Taurus MM: bringing multi-master to the cloud

Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, Calvin Sun
Huawei Research
Toronto, Canada
firstname.lastname@huawei.com

ABSTRACT

A single-master database has limited update capacity because a single node handles all updates. A multi-master database potentially has higher update capacity because the load is spread across multiple nodes. However, the need to coordinate updates and ensure durability can generate high network traffic. Reducing network load is particularly important in a cloud environment where the network infrastructure is shared among thousands of tenants. In this paper, we present Taurus MM, a shared-storage multi-master database optimized for cloud environments. It implements two novel algorithms aimed at reducing network traffic plus a number of additional optimizations. The first algorithm is a new type of distributed clock that combines the small size of Lamport clocks with the effective support of distributed snapshots of vector clocks. The second algorithm is a new hybrid page and row locking protocol that significantly reduces the number of lock requests sent over the network. Experimental results on a cluster with up to eight masters demonstrate superior performance compared to Aurora multi-master and CockroachDB.

PVLDB Reference Format:
Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, Calvin Sun. Taurus MM: bringing multi-master to the cloud. PVLDB, 16(12): 3488 - 3500, 2023.
doi:10.14778/3611540.3611542

1 INTRODUCTION

A multi-master or multi-writer database provides a single unified view of data but allows more than one database node to update the database concurrently. Multi-master databases can be used for multiple purposes: for increased throughput and/or lower latency; for continuous availability during database node maintenance, crashes, and recovery; for on-demand scale out and scale in; and for adapting to highly variable workloads. In a single-master system, update throughput is limited because a single node handles all updates; a multi-master system removes this bottleneck [1, 14, 23, 28].

There are two main multi-master architectures: shared-nothing and shared-storage. Google Spanner, Amazon DynamoDB, CockroachDB, Alibaba OceanBase, and others have adopted a shared-nothing architecture [8, 10, 21, 31, 33, 39]. The shared-storage approach is used by Amazon Aurora multi-master, Oracle RAC, and IBM Db2 pureScale [4, 6, 38].

It is generally held that shared-nothing systems have better scalability but lower efficiency (higher overhead) than shared-storage systems. The shared-nothing architecture generates the same or more inter-node messages compared to the shared-storage architecture [6]. One reason is that shared-nothing systems employ distributed commit protocols. Such protocols degrade the performance of replicated multi-master systems due to multiple rounds of synchronous message exchange [1, 14, 23].

To achieve good performance, traditional shared-storage systems, like Oracle RAC and IBM pureScale, need high-end networking hardware [30]. The recommendation is to have a dedicated RDMA-enabled network for each cluster. This is not possible in a cloud environment, especially across availability zones. Cost savings by sharing infrastructure among tenants is a key driver in the cloud. It is difficult to provide guaranteed latency and bandwidth, similar to what on-premise solutions, such as RAC or pureScale require, at a competitive price [37].

Performance bottlenecks in distributed database systems are often caused by the limited bandwidth, high latency, and limited message rates that the network can deliver. There are two main reasons for this: high message rates and write amplification. Page and row locking and log record propagation may generate hundreds of thousands or even millions of messages per second [30]. Write amplification may be as high as 46X because of sending fully modified pages to storage [37].

Taurus MM is a cloud-native, multi-master, scaleable OLTP database system designed to achieve good performance and scalability on modest-size clusters (2-16 masters) in a cloud environment. It is designed to handle a wide variety of partitioned and shared workloads. It adopts a shared-storage approach with separate compute and storage layers. It relies on pessimistic concurrency control (PCC) instead of optimistic concurrently control (OCC). OCC suffers from high rates of transaction aborts due to write conflicts for some workloads, while aborts are very rare in PCC. Aborts not only reduce throughput and consume additional resources, but few applications handle high abort rates well.

The contributions of this paper include two novel algorithms that significantly reduce the amount of network traffic, a new database architecture, and performance evaluation.

Vector-scalar (VS) clocks algorithm. Logical clocks are commonly used to order updates and create consistent snapshots in distributed systems [5]. Lamport clocks are used, for example, by Oracle RAC

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611542

and IBM pureScale [22]. However, Lamport clocks are impractical for creating global snapshots and don't preserve causality. Instead, vector clocks or similar approaches are used [12, 25]. The disadvantage of vector clocks is high space overhead, making message size proportional to the number of nodes. Vector-scalar (VS) clocks combine the low overhead of Lamport clocks with the ability of vector clocks to preserve causality and create snapshots effectively. Rapid creation of global snapshots is important because it makes it possible to read pages without locking. Using VS clocks reduces space and network bandwidth consumption by up to 60% compared to vector clocks, even for a modest 8-master cluster.

Hybrid page-row locking algorithm. Another source of network traffic is row locking. Before updating a row, a transaction must first acquire a lock on the row to prevent other transactions from reading or changing it. We designed a protocol where row locks are granted locally by each master without contacting a global lock manager. This hybrid page-row locking protocol eliminates separate row lock and unlock requests altogether by piggybacking information on page lock requests. This noticeably reduces the number of messages being sent over the network.

Cloud-native multi-master database architecture. Our third contribution is a detailed description of the overall architecture of Taurus MM. We describe how to implement and use the above algorithms in a database system, as well as other performance optimizations, in particular, how to read pages without any locking at all.

Performance evaluation Our final contribution is an experimental performance evaluation of Taurus MM. We report its performance and scalability on workloads with different degrees of data sharing. We compare its performance on TPC-C with CockroachDB, a shared-nothing NewSQL system, and Amazon Aurora multi-master shared-data system. Taurus MM achieves 2-4X higher throughput than CockroachDB on clusters of up to 12 nodes. It scales significantly better and has higher throughput than Aurora multi-master.

The remainder of this paper is organized as follows. In the next section, we describe VS clocks. Section 3 provides a description of our hybrid page-row locking protocol. Section 4 gives a high-level overview of the single-master Taurus database, on which Taurus MM is based. We describe the Taurus MM architecture, the usage of VS clocks, and the hybrid locking protocol in Section 5. Section 6 reviews the related work, and performance results are presented in Section 7. We conclude in Section 8. The results from experiments estimating the impact of individual improvements are summarized in appendix A.

2 VECTOR-SCALAR CLOCKS

Every database system must ensure that its internal data structures remain consistent and that user data is not corrupted. For example, updates must be applied in a valid order. The ANSI/ISO SQL standard defines several transaction isolation levels that specify strict rules on data visibility that depend on the ordering of events, such as transaction start, end, and data changes [9]. Key to tracking ordering relationships is the notion of clocks, which make it possible to label events with timestamps that allow a comparison, i.e., a "happens before" relationship. In a distributed system, physical clocks running on different nodes are difficult to keep synchronized accurately enough. Having a single network-connected clock device would cause intolerable delays in obtaining timestamps. Thus, distributed systems often use logical clocks running on each node that count "events" rather than measure physical time. A logical clock is local to a node and represents the node's view of system time. A logical clock must satisfy the following requirements.

(C1) Before each event happens on a node, e.g., sending or receiving a message, or a local event, the logical clock value of the node is incremented. The new value is the timestamp assigned to the event. Thus, all local events that happened before the current event will have lower timestamps.

(C2) Messages between nodes are timestamped by the sender. Upon receiving a message, the receiver's local clock is set to a value higher than the message's timestamp.

The first logical clock algorithm was proposed by Lamport [22]. Lamport clocks are represented by a single integer counter on each node, so we call them scalar clocks. Each event happening on a node increments its local clock by some amount. Upon receiving a message with a timestamp, the local clock value is set to the maximum of the old value and the message's timestamp plus an increment. If $Clock_i$ is a logical scalar clock of node i, and $TS(m)$ is a timestamp of message m, then conditions C1-C2 are satisfied by using the following formulas:

$$Clock_i = Clock_i + s, TS(m) = Clock_i \tag{S1}$$

when message m is sent by node i.

$$Clock_j = max(TS(m), Clock_j) + r, \tag{S2}$$

when message m is received by node j. Constants s and r are often equal to 1, however, they can have arbitrary values greater than zero.

Although widely used by applications, scalar clocks have severe limitations. First, they don't preserve causality, meaning that solely from the fact that $TS(a) < TS(b)$, we cannot tell if a caused b or if a and b happened concurrently without a causal relationship between them [12, 25]. Second, scalar clock algorithms that create a distributed snapshot of the whole system require either that the system delivers messages in the order sent or that the entire message history is preserved [7, 20].

Vector clocks were proposed to address the limitations of scalar clocks [12, 25]. Each node i maintains a vector $Clock_i$ of N integers, one for each node in the system. $Clock_i[i]$ is a local clock component and describes the time progression of node i. Any event on node i causes $Clock_i[i]$ to be increased. $Clock_i[j], i \neq j$ represents the knowledge of node i about the time progression of node j. Conditions C1-C2 are satisfied by using the formulas:

$$Clock_i[i] = Clock_i[i] + s, TS(m) = Clock_i \tag{V1}$$

when message m is sent by node i.

$$Clock_j[k] = max(TS(m)[k], Clock_j[k]), 0 \leq k < N, k \neq j \tag{V2}$$

when message m is received by node j.

Vector clocks preserve causality and allow the effective creation of distributed snapshots. However, they are not space efficient: a system that uses vector clocks has to store and exchange timestamps of a size proportional to the number of nodes [5].

Vector clocks for a cluster of 8 nodes will have a size of 64 bytes. This introduces a significant increase in the amount of data sent over the network. For example, cloud databases often send log records to the storage layer and among nodes for persistence and change-tracking [3, 11, 37]. Each log record needs a timestamp, often called a log sequence number (LSN). Our experiments with the Taurus database showed that log records with scalar timestamps are 40 bytes on average. Vector timestamps increase the size to 96 bytes. Another example is lock and unlock messages for pages and rows, which happen at the rate of up to millions per second. The size of such a message in Taurus is also about 40 bytes. Vector timestamps increase log records and locking message size by 140%, consuming additional network bandwidth and disk space. In Taurus MM, two timestamps are used per record, making the difference in size even more pronounced. Several authors have showed that in cloud databases, the performance bottleneck moves to the network rather than disk or other resources [30, 37].

Vector clocks and scalar clocks have many similarities: they satisfy conditions C1-C2, use integer counters, and generate timestamps by incrementing the local clock. Although space efficient, scalar clocks have the disadvantages of lost causality and difficulties with snapshot creation. However, in many practical situations, causality is already known from the context or is not important. For example, when two events are changes to the same item serialized by a lock, their causality is known. At the same time, requests for a global snapshot happen infrequently compared to other events in the system. One might naively suggest having two types of clocks in the system: scalar and vector, using each of them depending on the context. However, in this case, the values of the clocks would be incomparable. It would be impossible to determine if a change to an item stamped with a scalar timestamp happened before or after a snapshot created with a vector timestamp. To solve this problem and create clocks that have the size benefits of scalar and support the causality and snapshots features of vector clocks, we introduce VS (vector-scalar) clocks. Depending on the usage scenario, VS clocks can stamp events and messages with either a scalar or a vector timestamp, based on the following formulas:

$$Clock_i[i] = Clock_i[i] + s, TS(m) = Clock_i, s > 0 \tag{VS1}$$

when message m is sent by node i with a full vector timestamp.

$$Clock_j[k] = max(TS(m)[k], Clock_j[k]), k \neq j \tag{VS2a}$$

$$Clock_j[j] = max(TS(m)[k], Clock_j[j]) + r, 0 \leq k \leq N \tag{VS2b}$$

when message m is received by node j.

$$Clock_i[i] = Clock_i[i] + s, TS(m) = Clock_i[i], s > 0 \tag{VS3}$$

when message m is sent by node i with a scalar timestamp.

$$Clock_j[j] = max(TS(m), Clock_j[j]) + r \tag{VS4}$$

when message m with a scalar timestamp is received by node j.

The difference between the VS1-VS2 update formulas and the V1-V2 formulas are that the local clock component of the receiving node changes and becomes greater than all other clock components. With this change, the local clock component behaves exactly as a scalar clock. At the same time, formulas VS1-VS2 are a special case of the more generic formulas V1-V2. The important consequence of the proposed change is that VS clocks behave like vector clocks, preserving the benefits of causality and effective global snapshot creation, while the local component of VS clocks behaves the same way as scalar clocks. VS3-VS4 allow us to timestamp a message with a scalar local VS clock component. Receiving such a message breaks the assumptions of vector clocks and leaves only the scalar clock valid until the node receives the next full vector timestamp.

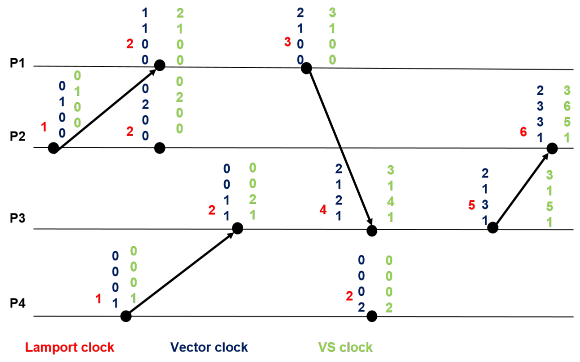


Figure 1: Comparison of vector, scalar, and VS clocks

The above properties of the VS clocks provide an important benefit. In scenarios where preserving causality or global snapshot creation are required, we timestamp each message with the vector value of the VS clocks. However, when causality is known from the context or is unimportant, we timestamp the message only with a scalar component of the clock. For example, the most common messages sent in a distributed database are page lock and unlock messages, which are required to serialize changes to the same page.

Messages requiring additional properties of vector clocks are sent much less frequently than messages that need simple ordering. This allows timestamping most of the messages with a simple 8-byte timestamp, reducing network and storage requirements.

Fig.1 shows the differences and similarities of the scalar, vector, and VS clocks by showing their values for the same sequence of events. Each message send and receive operation increments the local event counter by 1. We can see that each node's local VS clock component after receiving an event is always the same as the scalar clock value. Other components are equal to the corresponding values of the sender's vector clock at the time when the message was sent.

The usage scenarios for vector and scalar timestamps of VS clocks are different: vectors are used to establish causality and create global snapshots, while scalars are used for establishing the order of events when causality is known already. In the section describing the Taurus MM architecture, we will describe the details of how VS clocks are used.

3 HYBRID PAGE-ROW LOCKING

We assume a database system that use short-term page locks (historically also called latches) and transactional row locks to ensure the physical and logical consistency of the database. Page locks come in two varieties, shared (S) and exclusive (X). Page locks guard the *physical consistency* of a page by serializing modifications of the page among transactions. Row locks guard the *transactional consistency* of user data and are held until the end of the transaction.

To read a page, a transaction must hold or acquire an S-lock on the page. To update a page, it needs an X-lock. A transaction is allowed to release a page lock immediately after it has finished reading or writing the page, while row locks are held until the end of the transaction.

A lock manager is used to track, grant, and release locks. In a single master database, the lock manager is a local component, and all calls to it are local. In a multi-master database, the lock manager is a separate component, a global lock manager (GLM), and all calls are remote. Each master still has a local lock manager (LLM).

Taurus MM employs global page S- and X-locks to extend page protection to transactions on different masters. Global page locks are longer-lived than local page locks and are held by a master, not a transaction. A global page lock acts like a ticket; holding a global lock on page P gives a master the following rights depending on the type of lock it holds.

Page S-lock: the master can latch P in shared mode and grant shared row locks on P.

Page X-lock: the master can latch P and grant row locks on P in shared and exclusive modes. Only one master at a time can hold an X-lock on P.

A master is not required to release a page lock immediately. If it is likely to need the lock again soon, it may just keep it and only release it when the GLM reclaims it. Before releasing a page lock, a master must ensure that all its modifications to the page have been persisted, flushing the log if necessary. However, the modifications need not be committed, just persisted. Allowing a master to keep a page lock reduces round trips to the GLM, especially if the workload is fully or mostly partitioned by master. When and for how long a master can keep a page lock are policy decisions. In our current implementation, a master releases a page lock when the page is evicted from the buffer pool or when the lock is reclaimed.

3.1 Approaches to managing row locks

Database systems use several types of row locks: (plain) shared and exclusive row locks, gap locks, next-key locks, and insert-intention locks [13, 16, 24]. In this paper, the term “row lock” includes all such types of locks. Row locks can be managed in different ways; we considered three different approaches.

GLM manages row locks. One option is to have the GLM manage both page locks and row locks. Then every row lock acquisition requires a message round trip to the GLM, increasing the load on the network. DB2 pureScale uses this approach, with various optimizations. We rejected this approach because of its high network traffic, even if the workload is mostly partitioned. Furthermore, the GLM must faithfully implement all row lock types used by the underlying system and their, sometimes complicated, interactions.

Row locks stored on pages. A second option is to store the row locks of a page on the page itself. Oracle RAC implements this approach. We rejected this approach for two reasons: 1) it would have required writing row lock acquire/release records to the log, thereby increasing network load, and 2) it would have required changing the on-disk page format, forcing a database conversion when upgrading to Taurus MM.

Row locks follow page locks. When a master acquires a page lock on a page P, it also receives the list of row locks on P, both locks held and pending lock requests. The master may then grant additional row locks on P, provided they are compatible with its page lock. When it releases the page lock to the GLM, it also sends information about the current row locks on the page. Information about pending lock requests is not crucial, but it is useful for scheduling decisions at the GLM and masters. We chose this approach for reasons discussed below.

3.2 Locking protocol

In our approach, the GLM manages only page locks. It relays row lock information but does not grant or release row locks. When it grants a lock on a page P to a master, it returns the page version number, which master, if any, has the latest version of the page, and the list of row locks on the page. The receiving master then adds the row locks to its LLM. When a master releases a page lock (voluntarily or on request), it returns the page version number and the list of row locks on the page. Note that a master does not need complete information about row locks but only what is required to correctly decide whether to grant a row lock or not.

Master holds an X-lock on page P. In this case, no other master can have the page locked in any mode, but rows on the page may be locked. The current master can grant an X-lock on a row unless the row is already S-locked or X-locked. It can grant an S-lock unless the row is locked in X mode. This implies that the master must know about all the row locks currently held on P. Note that including some already-released row locks does not jeopardize correctness but may cause unnecessary waiting on a lock that, in fact, has already been released.

Master holds an S-lock on page P. In this case, other masters may also have the page S-locked, but it cannot be X-locked by any master. The current master can grant an S-lock on a row of P unless the row is already X-locked. This implies that it needs to know about all X-locks but not necessarily about S-locks. Again, including some already-released locks does not jeopardize correctness.

Master does not hold a lock on page P. In this case, the master cannot grant any row locks on the page.

Without jeopardizing the consistency of the database, row lock changes on a master do not need to be synchronized with other masters immediately; rather, it can be done on demand when a covering page lock is requested by another master. This approach uses the page lock release and reclaim flow to send row lock information to GLM. This avoids contacting the GLM on every row lock grant or wait, which can significantly reduce the amount of row-locking network traffic. Here is the flow in more detail:

- **When releasing a page lock** (voluntarily or when reclaimed):
  - *On the master:* information about all row locks on the page is sent to GLM.
  - *On the GLM:* the received row lock information is cached in memory.
- **When requesting and granting a page lock:**
  - *On the master:* to grant a row lock requested by a local transaction, the master must hold a covering page lock in a sufficiently strong mode. If the master does not currently have the required page lock, it first sends a page lock request to the GLM.

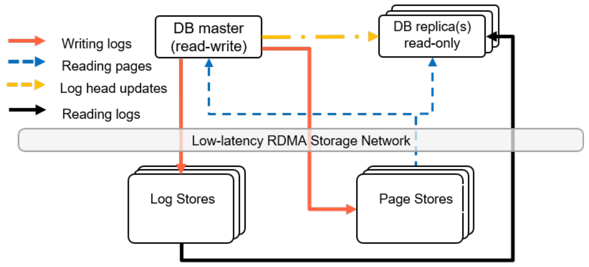


Figure 2: Taurus components and layers

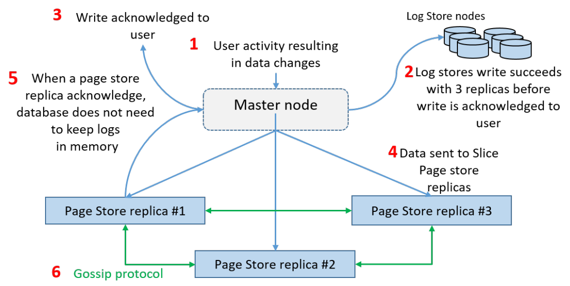

3 Write acknowledged to user
1 User activity resulting in data changes
Log Store nodes
2 Log stores write succeeds with 3 replicas before write is acknowledged to user
5 When a page store replica acknowledge, database does not need to keep logs in memory
Master node
4 Data sent to Slice Page store replicas
Page Store replica #1
Page Store replica #3
Page Store replica #2
6 Gossip protocol

Figure 3: Taurus write path

– *On the GLM:* When a page lock request arrives, there may be other pending requests for locks on the same page, and the request may have to wait. When the GLM is ready to grant the lock, it first reclaims conflicting locks on the page, if any, and captures the newly received row lock information. It then grants the lock and sends the response, together with the row lock information and the latest version number of the page.
– *On the master:* on receiving the grant response, the row lock information is added to the LLM, and the transaction attempts again to acquire the desired row lock.

4 TAURUS SINGLE MASTER

Taurus MM is a further development of Taurus single master (later referred to as just Taurus in contrast to Taurus MM), a relational database designed specifically for cloud environments with separate computer and storage layers [11]. Taurus has been offered for several years as a part of Huawei cloud services under the name GaussDB for MySQL. Taurus offers read replica support, fast recovery, and hardware sharing. The high-level Taurus architecture is presented in Fig. 2.

The Taurus compute layer consists of a master and multiple read-only replicas. Update transactions generate log records, which the master ships to the storage layer. No full pages are written across the network, reducing the required bandwidth. The compute layer running a modified version of MySQL is responsible for accepting incoming connections, executing queries, managing transactions, and producing log records that describe modifications made to database pages.

The Taurus storage layer consists of Log Stores and Page Stores. Log Stores serve two purposes. First and foremost, they ensure the durability of log records. Once all log records generated by a transaction have been made durable, transaction completion can be acknowledged to the client. Second, they serve log records to read replicas so that the replicas can apply them to bring pages in their buffer pools up to date. The master periodically communicates the location of the latest log records so that read replicas can read the latest log.

The master also distributes log records to Page Store servers. Their main function is to create new versions of pages by continuously applying the received log records to previous page versions. Taurus divides a database into small (10GB) sets of pages, called slices. Each Page Store server handles multiple slices from different Taurus instances. A database has multiple slices, and each slice is replicated to three Page Stores for durability and availability.

The write flow is summarized in Fig. 3. When a log record is generated on the master by a user transaction (Step 1), it is stamped with the local clock value. This stamp, an 8-byte integer, is called a log sequence number (LSN) and is used to order all changes in the database. To make log records durable, the master writes log records, accumulated in batches called log flush buffers (LFB), to three Log Stores (Step 2). Each LFB contains the largest LSN of the previous LFB to establish order and detect missing log messages. Once all Log Stores acknowledge the write, the database considers the data to be persistent and the write complete. Transactions whose commit depends on the write can then be marked as committed (Step 3).

Once a log record has been written to the Log Stores, the master copies it into the write buffer of the slice that contains the page. When the buffer is full, it is sent to the Page Stores (Step 4). Each buffer also contains the LSN of the last record of the previous buffer so buffers can be ordered in Page Stores. The master waits for a reply from one of the Page Stores and releases the buffer (Step 5). Page Stores that host replicas of the same slice also periodically exchange messages with each other using a gossip protocol to detect and recover missing buffers (Step 6).

The operation of read replicas is shown in Fig. 4. When the master updates the database by writing log records to Log Stores (Step 1), read replicas get notifications that include the location of log records (Step 2). Next, replicas read all log records from the Log Stores in order to update pages in its buffer pool (step 3). Finally, read replicas also read pages from Page Stores (step 4) as needed.

An important challenge is maintaining a consistent view of the data on read replicas. There are two types of consistency. First, physical consistency refers to the consistency of internal structures in the database, such as B-tree pages. For example, splitting a page in an index tree modifies multiple pages. A read replica that traverses the B-tree must observe changes to these pages as if they happened atomically. On the master, physical page consistency is achieved by locking the pages before they are modified. However, it would be suboptimal to coordinate locks with read replicas. To avoid explicit synchronization, the master writes log records in groups, always setting the group boundary at a physically consistent point. Read replicas read and apply log records atomically per these group boundaries keeping its database view physically consistent. The LSN of the last log record processed by a read replica represents

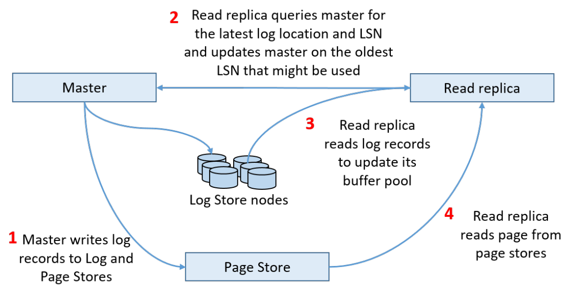

2 Read replica queries master for the latest log location and LSN and updates master on the oldest LSN that might be used
Master
Read replica
3 Read replica reads log records to update its buffer pool
Log Store nodes
4 Read replica reads page from page stores
1 Master writes log records to Log and Page Stores
Page Store

Figure 4: Read replica workflow

the replica's physical view of the database and is called the replica visible LSN.

A read replica reads and parses the log from Log Stores and continuously advances its visible LSN. When a read transaction needs to perform an operation that requires physical consistency (e.g., an index lookup), it creates its own physical view for the duration of the operation by recording the current replica visible LSN, called a transaction visible LSN (TV-LSN). Different transactions can have different TV-LSNs. While the read replica keeps advancing its visible LSN, a transaction's TV-LSN can lag behind. Since such operations are short, read replicas advance TV-LSN fairly quickly.

Many databases, including MySQL, maintain multiple versions of rows to reduce conflicts between readers and writers. Logical consistency refers to the consistency of user data as required by the transaction's isolation level. When a write transaction commits on the master, a commit record is written to the log. While parsing the log, a read replica updates its active transaction list. When a read transaction starts on the read replica, it records the active and committed transaction list. This list determines a transaction's logical data view, i.e., which data is visible to a transaction.

The buffer pool on a read replica can store multiple versions of the same page. As the read replica reads and parses the log, it applies the log records to the buffer pool pages and produces newer versions of the pages on demand. This way, the read replica already has most of the frequently used pages in its buffer pool, thus relieving pressure on Page Stores.

5 MULTI-MASTER ARCHITECTURE

The Taurus MM high-level architecture is shown in Figure 5. It is based on Taurus and reuses basic ideas from the read replica support described in Section 4. Taurus MM has a shared-storage architecture where both Log Stores and Page Stores are shared among all masters. It uses pessimistic concurrency control and a Global Lock Manager (GLM), as described in Section 3.

Each master maintains its own write-ahead log (WAL). A user transaction executes on a single master – there are no distributed transactions. Log records are written to the WAL of the master executing the transaction. Each master periodically sends the location of newly generated log records to all other masters. Using this location information, a master reads log records generated by all other masters and updates pages in its own buffer pool. Like

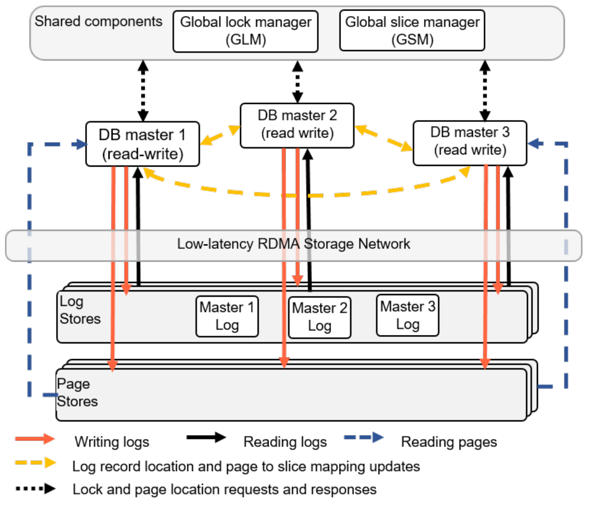


Figure 5: Taurus MM components

single-master Taurus, logs of all masters are sent to Page Stores, which update pages continuously and serve read page requests.

The Global Slice Manager (GSM) is responsible for creating new slices and assigning newly created pages to a slice. The GSM is also responsible for persisting the list of slices and page-to-slice mappings. At startup, each master reads the complete set of mapping information and caches it locally. Pages are allocated and mapped to a slice in batches of 4MB. Every time a new allocation is made, it is asynchronously propagated to all masters in the cluster. This mechanism ensures that calls to the GSM are infrequent and do not affect performance in a noticeable way.

In the remainder of this section, we will describe in detail how the concepts of VS clocks and hybrid page-row locks are implemented and used to minimize network utilization and improve the performance of Taurus MM.

5.1 Ordering and consistency

Taurus MM must maintain two types of consistency: physical and logical. Physical consistency refers to the consistency of internal data structures. For example, pages containing data must be recorded as in-use in database metadata, and B-tree parent nodes must point to the correct child nodes. Logical consistency refers to the consistency of user data, which in the case of Taurus MM, guarantees conformance to ANSI SQL [9]. For example, no uncommitted changes may be visible to any transaction apart from the one that made the changes.

A single-master database relies on a sequence of ordered atomic operations to ensure physical consistency in the face of multiple concurrent updates. To perform an atomic operation, a thread obtains one or more page locks and keeps them until the data is back in a consistent state, thus preventing other threads from observing inconsistencies. For example, when adding a row to a page, the page is X-locked, after which space is allocated, the new row is added to the page, and the X-lock is released. Ordering of log records is ensured by using an LSN counter to stamp log records from every database change operation, ensuring a total ordering.

In a multi-master database, several masters update the database simultaneously, so using local locks is not sufficient. Also, using a global clock service to totally order change operations would require a message round trip to obtain a global timestamp for every operation, which is not practical. Taurus MM uses a Global Lock Manager (GLM) to maintain global page locks for atomic page operations, while the ordering of operations is done by using VS clock timestamps.

As described in Lamport and vector clock algorithms, messages between nodes that carry information about system state must carry timestamps [22, 25]. For Taurus MM, system state is the data stored in the database, including internal structures and user data. Log records and full pages are messages and must be timestamped. These messages can be sent from master to master directly or indirectly by writing them to and then reading them from Log or Page Stores. Masters generate events that change the database state and need to create timestamps. Timestamps are used to order messages and to create a globally consistent snapshot of the system.

Since we are using VS clocks, an important question is whether to use scalar or vector timestamps for each particular message type. The overall goal is to minimize the use of full vector timestamps as they have higher network bandwidth and disk space overhead. In Section 2, we determined that when causality is known from the context, a message can be stamped only with a scalar component of the clock. Below we will describe in more detail when and how to use vector or scalar timestamps of VS clocks.

5.1.1 Ordering page versions and log records. Each log record reflects a change to a single page. Also, all changes to a page are known to be causally dependent because a master can only modify the latest version of a page, and operations on a page are serialized. Consequently, pages and log records can be stamped with the scalar component of the VS clock of the node that modified the page. This timestamp serves as the LSN for the log record and the page.

When a page is to be modified by a master that does not have the page X-locked, the master sends an X-lock request to the GLM. The GLM grants the lock and returns the latest scalar timestamp of the page. The master then updates the local component of its clock using formula VS4. Using this timestamp, the master checks whether it already has the latest version of the page in its buffer pool or needs to read it from the Page Store. When a new version of a page is created, the new version and the corresponding log record are stamped with the new timestamp using formula VS3. All log records also contain the previous timestamp of the page. When a page is unlocked by a master, the unlock message to the GLM includes the page's scalar timestamp, which is then passed to the next master to be granted the lock. This way, all log records for a given page can be ordered by using the page LSN, and gaps created by missing log records can be detected using the previous timestamp.

5.1.2 Ordering log flush buffers. As mentioned above, log records are buffered and sent to Log and Page stores in the form of LFBs containing multiple log records, possibly from multiple transactions. As an LFB is a message, it needs to be timestamped. An LFB contains log records for multiple pages, so we cannot know in advance if

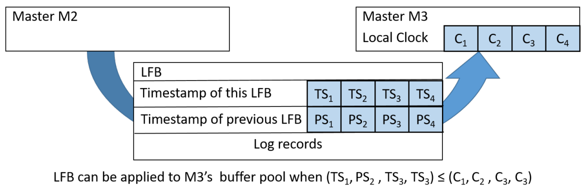

Master M2
Master M3
Local Clock
C_1 C_2 C_3 C_4
LFB
Timestamp of this LFB
TS_1 TS_2 TS_3 TS_4
Timestamp of previous LFB
PS_1 PS_2 PS_3 PS_4
Log records
LFB can be applied to M3's buffer pool when $(TS_1, PS_2, TS_3, TS_3) \leq (C_1, C_2, C_3, C_3)$

Figure 6: LFB timestamp and master clocks

one LFB is causally dependent on another one, as they may affect different pages and may or may not be generated independently. Thus, to establish causal dependency between LFBs, it is not enough to stamp them with scalar timestamps; a full vector clock timestamp is required. Each master, before sending an LFB to a log store, stamps it with the current vector value of its VS clock. Also, the timestamp of the previous LFB is written so that the receiving side can detect a missing LFB.

Page stores receive LFBs from masters and apply log records to the pages. Page Stores operate on each page independently, applying records in the order of their timestamps. Since all changes to the same page are known to be causally dependent, Page Stores use the scalar timestamp of each individual log record, ignoring the vector timestamp of the LFB.

Each master also receives LFBs from every other master by reading them from the Log Stores. However, masters do not treat each page independently. For example, when a master traverses pages of a B-tree to find a specific record, all page versions read must be consistent to ensure correctness. For a single master, this consistency is achieved by writing LFBs at physical consistency points. A read replica can then update pages in its buffer pool by reading and applying the master's log in order, at LFB boundaries. However, in the multi-master case, there are multiple logs, and log records in a log from one master may have causal dependencies on log records from another master. This means causality dependency should be enforced during every log read operation to guarantee a consistent snapshot of the database. As discussed in Section 2, creating a consistent snapshot requires using full vector timestamps. To be causally consistent, master A reading an LFB from master B's log must be aware of all changes that master B was aware of when writing this LFB. Consequently, all components of master A's clock, except the one that corresponds to master B, must be no less than the corresponding component of the timestamp. The master A's clock component that corresponds to master B should be equal to the master B component from the timestamp of the master's B previous LFB, which means that we have processed the previous LFB from master B. Fig. 6 has an example of the criteria used to check if an LFB from master M2 can be applied immediately on master M3 or needs to wait for other LFBs. When the LFB is processed, the processing master's clocks are advanced using formulas VS2(a-b).

5.1.3 Consistent page reads. One way to ensure physical consistency of an operation that requires reading multiple pages is to use locking reads. Locking reads acquire a lock on each page involved and ensure that we read the latest version of the page. However, this results in a multitude of network locking requests that slow down performance. A more efficient alternative, viable in many cases, is to use versioned reads. The idea is to fix the point in time and read all pages involved as of this point. Versioned reads are useful for cases when reading an older version of a page suffices, such as when there is no intention to modify data being read under read committed or snapshot isolation. Versioned reads avoid global page locks reducing network traffic and improving performance.

The local VS clock on a master encodes the progress of the master and its knowledge of other masters' progress, representing a globally consistent view of the system on this master. It corresponds to the replica visible LSN for a single-master Taurus database described in Section 4), and for this reason, the current value of the clock is called a visible LSN in Taurus MM as well. As in the single-master replica case, when a transaction needs to perform a multi-page read operation (e.g., an index lookup), it creates its own physical read view of the database by recording the current master clock value, which we call the transaction visible LSN (TV-LSN). Whenever a page needs to be read, it is read either from the local buffer pool or brought in from a Page Store by asking it to produce the latest version of the page prior to the TV-LSN. Thus, the TV-LSN represents an instantaneous consistent snapshot of the database. We discuss the implementation details of version reads in Section 5.3

5.1.4 *Strict transaction consistency.*

A master's knowledge about the progress of other masters advances as it reads their logs. However, this knowledge lags relative to real time. An application may receive an update commit message, while some masters are still not aware of it. In this case, the application trying to read from another master, might get an outdated result. Some applications can tolerate out-of-date results, however, a significant number cannot [10, 26].

In order to guarantee reading the latest data, Taurus MM uses VS timestamps. A dedicated thread on each master continuously sends a message to every other master requesting the VS timestamp of the last LFB. Once replies from all masters are received, the next messages are sent, thus capping the rate of messages exchanged. When a transaction specifying that it requires strict consistency arrives at a master, the master puts it on hold until the next timestamp request is sent and replies are received. The received timestamps are combined by selecting the maximum across all replies for each vector component. The resulting vector timestamp is not less than the timestamp of any master in the cluster as of the time when the transaction arrived. In other words, it includes all changes to the database prior to when the transaction arrived. The transaction then waits until the local master's VS clock passes this timestamp and, by that time, the local master is aware of all database changes that happened before the time when the transaction began. This consistency level can be configured at transaction granularity, allowing maximum flexibility for a user.

Summarizing the contents of this section, we described situations when scalar and vector timestamps of VS clocks are required. When stamping log records and pages, lock messages, and ordering log records, scalar timestamps are sufficient, as these operations involve only one page plus the knowledge that all operations are causally dependent. When updating a master's buffer pool with the contents of an LFB, creating a physically consistent read view, or supporting strict consistency, vector VS timestamps are required, as these operations involve multiple pages, and causality cannot be determined without timestamps.

5.2 Lock processing

This section provides additional implementation details of the handling of page locks and row locks in the global lock manager (GLM) and in the masters.

5.2.1 *Lock processing in the GLM.*

The GLM is responsible for granting, reclaiming, and releasing page locks. When granting a page lock, the GLM also sends the requesting master information about row locks currently held on the page. The master needs this information to avoid granting conflicting row locks. When the GLM processes a page lock request, it checks if the requested lock is compatible with existing locks, if any, on the page. If the lock cannot be granted immediately, the GLM sends a lock reclaim request to all masters holding the lock in conflicting modes. When a master releases a lock, it sends information about all row locks on the page back to the GLM for processing.

The GLM also keeps track of completed remote transactions. For each received row lock, if the owning transaction has completed already, the GLM discards the row lock. If the row lock is new, it is added to the GLM's row lock information. A row lock that exists in the GLM but is not found in the reclaim response message is removed from the GLM since either the row lock has been released or moved to a different page.

5.2.2 *Page lock processing in a master.*

When a master receives a lock-granted message from the GLM, it merges the row lock information contained in the message with its local row lock information. A master also keeps track of terminated remote transactions. If the transaction owning a row lock has terminated already, the master discards the row lock. A row lock owned by an active transaction that occurs in the lock-granted message but not on the master is added to the LLM.

5.2.3 *Row lock handling in a master.*

When a transaction terminates, it releases all its row locks. If there are transactions waiting for a released row lock, one of them is chosen to be the next lock owner. To grant a row lock, however, the covering page lock must be held as we have described. The protocol uses an approach that avoids adding extra page lock requests just to grant row locks to a waiting local transaction.

When a master learns that a transaction has terminated, its LLM examines the row locks released by the transaction. A local transaction that is waiting for a released row lock is woken up if the LLM determines that the transaction can get the lock according to the scheduling policy. At that point, if the master is holding the covering page lock, the row lock is granted to the transaction immediately. Otherwise, it waits until the master has acquired the covering page lock.

Once the page lock is re-acquired, the transaction confirms that no other transaction is holding the row lock and converts the row lock from a grant-pending to a granted state. If the row lock has been granted to another transaction or the transaction now requires a stronger and incompatible row lock mode, the transaction updates the row lock waiter information, resets the grant-pending state, goes into a wait state, and retries the operation later when it is woken up again. At that point, the covering page lock can be reclaimed by the GLM.

5.2.4 Row lock migration. In a B-tree index, a row can be moved to a new page by a page split or page merge. A page split occurs during a row insert or update when the target page does not have sufficient space. A new page is allocated, and some rows from the target page are moved to the new page. A page merge occurs during a row delete when the space used on the target page falls below a threshold. Rows from the target page are moved to an adjacent page with sufficient space, and the now-empty page is deleted. If a moved row is locked, the row lock must also be migrated to the corresponding page.

5.2.5 Row lock cleanup. When a transaction terminates, it releases all its row locks locally but some or all of the row locks may be cached by other masters or by the GLM. We must ensure that these now stale copies of the row locks are eventually deleted. When a transaction terminates, it does not notify the GLM immediately to delete its row locks. Each master periodically (e.g., every second) sends the IDs of terminated transactions to the GLM, which then garbage collects its cached row locks.

Cleanup of the stale copies of row locks on a master is built on top of the SQL read replica's transaction handling mechanism in the Taurus single-master architecture. When a write transaction ends, a commit record is written to the log, and it is read by remote masters, which determine what row locks were held by the terminated transaction and deletes them from their LLMs.

5.2.6 Handling of shared row locks. To reduce network messages, the protocol allows multiple masters to independently grant shared row locks without immediate synchronization while holding a covering page lock in a compatible mode.

Since different masters can grant S-locks independently, a master's view of row S-locks on a page can be different and incomplete until row lock information is merged by the GLM and re-distributed to masters together with new page lock requests. Even though there is a time window in which knowledge about all shared row lock owners is distributed across the cluster, with each master and the GLM potentially having only partial knowledge about the shared row locks in the cluster, this does not pose any data correctness or consistency danger. Since S and X are incompatible lock modes, when a master grants a row S-lock while holding the covering page lock in S, it can be certain that no other master can possibly be holding the same page lock in X. Furthermore, as a consequence of the rule that a row X-lock can only be granted while holding the covering page X-lock, the master can also be certain that no other transaction can possibly have granted an X-lock to the same row.

For a local transaction holding the row lock in S, the only possibilities are 1) no other transaction is holding the same row lock or 2) one or more transactions are holding the same row lock in S. Neither case would impact the consistency of reading the row.

When another transaction wants to grant a row X-lock on that page, it must first acquire the covering page X-lock. This causes the GLM to reclaim the page lock and forces a synchronization of all row locks held on the page by different masters. The GLM merges the received row locks into a globally complete view and forwards it to the requesting master.

5.3 Versioned reads

As described in Section 5.1.3, versioned reads are provided by creating a globally consistent snapshot using a vector clock timestamp, called a TV-LSN. Although, theoretically, any page version has to be read as it was at this timestamp, in practice, there is a difficulty associated with local changes. Changes to a page coming from remote masters are received using a flush buffer that merges dozens of changes to each page into one. However, local changes to a page are created one by one, and multiple page versions produced by the local master could quickly pollute its buffer pool and slow down execution. The solution to this problem is that the master must always observe the latest local changes to a page.

However, reading the latest locally modified version of a page may result in a consistency violation, when at least one of the page's prior versions is not visible according to the TV-LSN timestamp of the globally consistent snapshot. For example, if a page's version chain is (M2, 20), (M1, 30), then a read of the page using TV-LSN timestamp [30, 10] is invalid because, according to the snapshot timestamp, page version (M1, 30) is visible, but the page version (M2, 20) is not visible. In other words, for a versioned read to be consistent, it must satisfy the following property: if a version of a page that was read was modified locally, then all its previous versions must have been created earlier than the TV-LSN timestamp. Violation of this property renders the read inconsistent and thus invalid. When such an invalid read is detected, we abort the current snapshot and fall back to locking reads. Experiments show that such invalid reads are very rare.

Even when doing a versioned read, we need to ensure that local transactions do not modify the local copy of the page being read. To guarantee this, a master needs to lock the page locally, even for versioned reads. However, local locks do not affect the network because this lock is intended to prevent only local page modifications. Thus, a Taurus MM versioned read is a hybrid of a read replica versioned read and a single-master locking read.

6 RELATED WORK

Taurus MM uses a shared-storage approach in order to minimize network utilization for a broad class of OLTP workloads. IBM's Db2 pureScale also uses a shared-storage approach. It relies on a centralized caching facility (CF) node to act as a global buffer pool for dirty pages. If a master needs to read a previously modified page, it reads the page from the CF node. After a master modifies a page, it also copies the page to the CF node. The resulting network load requires a dedicated high-end network connection between the masters and the CF node [38]. Also, the memory requirement for the CF is as high as 40 % of the sum of memory sizes across all masters limiting the scalability of pureScale [30].

Oracle's RAC architecture also uses a shared-storage approach, as well as a distributed lock manager and buffer pool [27]. However, it does not have a separate storage layer that updates pages based on log records, thus causing high network utilization due to write amplification and a higher load on masters due to the flushing of full pages. If a RAC node has a slightly outdated version of a page, it asks the page owner for the current holder of the latest version of the page and then asks the holder for the desired version of the page. Taurus MM, on the other hand, transfers only log records between masters, reducing network utilization. Oracle RAC stores row locks on the page itself, using an interested transaction list area of each page. If space is not available for a row lock, the transaction requesting the lock fails. This approach places a limitation on the number of transactions that can lock a row. In contrast, Taurus MM never writes full pages to disk which reduces the required network bandwidth. Also, it does not store row lock information on pages, thus not limiting the number of row locks and reducing the number of page modifications. Finally, separate storage and compute layers allow Taurus MM to share storage nodes among multiple tenants, reducing costs and improving scalability.

Aurora multi-master is another commercially available multi-master database with a shared-storage architecture [4]. Like Taurus MM, it is designed for the cloud with separate storage and compute layers, and avoids write amplification by shipping logs from masters to storage nodes. Aurora multi-master does not have a lock manager. Instead, it relies on optimistic concurrency control implemented in the storage layer. When a log record reaches a storage node, the node checks the log record for possible conflicts and rejects the change if there is a conflict. The master that produces a log record collects the replies from all storage nodes and aborts the transaction if the log record is rejected by storage nodes. This approach results in frequent transaction aborts on a workload with substantial data sharing among masters. Applications may need to be modified to handle these transaction aborts. Performance and scalability are also affected by executing and rolling back aborted transactions. Taurus MM only aborts transactions when there is a deadlock.

Using numerical counters as logical clocks was first proposed by Lamport [22]. Later an algorithm for global snapshots using Lamport clocks was proposed [7]. However, restrictions of this algorithm and the inability to preserve causality resulted in the invention of the vector clocks [12, 25]. The space overhead of vector clocks prompted multiple optimizations [2, 10, 17, 34]. However, according to other publications, these optimizations increase algorithm complexity, compromise accuracy, and, in the worst case, take as much space as vector clocks [17, 18]. The VS clocks used by Taurus MM can produce scalar or vector timestamps, depending on the use case, without sacrificing simplicity and accuracy, allowing space-efficient timestamps to be used in most use cases.

Hybrid Logical Clocks (HCL) combine physical time information with scalar clocks in one timestamp [19, 36]. This timestamp values close to physical time and provides a method for creating consistent snapshots. HCL is used by MongoDB and CochroachDB. However, similar to Lamport clocks, HCL does not preserve causal dependencies from timestamps. It is not possible to say based only on the timestamps of two events if the events are not causally dependent. VS clocks fully preserve dependencies which allow us to do important optimizations, such as allowing us to apply LFBs read from masters immediately if they are not causally dependent on other master's LFBs, as described in the section 5.1.2.

7 EXPERIMENTAL EVALUATION

In this section, we report experimental results for Taurus MM. We begin by measuring absolute performance and scalability. Next, we compare Taurus MM with Amazon Aurora multi-master (Aurora MM) - the only other cloud-native shared-storage database that we know of. Then we compare Taurus MM with CockroachDB, which is based on a shared-nothing architecture.

7.1 Experimental setup

We ran our experiments on a cluster with up to 8 master nodes. Each node had two Intel Xeon Gold 6278C 2.6GHz CPUs running CentOS 7. Buffer pool size was 128GB. Each master was restricted to use only one CPU with 28 cores. The workload drivers ran on the same machine but were restricted to the second CPU. The masters were connected via a 25Gbps network. In addition, we deployed the storage layer (Slice Stores and Log Stores) on 4 nodes with the same hardware configuration.

We used two standard workloads for all our experiments: Sysbench and a TPC-C variant created by Percona [29, 32, 35]). SysBench is a popular benchmark that generates an adjustable mix of insert, delete, update, point-select, and range queries. We extended SysBench to make it possible to control the degree of data sharing. On a cluster of N masters, we logically divided the tables into N + 1 groups. The tables in the first N groups were private, i.e., each group was assigned to a separate master, and only the designated master accessed tables in the group. The last group was shared, i.e., any master could access tables in this group. When an experiment specified a sharing degree of X%, X% of queries were made against the shared tables, and the rest against the master's private tables. A fully partitioned workload corresponds to X=0%, and a fully shared workload to X=100%. In our setup, each group consisted of 100 tables for a shared workload and 200 tables for a fully partitioned workload. Each table had 2.35M rows, so that each master accessed 100GB of data. The TPC-C benchmark is an industry-standard benchmark for evaluating the performance of OLTP systems. Data was partitioned by warehouse across masters, but 10% of the transactions accessed data in another master's partition. Unless mentioned otherwise, we used 1000 warehouses.

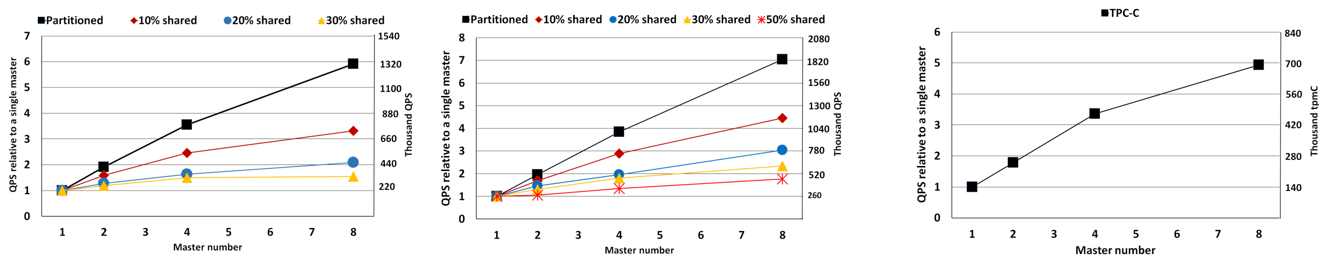
7.2 Taurus MM overall performance

Taurus MM performance on SysBench write-only, SysBench read-write (80% reads, 20% writes), and the TPC-C benchmarks are presented in Fig. 7(a-c). On the X-axis, we vary the cluster size from 1 to 8 masters. On the Y-axis, we show throughput relative to single master as well as absolute throughput. Each line corresponds to a different data sharing. TPC-C has a fixed 10% of shared queries.

Results for the read-only workload are omitted as it scales perfectly due to versioned reads. The fully partitioned workload scales nearly linearly with the number of masters. Sysbench write-only and read-write with 10% sharing and TPC-C workloads achieve 3.5x, 4.5x and 5x speedup, respectively, on a cluster with 8 masters. As expected, on workloads with higher degrees of data sharing, scalability suffers, with write-only and read-write workloads achieving less than 2x speedup on an 8 node cluster at 30% and 50% sharing, respectively. Due to the limited hardware available, we were not able to get results for a cluster of 16 masters, but scalability up to 8 nodes was close to linear.

7.3 Comparison with Aurora MM

Fig. 8 compares Taurus MM with Aurora MM using the same number of cores and buffer pool memory. In all tests, we observed that



(a) SysBench write-only (b) SysBench read-write (c) TPC-C

Figure 7: Taurus MM performance on SysBench and TPC-C

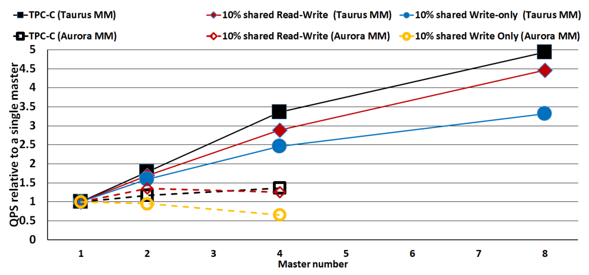


Figure 8: Taurus MM vs. Aurora MM

Table 1: Taurus MM vs. Cockroach DB: TPC-C results

		Taurus MM		CRDB	
		1000w	5000w	1000w	5000w
6 nodes	tpmC	250000	216000	121000	137000
	Latency (ms)	17/48	16/35	90/150	220/620
12 nodes	tpmC	691000	734000	164000	279000
	Latency (ms)	21/106	19/80	150/300	590/1280
Scale factor		2.8	3.4	1.4	2
Efficiency		0.7	0.8	0.7	1.0

single master performance of Taurus MM exceeded that of Aurora MM. However, Aurora MM does not expose hardware details, apart from the compute layer, so we report only performance numbers relative to a single master. Aurora does not allow more than 4 masters, so 8 master performance is measured only for Taurus MM. Both databases scale nearly identically for fully partitioned workloads, thus partitioned results are omitted. However, Taurus handles workloads with even a small degree of data sharing much better. In Aurora with shared write workloads, we have observed a large number of conflicts resulting in the majority of transactions being aborted on all masters but one and uneven performance between masters. Taurus MM did not experience any transaction aborts due to its lock-based concurrency control.

7.4 Comparison with CockroachDB

In this section, we compare Taurus MM with a system based on a shared-nothing architecture. We chose CockroachDB (CRDB), which is open source and has demonstrated better performance than Spanner and TiDB [15, 33]. We deployed CRDB on the same cluster as Taurus using two configurations with 6 and 12 nodes, respectively. The goal was to compare the performance of the systems using identical hardware. Since Taurus uses 4 nodes exclusively for the storage layer, and CRDB combines compute and storage layers on each node, we compared the 6 and 12 node CRDB configuration with 2 and 8 masters in Taurus MM, respectively. For both CRDB and Taurus MM, we experimentally determined and used the number of connections that maximizes the throughput. For Taurus MM, we used 64 connections per master, and for CRDB, the number of connections varied from 128 to 512 per node.

For CRDB, we ran the TPC-C like benchmark that comes with CRDB. For Taurus MM, we ran the Percona variant of TPC-C. TPC-C think/keying time was set to zero for both benchmarks. We present results for 1000 and 5000 warehouses in Table 1. For each run, we recorded New Order transactions per second (tpmC) as well as transaction average/95% latencies. In all cases, Taurus MM throughput was noticeably higher, varying from 60% higher for 6 nodes and 5000 warehouses to 320% higher for 12 nodes and 1000 warehouses. Taurus MM also had considerably lower transaction latency, both average and 95%. In addition, we report a *scale factor*, i.e., relative throughput increase between 6 and 12 nodes, and *efficiency*, i.e., scale factor divided by the relative increase of the number of front-end nodes (2x for CRDB and 4x for Taurus MM). As noted in the Introduction, a shared nothing architecture is subject to overhead of the distributed commit, which grows with the number of nodes involved in a transaction. For a given workload, this number is limited by the complexity of transactions. Once the number of nodes in a cluster exceeds this number of nodes involved in a transaction, scalability becomes ideal. On a smaller database with 1000 warehouses, scaling efficiency of Taurus and CRDB are the same. However, on a larger database, where amount of data conflicts is smaller, CRDB demonstrates more efficient scaling.

8 CONCLUDING REMARKS

Taurus MM is a multi-master OLTP database system specifically designed for cloud environments. It is a shared-storage system with separate compute and storage layers and uses a global lock manager to coordinate read-write access to database pages. Many OLTP workloads are mostly partitionable where only a small fraction of pages are shared among multiple clients. For example, in TPC-C, only 10% of accesses are to a “foreign” warehouse. A key goal of

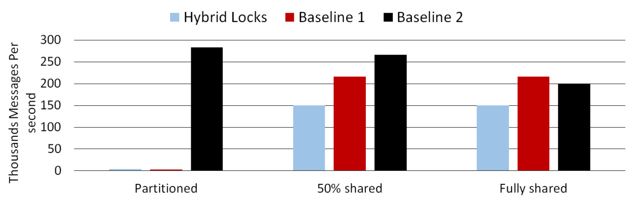


Figure 9: Impact of hybrid page-row locks

the design was to achieve good performance and scalability on workloads with a low degree of sharing.

Taurus MM incorporates two key innovations aimed at reducing network load and improving performance: vector-scalar (VS) clocks and hybrid page-row locking. VS clocks reduce network load by allowing the most frequent messages (log records) to be timestamped with a single scalar timestamp and far fewer messages with a vector timestamp. With VS clocks, the system retains the ability of transactions to see a system-wide consistent state. The purpose of hybrid page-row locking is to improve transaction latency and throughput by reducing the number of lock requests sent to the global lock manager. In particular, if a page is accessed by a master for some time, locking is automatically delegated to the master. Our experimental results confirm the performance and scalability of the system. On the TPC-C benchmark, scaling efficiency was 84% up to four masters and 62% up to eight masters, achieving a total throughput of 734,000 tpmC with eight masters. Experiments with the SysBench benchmark showed, as expected, that scalability deteriorates as the degree of shared access increases. On cluster size of up to 8 compute nodes, we demonstrated superior performance on TPC-C compared to Aurora MM and CockroachDB.

APPENDIX A. IMPACT OF INDIVIDUAL IMPROVEMENTS

We ran a number of experiments aimed at estimating the impact of three improvements in isolation: VS clocks, versioned reads, and hybrid row-page locks.

Impact of VS clocks

To estimate the impact of VS clocks on network bandwidth, we ran the fully partitioned write-only SysBench workload on a cluster with 8 masters. The total number of log records generated was 11.8 mln per second, corresponding to 570 MB/s of logs with VS clocks and 1,800 MB/s with vector clocks. The overall reduction in log data was 68%. Each log record is sent to three log servers and three page stores, and read by seven other masters, so transported over the network a total of 13 times. This adds up to a total network load of 13*1,800MB/s = 22.9GB/s with vector clocks and 13*570Mb/s = 7.2GB/s with VS clocks - a substantial saving of network bandwidth. It is worth noting that Taurus already performs simple compression of log records. Also, due to the batching of log records and lock messages, TCP packet size is maximized, resulting in TCP/IP header overhead being minimized as most segments are 1500 bytes.

The above experiment demonstrates the upper bound of the network traffic saved. In order to test a more realistic workload, we ran TPC-C on the 8 node cluster. Even then, the rate of log records generated by all masters was 72MB/s with VS clocks and 230MB/s with vector clocks. In summary, VS clocks significantly reduce network load, which helps reduce the investment in networking infrastructure required to support tenant workloads.

Impact of versioned reads

VS clocks enable instantaneous creation of global snapshots, thus enabling consistent versioned reads. To estimate the performance impact of versioned reads, we ran the SysBench workload with versioned reads and with locking reads on a 4-master cluster. Two masters ran a fully-partitioned SysBench write-only workload, and two masters ran a SysBench read-only workload reading the data modified by the first two masters. We ran the workloads once with locking reads and once with versioned reads and compared the results. Versioned reads with VS clocks resulted in 40% fewer page lock requests and 116% more queries per second.

Impact of hybrid locks

We also tried to estimate the reduction in network traffic due to our hybrid page-row locking protocol. We compared our approach with a system where row locks are explicitly granted by a GLM. The local lock manager is responsible for granting row locks to transactions executing on its master and for releasing locks back to the GLM. A similar approach is used in IBM Db2 pureScale [38].

There are multiple possible policies for releasing locks to the GLM. One option (Baseline 1), is to release locks to the GLM only upon request from another master. This approach works very well for fully-partitioned workloads, as after the initial warm-up period, no further lock requests are sent to the GLM. However, for a shared workload with lock migration between masters, each lock reclaim request involves up to four message exchanges.

An alternative approach (Baseline 2), is to release a row lock to the GLM as soon as the transaction holding the lock completes. This approach requires up to two messages for every lock request and one unlock message per transaction, as all row locks are released upon transaction completion.

Our hybrid page-row locks do not require any lock messages beyond page locks. To evaluate the impact of our locking approach, we ran three variants of the read-write SysBench workload (80% reads, 20% writes) on a 4-master cluster: fully partitioned, 50% shared, and 100% shared. We disabled versioned reads and measured how many lock-related messages would be sent with the two baseline policies and with our hybrid page-row locking algorithm. The results are presented in Fig.9. Hybrid page-row locking performs better than both baseline alternatives, reducing the number of messages by 25%-44%, with the exception of the fully partitioned workload where both Baseline 1 and hybrid locks produced no lock-related messages after the initial warm-up.

ACKNOWLEDGMENTS

This work would not be possible without people who greatly contributed to the Taurus MM: Jin Chen, Hongbin Lu, and Juncai Meng. We also want to express gratitude to Calvin Wong, David Bigagli, Gaghik Khachatrian, Sergey Vojtovich, Sergey Glushchenko, Alexey Kopytov, Zheshan Hou, Zixi Qu, Haoze Huang, and Gord Sissons for their expertise and contribution.

REFERENCES

[1] Michael Abebe et al. 2020. DynaMast: Adaptive dynamic mastering for replicated systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, Texas, USA, 1381–1392.

[2] P. S. Almeida, C. Baquero, and V. Fonte. 2008. Interval tree clocks. In *International Conference On Principles Of Distributed Systems*. Springer, Luxor, Egypt, 259–274.

[3] P. Antonopoulos et al. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047

[4] E. Boutin and S. Abraham. 2019. Amazon Aurora Multi-Master: Scaling out database write performance. Amazon ReInvent.

[5] M. Bravo et al. 2015. On the use of Clocks to Enforce Consistency in the Cloud. *IEEE Data Eng. Bull.* 38, 1 (2015), 18–31.

[6] S. Chandrasekaran and R. Bamford. 2003. Shared cache-the future of parallel databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE Computer Society, NY USA, 840–840.

[7] K Mani Chandy and L. Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[8] J. C. Corbett et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[9] C.J. Date and H. Darwen. 1997. *A Guide to the SQL Standard: A user's guide to the standard database language SQL*. Addison-Wesley, San Francisco, CA.

[10] G. DeCandia et al. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[11] A. Depoutovitch et al. 2020. Taurus Database: How to Be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1463–1478. https://doi.org/10.1145/3318464.3386129

[12] J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. 11th Australian Comput. Science Conf.* Australian National University. Department of Computer Science, Canberra, Australia, 56–66.

[13] Goetz Graefe. 2010. A Survey of B-Tree Locking Techniques. *ACM Trans. Database Syst.* 35, 3, Article 16 (jul 2010), 26 pages. https://doi.org/10.1145/1806907.1806908

[14] R. Harding et al. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (jan 2017), 553–564. https://doi.org/10.14778/3055540.3055548

[15] Dongxu Huang et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[16] Jesper Wisborg Krogh. 2021. *InnoDB Locks*. Apress, Berkeley, CA, 123–140. https://doi.org/10.1007/978-1-4842-6652-6_7

[17] A. D. Kshemkalyani and A. Misra. 2020. The bloom clock to characterize causality in distributed systems. In *International Conference on Network-Based Information Systems*. Springer, Victoria, Canada, 269–279.

[18] Ajay D Kshemkalyani, Min Shen, and Bhargav Voleti. 2020. Prime clock: Encoded vector clock to characterize causality in distributed systems. *J. Parallel and Distrib. Comput.* 140 (2020), 37–51.

[19] S. Kulkarni et al. 2014. Logical physical clocks. In *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, December 16-19, 2014. Proceedings 18*. Springer, Cortina d'Ampezzo, Italy, 17–32.

[20] Ten H Lai and Tao H Yang. 1987. On distributed snapshots. *Inform. Process. Lett.* 25, 3 (1987), 153–158.

[21] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[22] L. Lamport. July 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21 (July 1978), 558–564.

[23] Q. Lin et al. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1659–1674. https://doi.org/10.1145/2882903.2882923

[24] David B. Lomet. 1993. Key Range Locking Strategies for Improved Concurrency. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 655–664.

[25] F. Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, Netherlands, 215–226.

[26] Bronson N. and ohers. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[27] Oracle. 2020. Oracle Real Application Clusters 19c Technical Architecture. https://www.oracle.com/webfolder/technetwork/tutorials/architecture-diagrams/19/rac/pdf/rac-19c-architecture.pdf

[28] A. Pavlo et al. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[29] Percona. 2018. *TPCC-Like Workload for Sysbench 1.0*. Percona. Retrieved October 1, 2022 from https://www.percona.com/blog/2018/03/05/tpcc-like-workload-sysbench-1-0/

[30] Rees S. 2012. *DB2 pureScale: Best Practices for Performance and Monitoring*. Technical Report. IDUG DB2 Technical Conference, Denver, CO, USA.

[31] Michael Stonebraker. 1986. The case for shared nothing. *IEEE Database Eng. Bull.* 9, 1 (1986), 4–9.

[32] Sysbench. 2020. *Scriptable multi-threaded benchmark tool*. Sysbench. Retrieved October 1, 2022 from https://github.com/akopytov/sysbench

[33] R. Taft et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1493–1509.

[34] F. J. Torres-Rojas and M. Ahamad. 1999. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing* 12, 4 (1999), 179–195.

[35] TPC. 1992. TPC-C. https://www.tpc.org/tpcc/

[36] M. Tyulenev et al. 2019. Implementation of cluster-wide logical clock and causal consistency in mongodb. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam, Netherlands, 636–650.

[37] A. Verbitski et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[38] IBM white paper. 2009. Transparent application scaling with IBM DB2 pureScale.

[39] Zhenkun Yang et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.