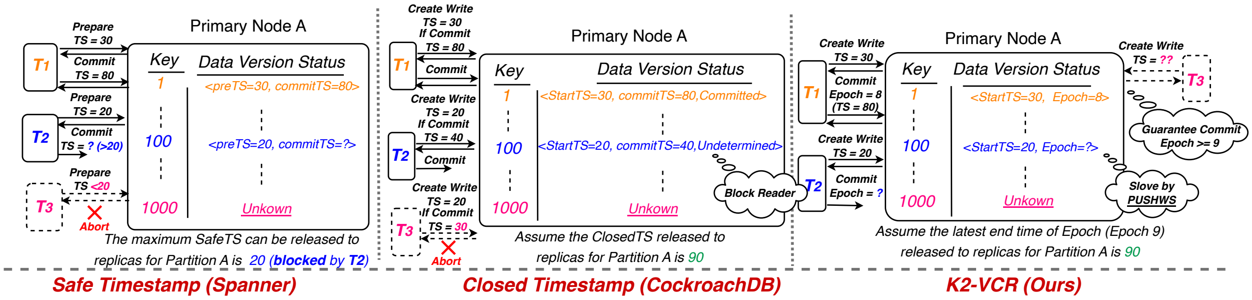


Figure 11: A comparison among safe timestamp, closed timestamp, and K2-VCR. In this example, we assume the epoch length = 10. The safe timestamp can potentially be blocked by a slow commit transaction (T_2) and needs to reject transactions with a small prepare timestamp (T_3). The closed timestamp only blocks readers that access undetermined keys but need to reject transactions that want to create a write with a small commit timestamp (T_3). K2-VCR triggers PUSHWS for readers that access undetermined keys and do not artificially reject any transactions.

9-10 in Figure 9). Upon receiving a finalization request, a data node in K2-MVTO cleans up the corresponding version in the write stream and records the decision in shared storage (see Lines 16-17 in Figure 10). For committed transactions, K2-MVTO creates a permanent data version, while for aborted transactions, the previously persisted version is marked as invisible in shared storage.

5.3 Correctness and Fault-tolerance

Due to space constraints, we discuss the correctness and fault-tolerance mechanisms of K2-MVTO in our technical report [40].

5.4 Design Consideration and Takeaways

A key foundation of K2-MVTO is the use of start timestamps for ordering and versioning, combining the start timestamp and commit timestamp into a single concept. This approach offers four major benefits. First, start timestamps allow for the creation of temporary data versions before committing, and the indexes of these data versions are immutable. This enables K2-MVTO to support early write visibility [13]. Second, start timestamps establish a serializable order prior to execution, ensuring linearizability and avoiding deadlocks. Third, by eliminating commit timestamps, we reduce the chance of commit waits. With start timestamps, the commit wait time (CWT) is timed before execution, while with commit timestamps, CWT is applied during the commit phase. Fourth, using start timestamps for versioning improves replica reads. Data versions can be sent to replicas before committing, as they have already been serialized.

One concern is that using start timestamps for orders could lead to a higher abort rate, as transactions might arrive at data nodes in a different order than their start timestamps. However, K2-MVTO avoids unnecessary aborts by allowing transactions with smaller timestamps to proceed, even if a larger timestamp transaction has already occurred, in cases of write-read and write-write conflicts. K2-MVTO only aborts a transaction when a writer attempts to write a version with a timestamp smaller than the maximum timestamp that has been read (Line 7 in Figure 10). It is important to note that even with commit timestamps and two-phase locking, as used by systems like Spanner, this scenario can still primarily result in aborts due to read locks conflicting with exclusive locks.

6 VISIBILITY CONTROL AT REPLICAS

In this section, we present K2-VCR. When serving reads at replicas, a protocol must ensure the reader works on a snapshot that provides atomicity and strong consistency: (1) the snapshot should obey all ACID properties; (2) the snapshot should be monotonic: the real-time order between two read-write transactions at primaries is preserved at replicas. We first analyze the limitations of existing approaches to motivate our design (§6.1). We then present our solution and finally prove the correctness in §6.5.

6.1 Limitation of Existing Solutions

To correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read, Spanner [5] implements safe timestamps, up to which all reads are guaranteed to be consistent: no transactions can commit with a commit timestamp below the safe timestamp. Thus, a read can be served at a replica if its read timestamp is below the safe timestamp; otherwise, it must be blocked until the safe timestamp becomes larger. Thus, keeping safe timestamps advancing efficiently is critical to shorten the blocking time and improve freshness. Generally, safe timestamps continuously grow since new transactions will likely commit at a bigger timestamp. However, if undetermined transactions exist (that have been prepared but not committed at primaries), safe timestamps should slow down to wait for the final decision on these transactions. We show an example in Figure 11. Assume node A has keys ranging from 1 to 1000, and there are two ongoing transactions, T_1 and T_2. T_2, which updates key 100, has been prepared with timestamp 20 but has not committed. T_1, which updates key 1, commits with timestamp 80. In such a case, the maximum safe timestamp that can be released by node A is 20 since T_2's commit timestamp is calculated by its coordinator (unknown by node A) and can be any value larger than 20. When releasing a safe timestamp of 20, node A also promises that it will not accept any prepare with a timestamp smaller than 20 in the future, so all later transactions (e.g., T_3) must be prepared and then committed with a timestamp larger than 20.

Limitation 1: A slow transaction may block safe timestamps, even if a transaction has already been committed with a larger timestamp. Slow transactions are common in multi-region databases, as a cross-region transaction incurs WAN latency to commit.

Limitation 2: A prepare request with a timestamp smaller than the safe timestamp will be rejected (e.g., T_3).

CockroachDB [41] implements a mechanism known as closed timestamps based on its transaction protocol. A closed timestamp guarantees that transactions whose commit timestamp is smaller than the closed timestamp **cannot** create create new writes. This can be realized in CockroachDB since it uses HLC (§2.1) to assign

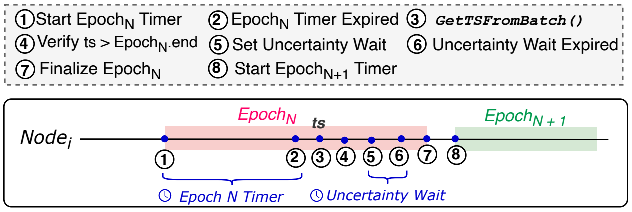


Figure 12: Epoch generation on nodes.

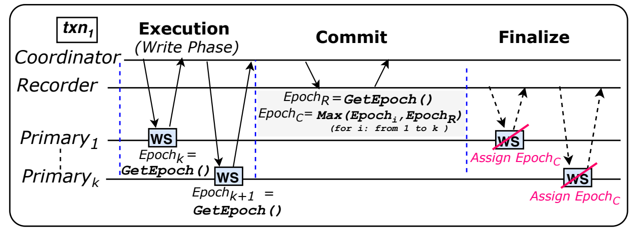


Figure 13: Consistent epoch assignment. We omit the read phase of K2-MVTO because it does not contribute to data visibility on replicas.

a commit timestamp for each transaction at the beginning. Then, CockroachDB can advance its closed timestamp at a fixed interval, even with a slow commit transaction (T_2), since its commit timestamp has been pre-calculated. The slow commit transaction (T_2) may block replica readers who want to access the undetermined keys but will not affect others. For instance, if a reader accesses key 100 with a read timestamp 90, it will be blocked and must wait for the final decision of T_2. In contrast, if a reader accesses key 1 with a read timestamp 90, it will not be blocked and can read the latest version made by T_1. The correctness guarantee is that the replica reader whose read timestamp is ≤ the closed timestamp will always see all writes whose commit timestamps are below the closed timestamp, even if their final decision (commit or abort) has not been made. However, this fixed interval unavoidably forces transactions with commit timestamps smaller than the latest closed timestamp to abort (e.g., T_3). Although CockroachDB can potentially increase the commit timestamps of a transaction based on HLC to seek another commit opportunity, a "read refresh" is required to maintain serializability. This involves checking all prior read operations to ensure that no writes occurred between the original and the increased timestamps. The check itself can be costly, and if it fails, the transaction is aborted. For long-running transactions involving popular keys, "abort" can be a common outcome. To avoid a high abort rate, the interval of a closed timestamp is recommended to be configured as a large value (e.g., 3s) in CockroachDB.

Limitation: A transaction whose pre-calculated commit timestamp is smaller than closed timestamp will be rejected. To avoid a high abort rate, the interval of a closed timestamp is set to a large value.

6.2 K2-VCR Overview

K2-VCR fundamentally overcomes the limitations of existing solutions by introducing a new TrueTime-based epoch approach. Since K2 has eliminated commit timestamps in its transaction protocol, epochs in K2-VCR serve two functionalities: (1). attaching a read view to each transaction (similar to commit timestamps); (2). suggesting visibility for a replica reader (similar to safe/closed timestamps). By doing so, K2-VCR subtly integrates the commit version assignment with the replica's visibility control algorithm.

The guarantee of K2-VCR is similar to a closed timestamp: transactions whose commit epoch is smaller than the data node's latest epoch **will not** create a new write. However, in K2-VCR, this guarantee is not enforced by rejecting new transactions but by calculating the commit epoch collaboratively between data nodes and coordinators. In particular, unlike CockroachDB, K2-VCR does not assign the commit epoch to a transaction at the beginning. When creating a new write, K2-VCR piggybacks the latest epoch of data nodes to the coordinator to decide the final commit epoch (§6.3). For instance, the commit epoch of T_3 in Figure 11 will be promised to be larger than the latest epoch of the data node A (i.e., 9) after a write is created on node A. As a result, K2-VCR does not need to artificially reject any transactions, since accepting new writes will not violate correctness guarantees. Note that this collaborative method also differs from Spanner, which allows the coordinator to determine the commit timestamp solely based on TrueTime.

For a slow commit transaction (T_2), K2-VCR preserves the same benefits as closed timestamps: undetermined keys will only affect the readers who access them (§6.4). Furthermore, since the epochs are generated according to TrueTime, they capture the real-time order between two transactions from different data nodes (§6.5).

6.3 Consistent Epoch Assigning

K2-VCR lets each node generate epochs independently and assigns commit epochs to transactions according to these generated epoch cuts. We first illustrate how epoch cuts are generated on the nodes.

TTC-based Epoch Cut on Data nodes and Coordinators. Figure 12 shows the procedure. Assume the epoch interval is set as $\mathcal{I}$ and the promised epoch end time for $epoch_n$ is T_{epoch_n}. Similar to our timestamp design (§4), K2-VCR first counts epoch interval locally. To do so, K2-VCR sets a local timer to record an amount of time that takes offsets and clock drifts into consideration (①-②). Given an assumed maximum clock drift as $\mathcal{D}$ (e.g., 200 ppm), we initialize the timer as $\mathcal{I} \times (1 + \mathcal{D})$ for the first epoch. After the timer expires, K2-VCR requests a timestamp ts via K2's timestamp algorithm (③). This timestamp verifies whether the wall clock time has passed by the promised epoch time (i.e., $ts > T_{epoch_n}$, ④). Otherwise, K2-VCR sets another timer, waits ($T_{epoch_n} - ts$), and verifies again. Then, K2-VCR waits out the uncertainty window of the given timestamp to ensure the wall clock time becomes larger than ts (⑤-⑥). Finally, K2-VCR can finalize the epoch and insert an epoch cut boundary into the node's data log (⑦). To start a new epoch (⑧), K2-VCR set the timer interval as $(\mathcal{I} + T_{epoch_n} - ts) \times (1 + \mathcal{D})$.

Based on this design, K2-VCR's epoch cut provides a property: although the wall-clock time for cuts is not exactly identical across different nodes (due to software execution delays), an epoch cut made by a node is always later than its promised time. Formally,

Property 6.1. *Denote the wall-clock time of epoch as* $abs(T_{epoch_n})$ *and the wall-clock time for cutting* $epoch_n$ *on* $node_i$ *as* $abs(T^{node_i}_{epoch_n})$. *Then, we have* $abs(T_{epoch_n}) < abs(T^{node_i}_{epoch_n})$.

Consistent epoch assignment. We then illustrate how to leverage the epoch cuts on primary nodes to assign epochs to transactions. For serializability, all writes of a transaction should be assigned to a single epoch. To do so, K2-VCR codesigns with K2-MVTO to piggyback epoch information during execution. Figure 13 shows our design. In the write phase, each write of a transaction (txn_1) will get a current epoch on $node_i$ (denoted as $epoch_i$) and responds with the epoch ($epoch_i$) to the coordinator (Line 10, Figure 10). Then, in the commit phase, the coordinator forwards the collected epochs to the recorder (Line 7, Figure 9). To assign a final epoch to txn_1, the recorder will get a current epoch on the recorder's node as $epoch_R$ and compute the commit epoch $epoch_C$ by calculating the maximum value of all proposed epochs (Line 18-19, Figure 10). After that, K2-VCR assigns all writes of txn_1 with $epoch_C$ in the finalization phase to finish the epoch assignment (Line 16, Figure 10).

This design ensures that the assigned epoch of a transaction is the maximum epoch observed by all participant nodes when receiving the transaction. Thus, it prevents the coordinator from assigning an epoch in the past (i.e., before other participant nodes are aware of the existence of the transaction) for correctness.

6.4 Reads at Replicas

K2-VCR supports both linearizable and snapshot reads at replica nodes. A linearizable read acquires its read timestamps from TTC using K2-TB (§4), while a snapshot read uses a stale timestamp as suggested by clients. To serve replica reads, K2-VCR first checks whether the replica's state (replayed epochs) is sufficiently up-to-date. K2-VCR uses the ceiling epoch of a read timestamp as its read view. For instance, assume a read-only transaction is given a read timestamp ts_{read}, and $T_{epoch_{k-1}} < ts_{read} \leq T_{epoch_k}$, then we use T_{epoch_k} as the read view. The read view includes all data replayed before $epoch_k$ and does not include data from future epochs.

Then, K2-VCR reuses K2-MVTO's read procedure for replica reads. In a given read view, K2-VCR compares the read timestamp of a replica read to the start timestamp of the write versions. As suggested in Figure 11, it's possible for a replica reader to see an undetermined write in a given read view (i.e., Key 100 created by T_2). Therefore, a replica read can also potentially trigger a PUSHWS to solve undetermined writes (Line 26, Figure 10). By doing so, K2-VCR guarantees the writes of a transaction are atomically visible to any given epoch (either all visible or all invisible). Formally:

Property 6.2. *All writes of a transaction (txn_1) that is assigned to $epoch_C$ are invisible in $epoch_X$, where $X < C$. On the contrary, all writes of txn_1 are visible in $epoch_X$, where $X \geq C$.*

Proof. For $X < C$, a reader will not see any finalized writes of txn_1 since the read view only includes finalized data up to $epoch_X$. However, a reader can potentially see undetermined writes of txn_1 and trigger PUSHWS. After PUSHWS, all writes of txn_1 are determined and assigned to $epoch_C$. As a result, these writes remain invisible. For $X \geq C$, all undetermined writes made by txn_1 on data nodes in $epoch_W$ must be included in $epoch_X$ since $X \geq C \geq W$, according to our epoch assignment (see §6.3). Thus, the reader must find the writes of txn_1 in either finalized or undetermined status. Any undetermined writes will be resolved by PUSHWS. □

6.5 Correctness of Consistency

Property 6.2 states that a transaction is atomically visible to a replica reader. Besides, we show that K2-VCR guarantees strong consistency in our technical report [40]: the real-time order between two read-write transactions at primaries is preserved at replicas.

	SH	BJ	GZ	GY	SG
SH	0.2				
BJ	27.3	0.2			
GZ	31.3	42.6	0.2		
GY	29.2	38.5	28.0	0.2	
SG	69.3	77.6	46.8	60.0	0.2

Table 1: Round-trip latencies between regions (in ms).

7 EVALUATION

7.1 Experimental Setup and Workloads

We implement K2 in C++ with Seastar [35] for message passing between processes. We conduct our experiments on our cloud. Each machine has 8 CPU cores (16 vCPUs), 64GB memory, and a 20Gbps network interface. In all cases, we only use physical cores; the other SMT logical core is left idle. All servers run Ubuntu 20.04 (Linux kernel 5.4). We deployed K2 over five regions (see Table 1).

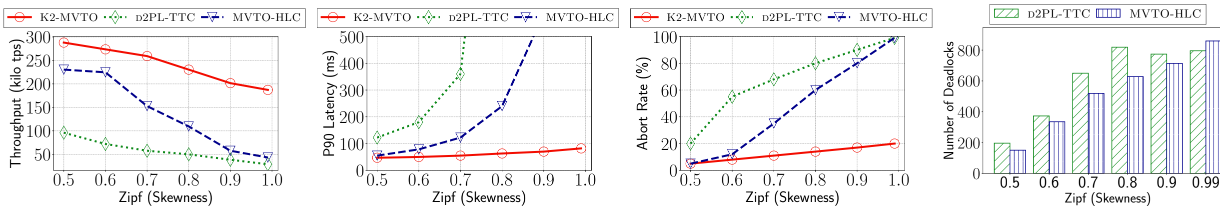
By default, in each region, we use one machine as a TTC Oracle server, five machines as data node servers, and five machines as clients to saturate the transaction throughput. Therefore, in total, we have 200 physical cores for data nodes. We use 100 cores for primaries and the other 100 cores for replicas. Replicas are only evaluated in §7.4. Unless otherwise stated, we run transactions on primaries and report the aggregated throughput results.

As suggested in §2.2, we use $100\mu s$ as an uncertainty bound for the TTC Oracle. A database is first partitioned across five regions and then further equally distributed across five machines. We use shared storage to persist transaction records and perform cross-region replication asynchronously. The epoch interval is set to $100ms$ by default. The clients were distributed evenly across all regions, ensuring sufficient capacity to prevent bottlenecks.

We run two benchmarks in our evaluation: TPC-C and YCSB-T. **TPC-C [7]** is typically partitioned by warehouses. We follow this partitioning and assign warehouses across the five regions. We initialized the database with 1000 warehouses and 10 districts per warehouse. Warehouses are equally distributed across CPU cores. **YCSB-T [8]** is used to fine-tune experimental parameters, thus studying the performance of K2 under different conditions. By default, we set three operations per transaction. Among all operations, 50% are writes, and 50% are reads. We used a Zipfian distribution, with the default Zipf parameter equal to 0.8.

7.2 Performance of TTC and K2-TB

Same as Spanner, we assume a very conservative clock drift rate (i.e., $200ppm$) and believe that TTC's implementation is trustworthy. Thanks to the advanced clock synchronization method (using PTP over BITS, §2.2) and the elimination of managing a large-scale clock cluster, K2 has a small assumed uncertainty bound (i.e., $100\mu s$, §2.2). In this experiment, we first evaluate the practical time uncertainty windows introduced by TTC. Figure 15 presents the aggregated ϵ of TTC deployed in five regions. It plots the $90th$, $99th$, and $99.9th$ percentiles of ϵ, sampled at TTC Oracle servers immediately after sending delay requests to time masters using PTP. Therefore, these results essentially suggest the uncertainty of the time master (which is typically in tens of nanoseconds) along with the communication delay from TTC Oracle servers to the time masters. It shows that the synchronization between TTC Oracle and time masters is efficient and sufficient for achieving our assumed uncertainty bound.



(a) Throughput **(b) Latency** **(c) Abort Rate** **(d) Deadlocks**

Figure 14: Impact of data skewness on the performance of K2-MVTO.

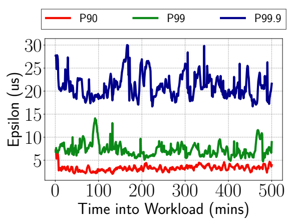


Figure 15: Epsilon over Time

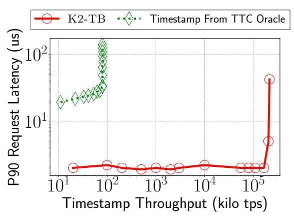
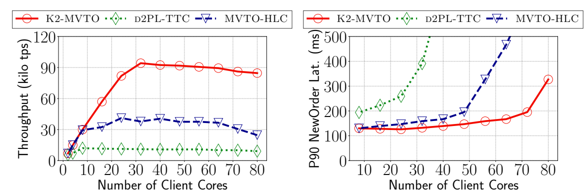


Figure 16: Performance of K2-TB



(a) Throughput **(b) Latency**

Figure 17: Peformance of K2-MVTO on TPC-C.

Figure 16 compares the latency-throughput curve of K2-TB to the strawman approach that directly requests timestamps from the TTC Oracle. The TTL and batch size are set to $100\mu s$. Both the x-axis and y-axis are in log scale. The results show that K2-TB significantly improves latency (15×) and throughput (1462×) for getting a timestamp since most timestamp requests ($> 99.9\%$) can be served locally. A side effect one may be concerned about is that K2-TB might increase the commit wait time (CWT) when applying TTL and batch size (with an increase from $200\mu s$ to $400\mu s$), potentially leading to more commit waits. However, experiments indicate that a small TTL and batch size are highly beneficial. Additionally, since K2-MVTO assigns timestamps at the beginning of transactions (unlike requesting commit timestamps during the commit phase, as in Spanner), the commit wait time is calculated at the beginning of transactions. Thus, latency in executing the read and write phase is included in CWT. In general, to commit a read-write transaction, K2-MVTO takes at least two SSD flushes (one for write operations and another for recording commit status).

7.3 Performance of Distributed Transaction

Baselines. The goal of K2-MVTO is to use TTC to achieve high performance for strictly serializable transactions over a multi-region data store. Hence, we intend to compare K2-MVTO to Spanner and CockroachDB, two leading systems supporting globally distributed transactions. These systems were selected because they complement each other and share at least one design aspect with K2-MVTO. For example, both Spanner and K2-MVTO utilize TTC, while both CockroachDB and K2-MVTO adopt MVTO's workflow.

Nevertheless, to our knowledge, Spanner is not open-sourced. The only available option, Cloud Spanner [42], is a managed service and does not reveal its hardware configuration. Moreover, both Spanner and CockroachDB have very different storage, replication, and distribution layer designs than ours. These different design choices introduce complex performance trade-offs, which are beyond the scope of our discussion. To avoid an apple-to-orange comparison, we re-implement Spanner's and CockroachDB's protocol in our codebase and call them D2PL-TTC and MVTO-HLC.

D2PL-TTC implements distributed two-phase locking (D2PL) and commits transaction using two-phase commit. Commit timestamps are generated in the commit phase from TTC clocks. Commit wait is disabled for a fair performance comparison.

MVTO-HLC adopts the same MVTO workflow for transaction coordination as K2-MVTO. The difference is that it uses HLC instead of TTC and generates commit timestamps for transactions at the beginning. It implements read refresh to adjust the commit timestamps and uses commit timestamps for data versioning.

7.3.1 ***TPC-C.*** We first evaluate the performance under the default setting of TPC-C. We launch each client core with 100 client instances. Each client instance executes one transaction at a time. As shown in Figure 17a, K2-MVTO achieves 2.32× and 9.82× higher peak throughput than MVTO-HLC and D2PL-TTC, respectively. Figure 17b shows the $90th$ latency of NewOrder transactions.

To identify the factors contributing to the performance improvements, we conducted factor breakdown experiments. Initially, all optimizations in K2-MVTO were disabled, including multi-version properties for reads (by blocking all reads on a key when write streams exist on that key), multi-version properties for writes (by restricting to only a single write intent in the write stream), and asynchronous finalization (by enforcing transactions complete finalization before commit). Then, we progressively re-enabled these optimizations. Figure 18 presents the results, highlighting the critical role of multi-version properties for read-write transactions (9.18× improvement). The benefits of asynchronous finalization appear less pronounced because most writes occur locally in TPC-C.

We study the latency breakdowns of K2-MVTO in Table 2. In particular, NewOrder transactions involve a query and an update on the Stock table, which can be located either in a local region or in remote regions. We present the breakdowns of the $90th$ latency of these two types. When querying and updating Stock in a remote region, K2-MVTO introduces cross-region latency in the read and write phase. Unlike some works that use a one-shot transaction model [27, 50] that requires rewriting transaction logic, K2-MVTO executes transactions in a multi-shot manner (i.e., the execution may involve multiple rounds) and thus supports general workloads. Since K2-MVTO always adopts asynchronous finalization and we do not perform synchronous cross-region replication in our experiments, the commit latency is low. It includes an intra-region write and flush latency to persist data in shared storage.

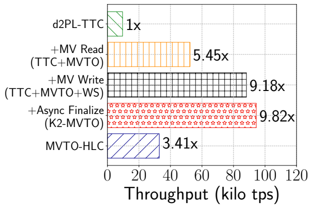


Figure 18: K2-MVTO's Performance Factor Breakdown.

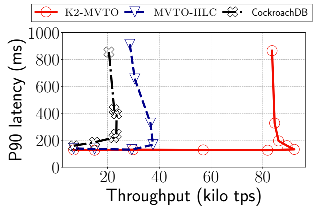


Figure 19: Performance Comparison with CockroachDB v24.1

Phase / Type	Get Ts	Read Phase	PushWS Phase	Write Phase	Commit Phase	Total
Stock in Local Region	0.008*ms*	0.052*ms*	0.018*ms*	0.552*ms*	0.526*ms*	1.156*ms*
Stock in Remote Region	0.008*ms*	76.312*ms*	0.025*ms*	43.207*ms*	0.530*ms*	120.057*ms*

Table 2: Lat. Breakdown of P90 NewOrder Transaction in K2-MVTO.

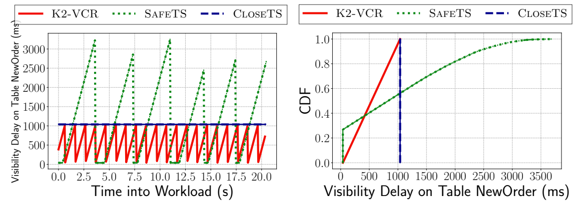
7.3.2 ***YCSB-T.*** We investigate how various workload properties affect K2-MVTO's performance using YCSB. As shown in Figure 14a, K2-MVTO outperforms its competitors in throughput across all levels of contention. This is because K2-MVTO's ability to lower the abort rate caused by conflicts by ordering transactions according to TTC timestamps. Along with write streams, read-write conflicts are the only type of conflict that causes aborts in K2-MVTO (§5.2). The collected abort rate results are presented in Figure 14c. Furthermore, by using start timestamps for ordering, K2-MVTO is deadlock-free. Figure 14d illustrates the number of deadlocks observed in the baselines over a 30*s* experimental interval, which negatively impacts their performance. We analyze the impact of varying write percentages on K2-MVTO in our technical report [40].

7.3.3 ***End-to-end Comparison with CockroachDB.*** We compared the end-to-end performance with CockroachDB v24.1 in Figure 19 with the same number of data nodes as K2.

7.4 Performance of Replica Read

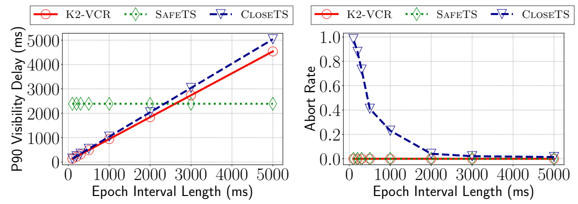
Baselines. We compare K2-VCR with Spanner's and CockroachDB's approaches (see §6.1). For a fair comparison, we re-implemented these two algorithms in our codebase, naming them SafeTs and CloseTs, respectively. SafeTs is developed based on d2PL-TTC and CloseTs is developed based on MVTO-HLC. By default, we set the interval of CloseTs to match the epoch length of K2-VCR.

7.4.1 ***Performance Overview.*** Similar to other works [6, 17, 39, 45], we define visibility delay as the maximum time between when an update is committed by a read-write transaction at the primaries and when it becomes readable by queries at the replicas. Figure 20a shows the results of Table New_Order in Region SG when peak throughput of TPC-C is reached (given an epoch length = 1*s*, a lower value causes significant performance issues in CloseTs, see Figure 21b). We focus on the New_Order table because it is a key table frequently updated in TPC-C. Results from other regions are omitted due to space constraints. Both K2-VCR and SafeTs exhibit sawtooth-shaped curves. This pattern arises because K2-VCR cuts epochs periodically, leading to transactions with commit times nearer to epoch boundaries becoming visible more quickly. In SafeTs, visibility is hindered by transactions that take a long time to commit on the table. Once long-running transactions are committed or aborted, the visibility delay reduces to the normal transfer delay (~50*ms*). CloseTs maintains a horizontal curve by



(a) Visibility Delay over Time **(b) Visibility Delay CDF**

Figure 20: Overview performance of K2-VCR.



(a) Visibility Delay **(b) Abort Rate**

Figure 21: Impact of epoch length on K2-VCR's performance.

enforcing a maximum interval for creating transactions' new writes and aborting read-write transactions that exceed this limit.

The CDF statistics in Figure 20b reveal that K2-VCR generally exhibits a lower average visibility delay compared to SafeTs and CloseTs. While SafeTs may achieve lower visibility delays than K2-VCR in the absence of blocking transactions, its tail delay can be up to 7.6× larger than average.

7.4.2 ***Impact of Epoch Length.*** In the final experiment, we examine how epoch length affects visibility delay. As shown in Figure 21a, K2-VCR and CloseTs experience a linear increase in visibility delay as epoch length increases, unlike SafeTS, which remains unaffected. In our experimental testbed, using an epoch length under 2000*ms* allows K2-VCR and CloseTs to outperform SafeTs. For practical deployment, we align epoch length with cross-region data transfer delays (e.g., setting epoch length as 100*ms*) to enhance performance. A smaller value does not introduce additional observable overhead since the epoch generation in K2-VCR is fully decentralized.

As shown in Figure 21b, the choice of epoch length significantly impacts the abort rate of read-write transactions at primaries when using CloseTs. The abort rate is particularly high when the epoch length is set below 2000*ms*. To avoid such penalties in TPC-C, in practice, CloseTs should be configured above this threshold (e.g., CockroachDB recommends 3*s* as the default value [41]).

8 CONCLUSION

This paper introduces K2, a multi-region data store that is designed to optimize distributed transactions via TrueTime clocks (TTCs). K2 has three key innovations: a TTC timestamp batching algorithm, a TTC-oriented MVTO protocol, and a TTC-based visibility control algorithm. Experiments highlight the effectiveness of K2.

ACKNOWLEDGMENTS

The work is supported in part by National Key R&D Program of China (2022ZD0160201), HK RGC RIF (R7030-22), HK RGC GRF (Ref: 17208223 & 17204424), a Huawei flagship research grant in 2023, SupernetAI, and the HKU-CAS Joint Laboratory for Intelligent System Software. K2 has been refined and developed by Huawei's Global TKV team to advance it into a full-fledged industrial product.

REFERENCES

[1] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[2] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. Polardb-x: An elastic distributed relational database for cloud-native applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2859–2872. IEEE, 2022.

[3] 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.

[4] Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, and Ion Stoica. Towards optimal transaction scheduling. *Proceedings of the VLDB Endowment*, 17(11):2694–2707, 2024.

[5] Google Cloud. Spanner replication. https://cloud.google.com/spanner/docs/replication.

[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

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

[8] 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.

[9] Google Cloud Spanner Official Documents. Timestamp bounds and Read Types. https://cloud.google.com/spanner/docs/timestamp-bounds.

[10] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 173–184. IEEE, 2013.

[11] J Eidson. Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)*, pages 1–499, 2020.

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

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

[14] Hua Fan and Wojciech Golab. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment*, 12(11):1471–1484, 2019.

[15] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[16] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. Caerus: Low-latency distributed transactions for geo-replicated systems. *Proceedings of the VLDB Endowment*, 17(3):469–482, 2023.

[17] 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.

[18] Huawei. NE20E-S V800R022C00SPC600 Configuration Guide. https://support.huawei.com/enterprise/en/doc/EDOC1100282195/329fe890/physical-layer-clock-synchronization-configuration.

[19] Ju-Shim. Time sync for linux vms in azure, 4 2023. https://learn.microsoft.com/en-us/azure/virtual-machines/linux/time-sync.

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

[21] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. ACM New York, NY, USA, 2019.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. ACM New York, NY, USA, 2019.

[23] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[24] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 1171–1186, 2020.

[25] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. {NCC}: Natural concurrency control for strictly serializable datastores by avoiding the {Timestamp-Inversion} pitfall. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 305–323, 2023.

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

[27] 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.

[28] Ali Najafi and Michael Wei. Graham: Synchronizing clocks by leveraging local clock properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 453–466, 2022.

[29] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. Detock: High performance multi-region transactions at scale. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[30] Oleg Obleukhov and Ahmad Byagowi. How Precision Time Protocol is being deployed at Meta, 2 2024. https://engineering.fb.com/2022/11/21/production-engineering/precision-time-protocol-at-meta/.

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

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

[33] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[34] Kun Ren, Dennis Li, and Daniel J Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11), 2019.

[35] ScyllaDB. Seastar - an advanced, open-source c++ framework for high-performance server applications on modern hardware. https://seastar.io/.

[36] Amazon Web Services. It's about time: Microsecond-accurate clocks on amazon ec2 instances, 11 2023. https://aws.amazon.com/blogs/compute/its-about-time-microsecond-accurate-clocks-on-amazon-ec2-instances/.

[37] Ohad Shacham, Yonatan Gottesman, Aran Bergman, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Taking omid to the clouds: Fast, scalable transactions for real-time cloud analytics. *Proceedings of the VLDB Endowment*, 11(12):1795–1808, 2018.

[38] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, pages 433–448, 2019.

[39] 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)*, pages 219–238, 2021.

[40] Haoze Song, Yongqi Wang, Xusheng Chen, Hao Feng, Yazhi Feng, Xieyun Fang, Heming Cui, and Linghe Kong. K2: On optimizing distributed transactions in a multi-region data store with truetime clocks (extended version), 2025. https://arxiv.org/pdf/2504.01460.

[41] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020.

[42] Google Cloud Team. Always on database with virtually unlimited scale. https://cloud.google.com/spanner?hl=en, 2024.

[43] Douglas Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, and Marcos K Aguilera. Transactions with consistency choices on geo-replicated cloud storage. *Transactions with consistency choices on georeplicated cloud storage*, 2013.

[44] 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*, pages 1–12, 2012.

[45] 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.

[46] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for {MVCC} with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 357–372, 2021.

[47] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. Oceanbase: a 707 million tpmc distributed relational database system. *Proceedings of the VLDB Endowment*, 15(12):3385–3397, 2022.

[48] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[49] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. *Proceedings of the VLDB Endowment*, 11(10):1289–1302, 2018.
[50] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.
[51] Zihao Zhang, Huiqi Hu, Xuan Zhou, Yaofeng Tu, Weining Qian, and Aoying Zhou. Fast commitment for geo-distributed transactions via decentralized co-coordinators. *Proceedings of the VLDB Endowment*, 17(10):2555–2567, 2024.