Scaling Write Throughput with Validation-Based Approach Between Regions

Abstract

Aurora-style disaggregated storage systems provide high write throughput at the primary node, near infinite read scalability, and high availability. However, single-writer systems incur write throughput limitations by being limited to one machine, incur additional latency because the client's location determines the transaction's perceived latency, and limit the write elasticity of the system. We attempt to show that the SunStorm architecture and algorithm can be used with various concurrency control protocols (2-phase locking, OCC, and MVCC) to create multi-writer systems for disaggregated storage system databases. The current iteration of the system fully implements single region transactions for all three concurrency control protocols in addition to the Write Ahead Log (WAL) writing and interpretation by other proxies. We experiment with local transactions for the three concurrency controls for this system for read-only, read-write, and mixed read-only/read-write transactions under high and low contention, measuring the throughput. Future implementations will complete the multiregion validation protocols and will experiment with multi-region transactions in a similar manner to the single region transactions with additional considerations for latency.

Introduction

The primary goal of the project is to demonstrate that any disaggregated storage system can be transformed into a multi-writer system provided it supports serializability in a single-node setting. We also prove that all the versions guarantee strict serializability as long as the initiating transaction tracks all the necessary metadata for the validation phase. The various versions of the systems will have the same architecture as SunStorm. However, each compute node will implement a simple Key-Value store rather than a full Database Management System (DBMS) such as PostgreSQL, as is the case in SunStorm. We choose to use a basic Key-Value store without supporting other DBMS functions such as secondary indices, foreign key relations, triggers, etc., because that shifts focus from the primary goal of our system. However, as described in the following sections and demonstrated via SunStorm, there is nothing inherent to the validation mechanism that prevents it from being applied to a DBMS with rich functionality.

In the "Related Works" section, we overview the landscape of systems that utilize disaggregated storage and why improvements to write throughput can be achieved. We also provide a simple overview of the systems we borrow from for our commit protocols. In the "System Architecture" section, we then describe the disaggregated storage system the idealized implementation runs on. In the "Transaction Workflow" section, we provide a high level overview of how local and multi-region transactions will be validated. In the “Implementation” section, we describe the differences between the system design and the current iteration of development. We also overview the proxy and WAL replay component of the system in addition to describing how each of the commit protocols works, delineating what is implemented and what remains to be implemented. In the “Experimental Results” section we describe throughput experiments with varying concurrency controls, contention, and type of transaction. Finally we conclude.

We note that we describe the ideal system in the “Introduction”, “System Architecture”, and “Transaction Workflow” sections. We were not able to fully implement multi-region transactions at this point so we delineate what is implemented and what is not implemented in the “Implementation” and “Experiment Results” sections. Also, much of the writing for this paper was directly written by Pooja, although we have made changes to many sections. The first paragraph of this “Introduction” section, most of the “System Architecture”, and most of the “Transaction Workflow” sections are taken directly from “Adopting the SunStorm algorithm to various concurrency control techniques” [1]. The “Implementation” section also borrows from it as well, although it includes much more of our own writing about the system. Additionally, much of the “Related Works” section summarizes or takes from Pooja’s “Scaling Write Throughput in Disaggregated Storage Databases” [2]. The “Experimental Results” and “Conclusion” sections are completely our own.

Related Works

Disaggregated Storage Systems

Disaggregated storage systems for DBMSs were pioneered by Amazon Aurora [3] [4] and have been developed by many other systems, including Microsoft Socrates [5], Alibaba PolarDB [6], and Huawei GaussDB (Taurus) [7]. Disaggregated storage systems provide the ability to independently scale the compute layer (which dictates operational workload) and the storage layer (which dictates the database’s size). These systems provide theoretically infinite read scalability in addition to better availability. The primary writer model that Aurora and similar systems use allows for a simple transactional model, identical to a single-node system. Thus, the primary node in such systems produces higher throughput than individual nodes in the shared-storage and shared-nothing models. The secondary nodes provide near-infinite read scalability, although often at a lower consistency and isolation levels. The disaggregation of storage from the compute layer also ensures that one of the secondary nodes can instantaneously handle failover if the primary node crashes, guaranteeing high availability without requiring the replica node to acknowledge every write before committing a transaction. The disaggregated model enables the system to limit the persistence responsibilities to the storage layer. Thereby providing high availability without paying the cost of consensus or replication at the compute layer. However, once the compute layer determines that a transaction can be committed, the storage layer persists the write-ahead-log (WAL) record using either replication or consensus across the individual storage nodes. Given that the WAL of a database is sufficient to generate the state of a database, the compute node does not persist changes to the data pages. Because a single primary node generates the WAL records and the WAL numbers generated by a database are monotonically increasing, the replication/commit protocol at the storage node is relatively straightforward. Since storage nodes are often deployed across availability zones (AZ) and regions, disaggregated storage systems remain available as long as a sufficient number of storage nodes are available. We attempt to retain the benefits of disaggregated storage systems (high write-throughput at the primary node, near infinite read scalability, and high availability across AZ/regions depending on the deployment) while increasing write throughput.

A clear downside of the Aurora-style primary node writer is that it creates a bottleneck for writes, which, while providing higher throughput than individual nodes in a shared-storage or shared-nothing setting, lowers the write-throughput when compared to multi-writer systems. Transaction latency can also become a problem. Most traffic in geo-distributed applications can be partitioned to leverage data access locality, however with a single writer, the client's location relative to the location of the writer determines the transaction's perceived latency. Many distributed systems also repartition data when the application traffic changes to keep access locality benefits. Disaggregated storage systems must scale up or down to accommodate changes in workload, so they do not support the same write elasticity as shared-storage systems. The following systems that attempt to address the problems of single-writer disaggregated storage systems can be categorized as sharded multi-writer systems and systems based on data movement.

Sharded Multi-Writer Systems

Sharded multi-writer systems shard the database and assign a primary node for each shard. Then, when a transaction modifies data for a single shard the transaction only needs to interact with the primary node. If the transaction touches multiple shards, then a 2-phase-commit protocol is determined by the specific system. Many of these systems also reduce the isolation levels provided by the system, mainly due to usage of the three-tier architecture. Aurora Limitless [8] utilizes horizontal scaling by sharding the database across multiple writers and uses proxies to send requests to corresponding database shards, although all writers are located in the same availability zone meaning that latency issues remain. Aurora DSQL [9] uses optimistic validation requests that go to the primary adjudicator of all the shards that a transaction touches and writes are validated using an OCC protocol but reads are not validated, meaning snapshot isolation is the highest level isolation level supported. In addition, DSQL does not support any features that rely on triggers and since DSQL journals are replicated across regions, each transaction incurs cross-region latency on commit (for the overall commit, not individual statements within the transaction). PolarDB-X [10] uses a traditional two-phase commit for cross-shard transactions and each statement is processed by the data node and supports scalability and elasticity bottlenecks but only guarantees snapshot isolation. SunStorm [11], however, utilizes a 2-phase-commit protocol to implement a serializable version, and we expand this idea to multiple concurrency control schemes in this system.

Data Movement Systems

Data movement based systems move data ownership across to enable a single node to commit a transaction, often using locking and cache coherence techniques to disable concurrent updates to the shared data items. Taurus-MM [12] uses hybrid locking with page locks managed by a GlobalLockManager and row locks within a page when it attempts to write data. With the locking approach, the system supports strict consistency and also uses 2-phase-commit to serialize transactions without cross-writer translations, however the GlobalLockingManager becomes a bottleneck and single point of failure. Gauss-DB [13] is an evolution of Taurus-MM that uses a 3 tier architecture with a “memory” mid-layer that is essentially stateless that manages page locks, however the benchmarks use RDMA and are located in close proximity to each other.

Concurrency Control Protocols

For the locking commit protocol, we implement an Assignment 2 based version of Two-Phase Locking, and we also include the design for multi-region transactions. For the optimistic commit protocol, we borrow from Silo [14]. Silo’s commit phase involved a phase locking the writeset, validating the reads, and then persisting the changes. We implement a version of Silo as a local commit protocol and also extend it to a multi-region validation protocol. We chose to implement the MVCC version of our protocol using Hyper/Hekaton after considering 4 different implementations of MVCC. MVTO [15] requires writes to each key on every read, which is infeasible when handling remote writes. MV2PL is relatively similar to our locking implementation already, and would require complex compare-and-swap operations. MVCC+SSI [16] contains added complexity since the SSI was “added on” to the original MVCC implementation. For these reasons, we chose MVOCC based on the implementations from HyPer/Hekaton [17] [18].

System Architecture

The idealized version of this implementation utilizes disaggregated compute and storage layers. For the purposes of this report we include this system architecture section for completeness, however our current implementation currently does not disaggregate compute from storage, as described in the “Implementation” section.

Compute Layer

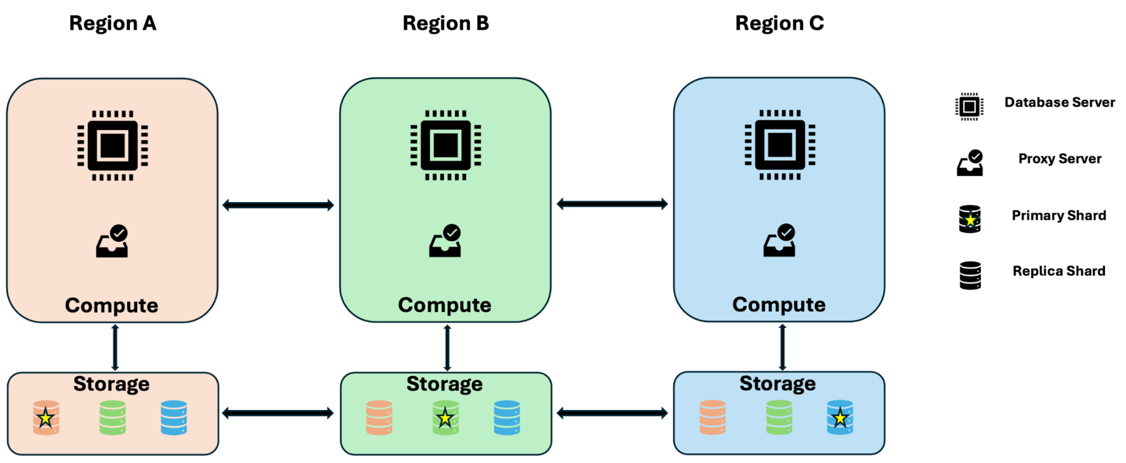
The compute layer is responsible for all database functionality, except for durability. In our prototype, the compute node supports data access and transaction management, including client-issued and validation transactions. The compute layer is further divided into two components: Database Server and Proxy Server. This is primarily meant to modularize and reuse code across various implementations. Additionally, making the proxy layer independent will reduce overheads incurred by WAL replay and validation.

The Database layer will maintain the active version of the data that is owned by its region and handle transaction management. All clients will directly interact with the database server. The proxy server, on the other hand, will be responsible for replaying and providing (somewhat) up-to-date copies of the data stored in the remote regions and managing validation requests, including the communication necessary for 2-phase commit (2PC).

Storage Layer

The storage layer consists of a set of (3 or more) active replicas in each region. Each replica simply persists the WAL records of the transaction that commits. The active replicas are always synchronized with other group members and need to be updated before a transaction is committed. The active replicas store and serve the data that belongs to the primary shard in a region. In addition, the storage layer consists of passive replicas that asynchronously capture updates from shards located in other regions.

The diagram below outlines the various components in the database system.



Transaction Workflow

Transactions that do not access data for reads or writes from other regions do not interact with other regions for any reason. Instead, they proceed as a local transaction, completing their local commit protocol, and writing their Write Ahead Logs (WAL). Other regions will then receive the broadcast WALs and asynchronously populate their replicated storage with the new changes. Whether a transaction will follow a local commit protocol only or a multi-region validation procedure is determined during the transaction, if the transaction accesses or writes data to which it is not the primary shard for. The following descriptions of single and multi-region transaction behavior are overviews and the “Implementation” section expands on how each commit protocol and concomitant multi-region validation protocol works.

Single Region Transactions

The goal of our system for single region transactions is to be similar to a single-writer in Aurora. (1) Clients will first connect to the database server and begin a transaction. (2) The database server will then execute the query and determine whether to commit or abort based on the given concurrency control algorithm. (3) The client will then notify the database server that it requests to commit. (4) If the transaction can commit based on the local concurrency control protocol (note this is not the multi-region validation protocol, this is only the local commit protocol), then we write a `COMMIT` WAL record to the storage layer which will include information that other regions will use to repopulate their copies of remote data asynchronously. If quorums are used then we wait for 2 of the 3 replicas before continuing and if full replication is used then we wait for all 3 responses from the storage layer before continuing. Once we have received either 2 (for quorums) or 3 (for full replication) responses from the storage layer acknowledging the persisted WAL records, we return the `COMMIT` result to the client. If we have determined that the transaction must abort (based only on the local commit protocol), then an `ABORT` result is returned to the client instead. (5) If we did not receive enough responses from sending our WAL records to the storage nodes, then a real world implementation would implement a form of cluster management and replace non-responsive storage nodes. For now, we can ignore this case as long as the database server never responds to the client without persisting the WAL records.

Multi-Region Transactions

A multi-region transaction reads or writes data from remote regions, and thus must use a multi-region validation protocol. The local commit protocol is distinct from the multi-region validation protocol, but it will determine some features of the multi-region protocol. Described here is an overview of how multi-region validation protocols will work, but the implementation section will further describe how the multi-region validation protocol changes for different local commit protocols.

(1) Similarly to a single region transaction, the client will start by connecting to the database server and request to begin the transaction. At this point, the database server will not know whether the transaction will be required to undergo a single region transaction commit protocol or a multi-region transaction validation protocol in addition to the local commit protocol. (2) During the execution, if the client query operates on local data, then the database server returns the data items from its state based on the visibility rules of the concurrency control protocol. If the client query operates on remote data, then the database server contacts the proxy server to read remote data. If the client tries to modify the remote data, the database server caches the writes locally and does not execute the writes according to the local protocol. At this point the database server knows that it is a multi-region transaction and thus will eventually execute the multi-region validation protocol (and the regions of any participating transactions are cached). To ensure that a transaction gets an atomic snapshot of data for each region, the database server caches the Commit Sequence Number (CSN) that is returned by the proxy server for the first remote data item that belongs to a particular region. All subsequent requests to the proxy server send the cached CSN with the expectation that the proxy server returns the data items

as of that CSN. The CSN for each region is independent, thereby requiring that the database server caches the CSN on a per region basis. (3) Once transaction processing is done, the client sends a `COMMIT` message to the database server. (4) The server will then commit according to the concurrency control rules and send a `PREPARE` WAL record to the storage layer and execute the validation protocol on the proxy server. The proxy server (initiator) collects the read and write sets of the transaction and sends it to proxy servers of the remote regions involved in the form of a `PREPARE` message. The proxy servers (validators) on the remote regions start a validation transaction on their database server. These remote database servers then verify that the read set collected has not been modified since the CSN they read with from the initiator region, and that the write set can be persisted without violating the concurrency control requirements. Once the database server responds to their proxy server at the remote regions, if the database server returns true, then the proxy server sends an accept message to all the other participants (including the initiator) and waits to learn the outcome of the transaction from the other regions. If the database server returns false, then the proxy server sends a reject message to all of the participants and the transaction is immediately aborted. (5) If all of the validator regions send `ACCEPT` messages back to the initiator's proxy, the transaction commits and the database server responds to the client. However, if a single validator sends a `REJECT` message back to the initiator, then the database server aborts and responds with an `ABORT` to the client as well. (6) The validator regions learn the commit outcome from other participants in parallel and execute the `COMMIT` or `ABORT` decision to the validation transaction at their region. Although each region transitions to the final state independently, decentralized 2-PC guarantees that each region reaches the same `COMMIT` or `ABORT` decision.

Implementation

Since we base our implementation off of assignment 2, some key differences exist between the proposed architecture and our own. Each server is a single proxy process that runs an instance of transaction_processor. Additionally, our storage layer is not completely separate, and we simply write our logs to the server device's storage without replication. This implementation avoids the separate communication between the compute and storage layers that the proposed design assumes will be done to achieve either a quorum or full replication before responding back to the database server. Instead, by writing in the same process, we are essentially guaranteeing successful WAL writes and can immediately proceed in processing. We assume that by disaggregating storage we would incur additional costs by waiting for WAL file writes, but probably not much because they are all writes to storage in the local region. We would also gain the advantages of disaggregated storage described in the "Related Works" section by disaggregating storage from the compute layer.

Assignment 2 also assumes the database server knows the read and write set of a transaction before running the transaction. Our implementation eliminates the need to know the readset and writeset at the beginning of the transaction. Achieving this in locking is trivial by requesting locks at the appropriate time during the running of the transaction. To achieve this in OCC and MVCC concurrency control systems, we read directly from storage rather than locally buffered reads that were performed at the start of a transaction as in assignment 2. Writes in the OCC implementation are still locally buffered, because they are saved until commit time to be written (if we do not write then the buffered writes are never written to storage). Writes in MVCC are now written to storage directly, however the nature of the MVCC version implementation means that versions that are not committed are never seen by other transactions.

At this point in development, we have working single region transactions for all three concurrency control systems. This means we have implemented strict 2-phase locking with deadlock detection, a version of Silo, and a version of MVOCC that are extendable to our proposed multi-region validation protocol. All three concurrency controls have asynchronously parseable WALs being broadcasted to other regions. We are still debugging multi-region transactions and their concomitant validation protocols, although most of the pieces for multi-region validation for OCC are written (with debugging required) and locking and MVCC implementations require some more work to fully integrate the proxy to do messages for the multi-region validation protocol.

Proxy Server

The proxy in our system implementation serves two primary responsibilities. First, it is responsible for replaying WAL records originating from other regions, thereby keeping a local replica of remote data relatively fresh. Second, it manages inter-proxy communication during multi-region validation, coordinating the necessary protocol steps across participating regions.

WAL Replay

Each proxy server maintains persistent connections to the proxies of all other regions. For each remote region, a dedicated thread is responsible for handling incoming WAL entries and applying them to the proxy’s replica of remote data. This replay mechanism is asynchronous, meaning the remote replica may lag behind the primary region, but provides a reasonably up-to-date view of the remote state for read-only access and validation. WAL entries are serialized as individual lines in JSON format, where each line represents a single transaction’s write operations to be replayed.

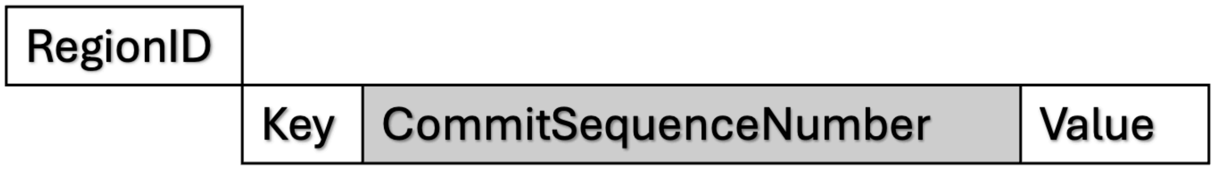
Inter-proxy Communication

The proxy uses conventional socket based communication, with one bidirectional TCP connection established between each pair of proxies in a peer-to-peer topology. Each proxy is responsible for decoding incoming messages, dispatching appropriate actions, and coordinating with the local database server. In the context of multi-region transactions, the proxy performs tasks such as locking for remote writes, initiating or responding to validation messages, and aborting transactions when validation fails.

Each proxy is tightly integrated with the local database server and serves as the communication backbone for 2PC coordination during multi-region validation. In our current implementation, this setup supports correctness and simplicity, while leaving room for future optimizations such as batching (e.g, group commit).

Locking (Strict 2-Phase Locking)

In order to guarantee serializability across regions while using locking, we maintain four fields per tuple: RegionID, Key, CommitSequenceNumber (CSN), and Value. The RegionID represents the home region of the tuple, and the CSN is the CSN of the transaction that wrote the newest version of this tuple. We only maintain one version of each tuple in locking. The CSN is only used for multi-region validation.



Our 2PL implementation is based on Assignment 2, but contains a few key differences. In assignment 2, transactions were only run after both its read and write sets were locked. However, our system acquires locks as the transactions run. Since lock acquisition is now performed by different threads in any order, we require a mechanism to detect deadlocks. We continue using Assignment 2's queue-based lock manager but with the addition of a waits-for graph. Whenever a transaction requests a lock, we update the waits-for graph. We have two rules:

1. Transactions must not create a cycle
2. Multi-region transactions must not wait for other multi-region transactions

The latter prevents excessive waiting. We follow a wait-die scheme, so transactions abort themselves upon breaking these rules.

In our locking protocol, the first operation on a remote region per transaction fetches and stores the latest cached CSN of that region from the proxy. Since CSNs are meaningless across regions, we must store one for each remote region we access. In order to simplify checking for rule 2, transactions are marked as single or multi-region at initialization. This simplification can be removed at the cost of storing and traversing a reverse waits-for graph and more aborts. Finally, transactions that are forced to abort are requeued.

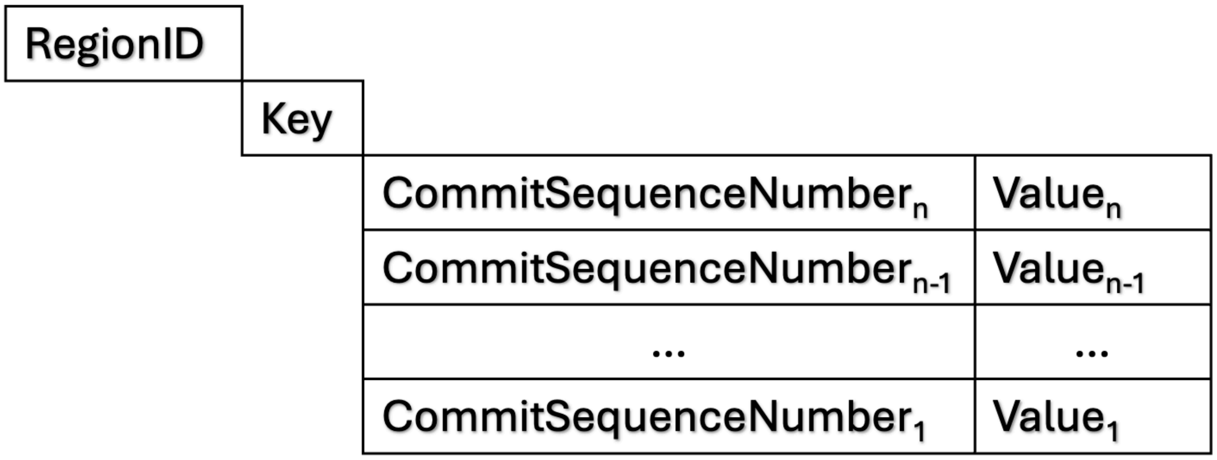
Single-region transactions simply acquire locks as they run. Once they finish executing, they log their writes, make them visible, commit, and release their locks. Since transactions have their own buffers, aborts only need to release locks and clear buffers.

Multi-region transactions run according to the following procedure. When performing a remote read, we use the transaction's saved CSN for that region to query our local versions of the remote databases. If we see a CSN higher than our cached value, it means another transaction has modified the tuple we read. If the transaction successfully reaches the end of its local execution, we log that we are ready to validate with a PREPARE and ask the home regions of our remote reads and writes to validate their CSN according to the logic above. For the side receiving the validation request, we log a PREPARE and send a commit message if we pass validation. If we fail, we log the abort and send this abort message. To ensure we are still following 2PL, all regions only release locks once validation is complete.

At the time of writing, single-region transactions are working, but multi-region transactions require further integrations and bug-fixing with the proxy.

OCC (Silo)

To implement an optimistic concurrency control version, we borrow from Silo. In addition to the performance improvements of Silo from traditional OCC, there are also other reasons Silo fits the specifications required of our multi-region commit protocol. (1) The commit outcome of a transaction may not be known at the time of validation because coordination between proxies of remote regions is required. Silo allows us to wait until the multi-region outcome is known to make a decision on whether to commit or abort. (2) The amount of state maintained to guarantee serializability must be bounded. Parallel OCC would guarantee serializability by maintaining each transaction's write set until all transactions that started before it committed have completed. Since there isn't a central Oracle that knows all the active transactions in the system, OCC-P would disallow us from ever pruning the write-set information. (3) Lastly, we inherently want to track the transactions that created a version to make the validation process straightforward. Silo's protocol allows us to achieve this. We don't implement Garbage collection currently, however Silo's approach can work with our system.


RegionID
Key
$\text{CommitSequenceNumber}_n$
Value_n
$\text{CommitSequenceNumber}_{n-1}$
Value_{n-1}
...
...
$\text{CommitSequenceNumber}_1$
Value_1

The data layout of the optimistic concurrency control (as shown in the figure above) contains 4 fields: RegionID, Key, CommitSequenceNumber (CSN), and Value. The RegionID corresponds to the primary region that a tuple belongs to. The RegionID represents the first-level key, and all keys within a region will be stored and operated on a single shard. The Key is the key of the tuple which is used to identify the data item. The CSN identifies the transaction that wrote that version of the tuple. The CSN is stored in descending order (with the most recent version being the first record). We use Silo’s optimizations to use higher order bits to act as write-locks (latches). The higher-order bits are meaningful only for local data. In case of remote data, we only care about the one latest version. The value column consists of the values associated with the key.

The version implemented for this report has fully working local transactions with WAL replays between regions, populating a remote storage asynchronously. Multi-region transactions are not fully implemented at this point, but below we describe how they will work (and at this point most of the implementation is finished, but further bug fixing with proxy sending and parsing is required).

Local Transactions

Local transactions will follow a commit protocol similar to the one described in the “Transaction Workflow” section. A client will make a transaction request and the database server will execute it. During the process of execution, the database will read from its storage and write to a local buffer. It will populate a transaction local cache that stores the CSN values that we read at that will be used to validate our local reads during the commit protocol. In the process of reading and writing, the transaction will see that no remote region keys were accessed, and will begin a local commit. The transaction starts by requesting locks on its writeset (by switching the lock bit of the top most version to be locked). If any of the versions in the writeset are already locked, we immediately abort (because another transaction has gotten to validation and will write to that version making the transaction trying to request a lock invalid) and release the locks we were able to obtain up to that point. If we manage to acquire all of the locks on our writeset, we try to validate our reads by checking the most recent version of our readset in storage. If the CSN of the most recent version is larger than the CSN value we read at, then our read was invalidated so we abort the transaction and release the locks on our writeset. If the most recent version is locked, then we also abort and unlock our writeset because another transaction got to the process of validating their write to that key before we validated our read. If all of the transaction’s reads are validated, then we persist the WAL to storage of the commit that includes a CSN value (atomically incremented value for each region that is unique for each set of writes), the number of writes, and the list of writes made by the transactions. We then apply the writes by iterating through the locally buffered writes and writing them to storage with the CSN of the transaction that was atomically obtained during the point of writing the WAL to storage. We then unlock the writeset that we locked at the beginning of the commit protocol and return the commit result to the client.

Multi-Region Transactions

Beginning like the local transaction, multi-region transactions are submitted to the database server and then executed. During the process of execution, other regions’ data are accessed as described in the “Transaction Workflow” section. Reads to the locally mastered data are done to local storage and writes to locally mastered data are buffered to the transaction locally. Reads to the foreign mastered data are done to our asynchronously replicated version with the CSN caching to read consistent versions from each region. Writes to foreign mastered data are cached locally and will be sent during the validation protocol. Once we finish processing the transaction, we start the multi-region validation process. We first try to lock the writeset of the transaction locally (only locking the writes of the writeset that are mastered at this region). If we are unable to obtain the locks, we abort and release the locks we were able to obtain. If we were able to obtain all of the locks locally, we ask the proxy to lock the writes in the remote regions. The proxy at the initiator region then communicates with all proxies of regions related to the transaction and asks them to lock writes. Since this is the first phase in which the proxies of the remote regions are contacted, they create a validation transaction at their region which is cached for future phases of communication. They then attempt to lock the writes of the writeset of the transaction that are mastered at that region. If the validating regions are able to obtain locks on all of their writes, they respond back to the initiator regions proxy. If a single validating region is unable to obtain all of its writeset locks, then it communicates to all of the other proxies that they should abort and they do so. If all validating regions are able to lock their writes, then the initiator proxy will receive all of the affirmative responses from the validator proxies.

The multi-region validation then continues at the initiator region which will attempt to validate reads of the local reads. If not successful, then we abort and also tell all of the other regions to abort. If we manage to validate our local reads at the initiator, then we use the proxy to contact the validator region proxies so that they can validate their reads. Notably, OCC is different from 2-phase locking and MVCC implementations because those multi-region validation protocols are able to request validation from participating regions, collect responses, and then immediately make and broadcast the decision to commit or abort. OCC, however, requires 2 phases of multiregion validation because the write-locking phase and read-validating phases are distinct. Therefore, at this point in the multi-region validation protocol, we send all of the validating regions a request to validate the reads for data mastered at their region. To validate at remote regions, we compare the CSN value we read at the initiator when executing the transaction to the CSN of the most recent version in the remote region. Like local read validation, if the CSN has changed or the most recent version is locked we abort. Similar to failing at the locking stage, if a single region aborts, it broadcasts to all other involved regions that they should abort. If the validating region does not abort, it will respond with an affirmative message to the initiator. If all affirmative messages are collected at the initiator, then the initiator sends a message to all of the validator regions to commit their writes (if they have any). At the point the initiator receives all affirmative responses to the validate reads requests and sends a request to all participating regions to commit is the point at which the decision to commit is finalized, all regions will then persist the changes. At each region, the transaction (either the initial transaction or the validation transactions) will persist a WAL record with their changes, apply their locally buffered writes, and release their locks.

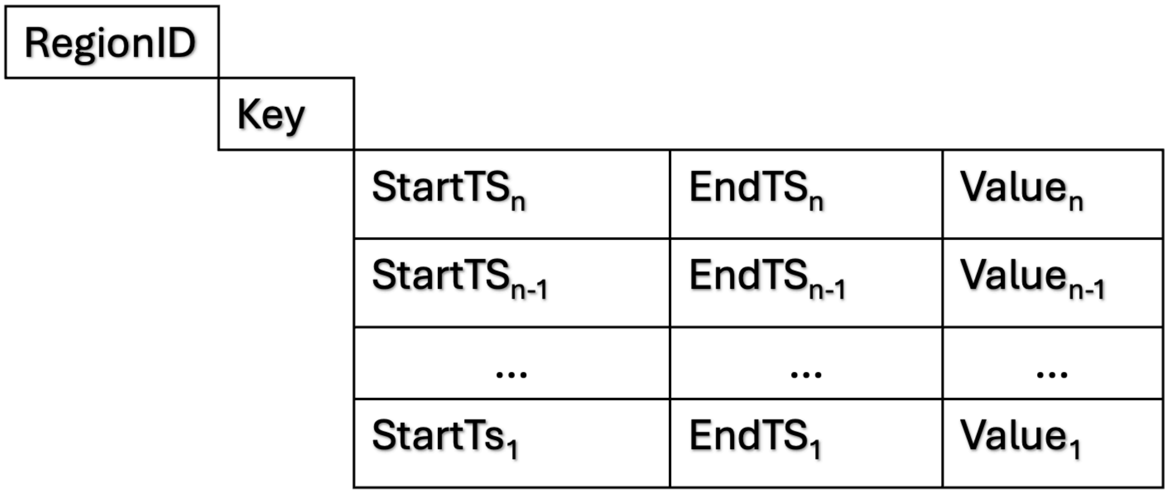
MVOCC (HyPer/Hekaton)

Local Transactions

Each transaction in MVOCC progresses through 3 phases. The execute phase is when the transaction executes its reads and writes. This corresponds with the transaction entering an ACTIVE state. The validation phase is similar to OCC's validation phase, which includes both local and remote validation, and corresponds with the PREPARE state. When it is certain the transaction will commit, it enters the postprocessing phase, where the transaction will persist its logs and change itself to the COMMIT state. Otherwise, the transaction will proceed to the ABORTED state.

The transaction local state stores a startID and a commitID, retrieved from an atomic monotonically increasing counter. It is helpful to be able to distinguish between the two to convey information about state; to accomplish this easily, we ensure all startIDs are odd and all commitIDs are even. Transactions acquire the startTS at the start of their execute phase. They acquire the commitTS at the start of their validation phase. Each startID and commitID are unique to the transaction due to the nature of the counter. The serializable order of the transactions are determined by their commitID.

MVOCC's tuple structure contains a deque of "versions" (tuples) for each key. Each version contains a startTS and endTS instead of the CSNs in the previous implementations, along with a value. This startTS and endTS can be populated with either startIDs or commitIDs depending on the states of transactions. Since IDs are unique, we will say a transaction populating a field is the "owner" of that field moving forward. In general, an odd startID in either the startTS or endTS field belongs to an ongoing transaction, which requires more casework to resolve. An even commitID in these fields always belongs to a committed transaction because we ensure that transactions update the versions to their commitID after they commit.



The txn_map stores info (startID, commitID, txn_state) about every transaction, with startID being the key. It is atomic, with accesses guarded by a read/write lock. The map is updated whenever the transaction state changes.

Note that aborting in the below contexts due to write/validation protocols can also mean restarting the transaction immediately. We choose to only abort the transaction in our current implementation.

Local Reads

The visible value is defined as the version with the highest valid startTS and endTS. We define transactions with valid startTSs to be any transaction in the COMMITTED or PREPARE state. An even startTS automatically implies the owning transaction is committed. However, any odd startTS requires a lookup in the txn_map. This usually happens when the owner of the startTS is in the PREPARE state (with some edge cases due to multithreading). We call such reads "speculative reads", which require further processing during validation (since the owner can still abort). We save the startID (as a copy) of each read version for validation. To support deletion, a check can be added on the endTS to ensure it is not committed. If it is committed, the reading transaction will be certain that their commitID is greater than the endTS, and the tuple should not be visible.

Local Writes

Writes employ a "first writer wins" policy, with the contention point being the endTS of the most recent valid version. Local writes begin with finding said version. When found, we check the version's endTS. If the owner has committed, there should be another valid version available to overwrite, and we restart the write to find it. If the owner is in the ACTIVE or PREPARE phase, it means some other writer is currently overwriting the most recent valid version, and the transaction trying to write should abort.

If the version found (refer to this as the "old version") is available to be overwritten, a new version is pushed onto the queue with (startTS = startID of transaction, endTS = infinity, value = value). Outside of speculative reads, this version is only visible once the transaction owning its startTS commits.

Next, a compare and swap is executed on the endTS of the old version, updating it to the writing transaction's startID. If it succeeds, the transaction has successfully become the first and only writer updating the old version. If it fails, some other transaction got there first, and we have to abort.

Finally, both the old and new version are saved to be edited during validation. This is because the writing transaction's update is pending until it finally commits and updates the endTS of the old version and the startTS of the new version to its even commitID.

Local Validation

Before starting validation, the transaction changes its state to PREPARE and updates the map. The local validation phase only validates any reads made. The validating transaction loops through the startIDs of the reads made. If the startID is odd, it signals that the read was speculative (owner was preparing when we read). The transaction looks up the owner of the startID in the txn_map.

If the owner is still in the prepare phase and is local (no remote reads or writes), we wait until it either commits or aborts. We abort instead of waiting on multiregion transactions due to their long validation phases. Otherwise, if the owner is in the active or aborted phase, we also abort.

Next, we reread the valid tuple for each key. If the endTS has changed to a value less than our commitID, we abort. There is also a check to exclude the endTS changing due to the validating transaction's own write.

Postprocessing Phase

The transaction's own logic may cause it to abort. If that is the case, it is handled here.
Next, the transaction write data is persisted to a local WAL and broadcasted to all other regions for asynchronous replication. Next, the transaction changes its state to COMMITTED. Finally, we iterate through the saved writes and change the old tuple's endTS and the new tuple's startTS to the transaction's commitID, finalizing the writes.

Multiregion Transactions

A multiregion transaction is one that has at least one remote read or write. Each node is equipped with a storage object per remote region that contains a delayed view of the region. We will refer to this object as the remote view.

Remote reads/writes

Remote reads read the most recent valid version (will be the topmost tuple) for the key from the remote view and save the key and version startTS for validation. Remote writes also read the most recent valid version and save (key, value, startTS) for validation.

Remote validation

Remote validation occurs after local validation for multi-region transactions. The initiating transaction simply sends the request to the proxy and waits for responses. If all remote regions vote yes, the initiating transaction enters postprocessing. Otherwise, the initiating transaction aborts.

Remote validation transaction

When another region receives a remote validation request, it initiates a special remote validation (RV) transaction, which will have data of each write and read from the initiating transaction corresponding to the region. The RV transaction has the same phases and statuses as a normal transaction (to handle visibility of its writes) and acquires a startID and commitID as usual. Its main purposes are to validate the reads and apply the writes locally to the region.

Accordingly, the execute phase of the RV transaction applies the writes requested. A check is added to ensure the expected startTS of the write is what the RV transaction should overwrite.

Next, the validation phase of the RV transaction loops through the reads and ensures they were valid. It finds the most recent valid version under the selected key and checks that the startTS is equal, and the endTS is not prepare or committed.

Once the validation phase is complete, it votes yes through the proxy and waits for other regions to respond. If all regions vote yes, the transaction runs postprocessing similar to a normal transaction. It persists and broadcasts the WAL, updates the transaction status, and updates the saved writes with its commitID. Otherwise if a single region votes no, the transaction aborts.

Experimental Results

Experiment Setup

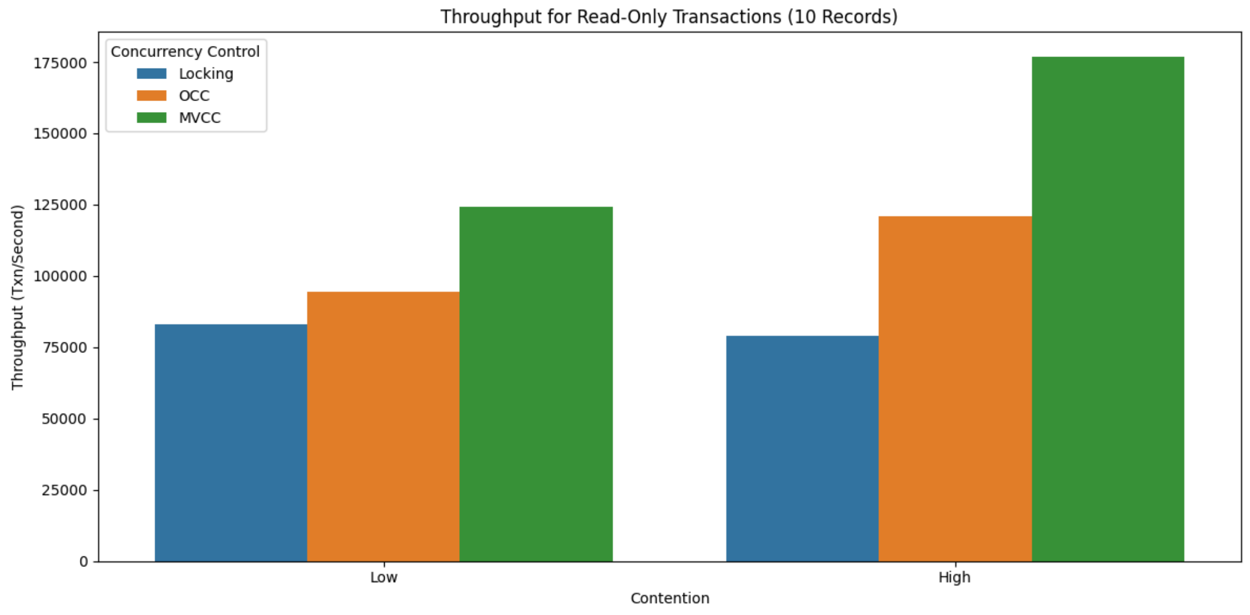
Component	Spec
CPU	AMD Threadripper PRO 5955WX 16-Cores
Clock Rate	4500 MHz
Hyperthread	ON
Memory	512 GiB
L1 Cache	512 KiB
L2 Cache	8 MiB
L3 Cache	64 MiB

Table 1. Hardware Specification

We run all experiments on the server depicted in Table 1. Our experiments rely on a microbenchmark, a variant of Assignment 2, where we added the configurations required for multi-region purposes. The experiment runs for 30 seconds while executing workloads of either read-only (RO), read-modify-write (RMW), or read-write-mixed (RW) transactions. Unless otherwise noted, all experiments were performed with transactions with a readset or writeset of 10 records. High contention experiments limited the transactions to only access 50 possible records (effectively limiting the size of the database to force contention). Low contention experiments were allowed to access 1,000,000 possible records. We measure throughput of the transactions as the number of transactions that completed per second. The figures show the throughput averaged between the 2 regions involved, and in all test cases, the throughput numbers between regions were similar.

It is worth noting that we developed the 3 concurrency control protocols separately and drastically changed the assignment 2 code base. This means that differences in throughput values seen in these experiments might be influenced by the differences in implementation, and not only the concurrency control method. Eventually we would like to integrate the three versions together and share as much code as possible to eliminate the effect of different implementations. An example of an implementation difference is that in OCC we restart the transaction when it fails to gather locks or validate and in MVCC we simply abort. Both implementations are technically fine (if they are aborted it would be up to the client to retry), but could have performance implications.

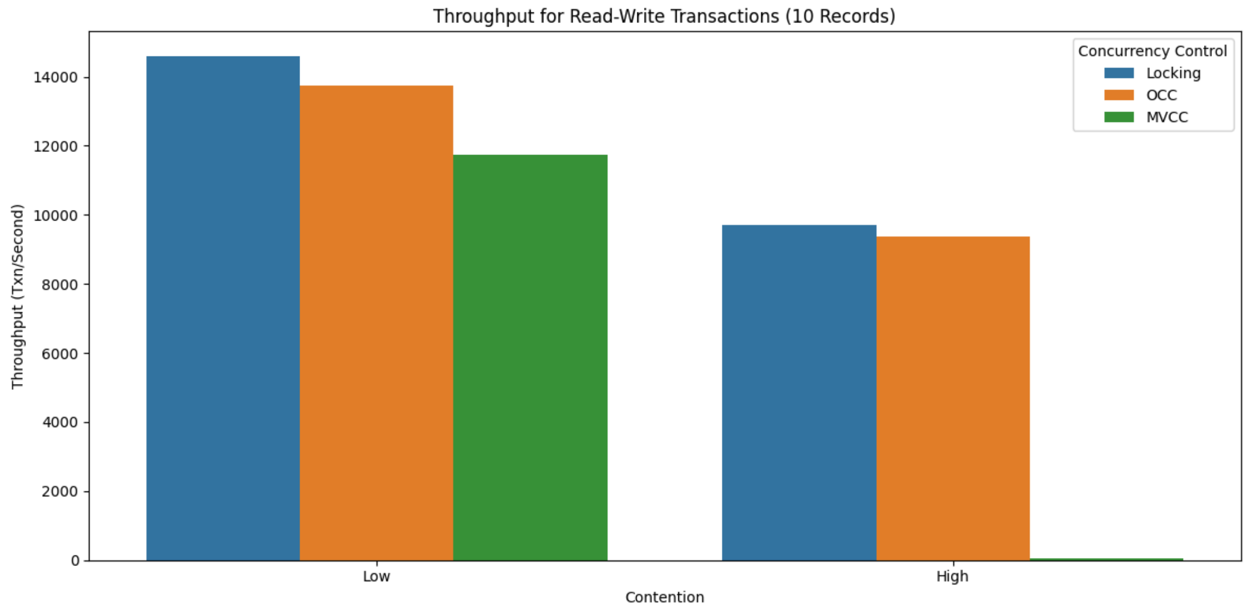
Read-Only Transactions



The figure above shows the throughput results for read-only transactions. We notice that counterintuitively, the throughput is higher in high contention in OCC and MVCC. This makes sense because we assume the caching means that these protocols will be better when they are continuously accessing data (under high contention they can only be accessing a set of 50 possible records). This means that, because of the nature of the protocols, they are able to immediately read the values and don't ever fail at validation because these are read-only transactions. Therefore, to these protocols, the difference in caching ability actually helps to increase contention helps them. Unlike OCC and MVCC, locking performs worse under high contention for read only transactions. This is mainly because we must access and update our lock queues atomically. Even acquiring only shared locks involves locking that key while we ensure there are no exclusive locks. This is further worsened by the fact that with higher contention, these queues on average will be longer with more shared locks.

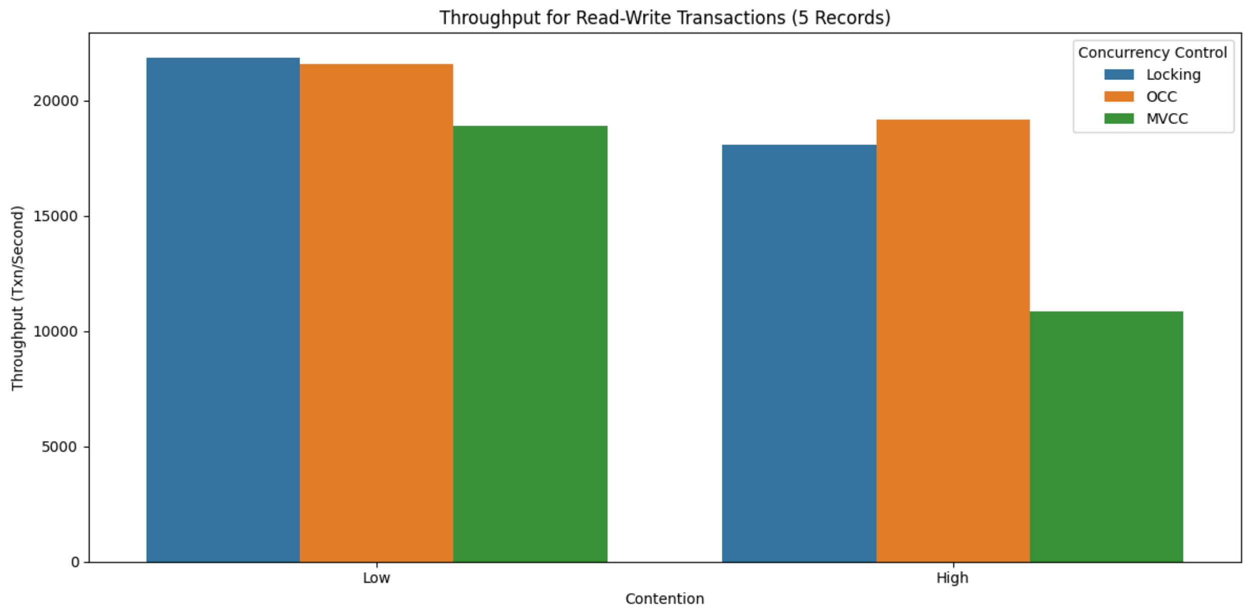
Relative to each other, we would expect locking to perform worse in both contention scenarios because of the overhead required in getting locks. Because there are no writes, we would never have an instance of deadlocking, so that is not considered in this evaluation. We would expect OCC and MVCC to perform similarly high because they both would be able to immediately process the reads according to their storage. OCC would never fail validation and MVCC would be able to read from storage according to its rules without requiring locks. MVCC's throughput abilities when compared to OCC for read-only transactions are most likely due to MVCC's ability to read from storage, while OCC must still go through the validation process (mainly phase 2 when it validates reads). Although this phase will never fail because no writes occur, it takes time to iterate over the reads and validate that the most recent CSNs have not changed.

Read-Write Transactions



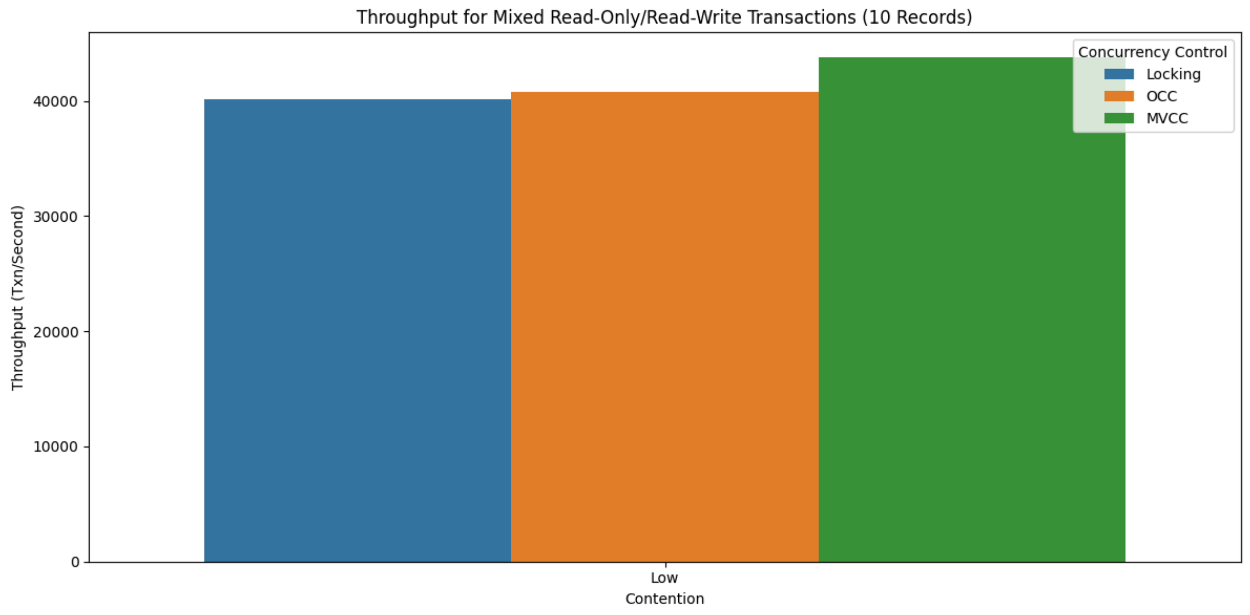
From these results we see that locking does better under low contention than the other protocols. This is out of line with what we would expect. For low contention, we would expect that OCC performs the best because at such low contention, there should virtually never be a failed validation phase. Of course, it is likely that there will be slightly more validation failures in this test case when compared to read-only transactions, so we would expect a slight dip in OCC performance compared to read-only transactions. However for low contention, we would still expect OCC to do better than the other systems. This difference, we then expect, is most likely due to implementation differences. MVCC performs the worst out of all of the concurrency control methods, likely because when it reads dirty data, its validation phase waits until the writer has completed validation. It is possible that in our benchmark where transactions perform only a small number of reads and writes, the validation phase for MVCC takes up a considerable amount of time compared to reads and writes, and this means a higher proportion of our time is spent waiting. This is compared to OCC where there is no waiting.

For high contention results, we again see locking as the best method, performing slightly higher than OCC. Compared to low contention, the proportional difference between the two is a tiny bit smaller, but still similar. OCC suffers from more failed validations, and locking suffers from increased contention when acquiring locks, which also leads to a higher deadlock detection rate. For 10 record read-write transactions under high contention, we also notice a weird anomalous result: MVCC throughput plummeets. We hypothesize that the MVCC implementation just has a lower ceiling of contention that it works up to. Our MVCC is optimistic, and in general optimistic schemes should perform drastically worse when it is not good to be optimistic for the workload. In the simplest case, OCC would also fail to perform if there was only one tuple to access. Perhaps in this case of 50 total tuples and 10 tuples per access, we have passed the threshold that our MVCC can operate at. It is also important to note that we have 16 threads per region, which exacerbates this issue. To further investigate, we also ran an experiment with 5 records instead of 10 (meaning that we would be trying to access 5 records from the 50 hot records).



When we reduce the write set to 5 records, we get MVCC throughput values of the same magnitude, although still much smaller than the throughput than the other concurrency control systems, supporting the idea of an exponential threshold for degradation. The other observations from the experiment for 10 records can be applied to this experiment for 5 records.

Mixed Read-Only/Read-Write Transactions



For this experiment, something about our testing script or our code did not allow us to run mixed read-only/read-write transactions under high contention so we only considered a low contention experiment. We observe that the throughput values for all 3 concurrency controls are very similar for mixed read-only/read-write over 10 records with low contention. We would expect that, similarly for assignment 2, MVCC is the best for this case because for the read-only transactions we are able to quickly read from storage and return to the user, while other concurrency controls, like OCC and locking have to do a validation or wait for the lock in the first place. In comparison to the read-write experiment, this situation has less contention because we are doing less writes so we should be aborting less frequently for MVCC as well. We leave this to a future investigation point.

WAL Record

While the current WAL format is human-readable and convenient for debugging, it incurs substantial space overhead. In our microbenchmark which runs for 30 seconds, the system generates approximately 125 MiB of WAL data per region. Considering the throughput, this corresponds to roughly 320 bytes per WAL record for a transaction with 10 write operations, indicating room for optimizations.

Proposed Future Experiments

For single-region transactions, our two benchmark variables are contention (database size) and transaction type (proportion of read only and read-modify-write transactions). Once multi-region transactions are supported for two regions, we will have an additional two axes to vary: inter-region latency (0-200 ms) and the percent of multi-region transactions. In the future, we aim to evaluate how the different concurrency control schemes perform for multi-region transactions under no latency, and then compare how much their performance degrades when latency is increased. In addition, we may want to convert real benchmarks into more realistic workloads, as there are far more variables to modify, especially once we support any number of regions. Ultimately, we hope to benchmark our database on real servers that are actually separated with disaggregated storage.

Conclusions

In this project, we integrated the Sunstorm architecture with various concurrency control schemes to demonstrate (as a proof of concept) that it is possible to guarantee serializability while supporting multiple writers, addressing key limitations of existing systems. Although the experimental results deviated from initial expectations, this is likely due to implementation specific divergences. We aim to develop a more stable version of the system and, if feasible, explore integration with real-world database engines in future work.

References

[1] Adopting the SunStorm algorithm to various concurrency control techniques
[2] Scaling Write Throughput in Disaggregated Storage Databases

[3] https://pages.cs.wisc.edu/~yxy/cs764-f20/papers/aurora-sigmod-17.pdf
[4] https://wzheng.github.io/silo.pdf
[5] https://www.microsoft.com/en-us/research/wp-content/uploads/2019/05/socrates.pdf
[6] https://www.vldb.org/pvldb/vol16/p3754-chen.pdf
[7]
https://dl.acm.org/doi/pdf/10.1145/3318464.3386129?casa_token=iviRQZIwQ7sAAAAA:4W5kTYtFePVDg81i_caB33xUxkBfff_CrO0rp2-4W0HzpcyiezQfvPIwkSeL6091Pyxw8WUAoC0Y
[8]
https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/limitless-architecture.html
[9] https://aws.amazon.com/blogs/database/introducing-amazon-aurora-dsql/
[10] https://ieeexplore.ieee.org/document/9835438
[11] Sunstorm Paper (not publicly linkable)
[12] https://dl.acm.org/doi/pdf/10.14778/3611540.3611542
[13] https://dbgroup.cs.tsinghua.edu.cn/ligl/papers/GaussDB-MP2024.pdf
[14] https://dl.acm.org/doi/10.1145/2517349.2522713
[15] https://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-205.pdf
[16] https://drkp.net/papers/ssi-vldb12.pdf
[17] https://dl.acm.org/doi/10.1145/2723372.2749436
[18] https://dl.acm.org/doi/10.14778/2095686.2095689