DynaMast: Adaptive Dynamic Mastering for Replicated Systems

Michael Abebe, Brad Glasbergen, Khuzaima Daudjee
Cheriton School of Computer Science, University of Waterloo
{mtabebe, bjglasbe, kdaudjee}@uwaterloo.ca

***Abstract*—Single-master replicated database systems strive to be scalable by offloading reads to replica nodes. However, single-master systems suffer from the performance bottleneck of all updates executing at a single site. Multi-master replicated systems distribute updates among sites but incur costly coordination for multi-site transactions. We present DynaMast, a lazily replicated, multi-master database system that guarantees one-site transaction execution while effectively distributing both reads and updates among multiple sites. DynaMast benefits from these advantages by dynamically transferring the mastership of data, or remastering, among sites using a lightweight metadata-based protocol. DynaMast leverages remastering to adaptively place master copies to balance load and minimize future remastering. Using benchmark workloads, we demonstrate that DynaMast delivers superior performance over existing replicated database system architectures.**

I. Introduction

The continuous growth in data processing demands has led to a renewed focus on large-scale database architectures. A popular solution to scaling a database server is to employ the *single master* approach [1]–[6] in which one node (machine) holds the master (writeable) copy of data while the remaining nodes hold read-only replicas of the data. By placing the master copies of all data items at a single master site, all update transactions can execute and commit locally at that site. Similarly, read-only transactions can execute and commit locally at any replica (replicated site). Thus, the single master architecture guarantees single-site transaction execution for all transactions, eliminating the need for expensive multi-site (distributed) transaction coordination. Another advantage of the single master architecture is that it distributes read-only transaction load among replicas, enjoying good scalability for read-intensive workloads. However, as the update workload scales-up, the performance of the replicated system suffers as the single master site that executes all update transactions becomes a bottleneck [7]–[9].

To alleviate this single master site bottleneck, *multi-master* replicated systems distribute master copies of data items to multiple sites [10] so as to distribute the update load among these sites. Consequently, the distribution of master copies results in transactions updating data at multiple sites [11]–[13]. To ensure transactional consistency and atomicity for multi-site update transactions, multi-master systems must employ a distributed commit protocol, e.g., two-phase commit. Unfortunately, such commit protocols significantly degrade the performance and scalability of replicated multi-master systems due to multiple rounds of messages and blocking [12]–[15].

This examination of the two replicated architectures stimulate us to pose the following challenging question: can we design a replicated database system that preserves the benefits of the single-master and multi-master architectural approaches while addressing their shortcomings? Such an architecture should support scalability by distributing the update load over multiple sites while also avoiding expensive multi-site transaction coordination. That is, the new architecture should (i) allow single-site transaction execution for *all transactions* (read-only and update), and (ii) support system scalability by distributing load of *both update and read-only transactions*.

While the design of this new architecture brings significant benefits, it poses several research challenges. First, we must support *dynamic* single-site transaction execution, that is, we need to select one site at which to execute a transaction *without* constraining execution to this same site for all transactions. Thus, the replicated system should be flexible so as to support one-site execution at *any site* in the system, thereby offering opportunities to distribute load among sites. Second, all master copies of data that a transaction needs to update as well as read need to be located at the execution site. Deciding where to locate master copies of data while ensuring that transactions see, and install, a consistent view of the data adds to the set of challenging problems to solve.

In this paper, we address these challenges by designing and building a new replicated system called **DynaMast** that provides all of the above desirable properties while addressing the deficiencies of prior approaches. DynaMast guarantees one-site transaction executions by *dynamically transferring data mastership* among sites in the distributed system while maintaining transactional consistency. We call this technique *remastering*, which also distributes update and read load among sites in the system.

DynaMast decides the master location of each data item using comprehensive strategies that consider data item access patterns to balance load among sites and minimize future remastering. When remastering occurs, it is efficient as DynaMast uses a lightweight protocol that exploits the presence of replicas to transfer mastership using metadata-only operations.

These design decisions enable DynaMast to significantly reduce transaction latency compared to both the single-master and multi-master replicated architectures resulting in up to a $15\times$ improvement in throughput. Moreover, we show DynaMast's ability to flexibly and dynamically select physical transaction execution sites in the distributed system by learning workload data access patterns, thereby delivering superior performance in the face of changing workloads.

The contributions of this paper are four-fold:

1) We propose a novel replication protocol that efficiently supports dynamic mastership transfer to guarantee single-site transactions while maintaining well-understood and established transactional semantics. (Section III)
2) We present remastering strategies that learn workload data-access patterns and exploit them to remaster data adaptively. Our strategies transfer the mastership of data among sites to minimize future remastering, which in turn reduces transaction processing latency. (Section IV)
3) We develop DynaMast, an in-memory, replicated, multi-master database system that provides low latency transaction execution through the use of the dynamic mastering protocol and remastering strategies. (Section V)
4) We empirically compare DynaMast with single-master and multi-master architectures, demonstrating DynaMast's superiority on a range of workloads. (Section VI)

Additionally, this technical report contains detailed proofs, additional experiments, and further details on DynaMast.

II. Background

In this section, we discuss limitations of the single-master and multi-master replication approaches and illustrate the benefits of dynamic mastering.

A. Limitations of Single-Master & Multi-Master Architectures

Replicated single-master systems route all update transactions to a single site, which eliminates multi-site transactions but overloads that site. Figure 1a illustrates this problem by example; a client submits three update transactions that all execute on the master copies (indicated by bold, uppercase letters) at the single-master site (Site 1). Hence, the update workload overloads the single-master site as it cannot offload any update transaction execution to a replicated site (Site 2).

By contrast, the replicated multi-master architecture distributes updates among the sites to balance the load. In the example in Figure 1b, updates to data item a and c execute at Site 1, while updates to b execute at Site 2. However, transactions that update both a and b, such as T_1 are forced to execute at multiple sites (Site 1 and Site 2), requiring a distributed commit protocol to ensure atomicity. As discussed in the last section, such protocols are expensive as they incur overhead from blocking while waiting for a global decision and suffer from latencies due to multiple rounds of communication. We illustrate this using the popular two-phase commit (2PC) protocol [16] in Figure 1b (Steps 2-4). Observe that all transactions with distributed write sets, such as T_2 in Figure 1b, must execute as expensive distributed transactions. Only transactions with single-site write sets, such as T_3, are free to execute as local transactions in the multi-master replicated architecture.

B. Dynamic Mastering

Figure 1c shows how transactions T_1, T_2 and T_3 execute using our proposed dynamic mastering protocol, which, as in the single-master case, guarantees single-site execution while allowing the distribution of load as in the multi-master architecture. Like a multi-master architecture, observe that both sites contain master copies of data. Executing T_1 at one site requires changing the mastership of either data item a or b. Our site selector recognizes this requirement and, without loss of generality, decides to execute the transaction at Site 2, and therefore dynamically remasters a from Site 1 to Site 2. To do so, the site selector sends a `release` message for a to Site 1, which releases a's mastership after any pending operations on a complete (Step 1). Next, the site selector informs Site 2 that it is now the master of a by issuing the site a `grant` message for a (Step 2). In Step 3, T_1 executes unhindered at Site 2 by applying the operations and committing locally, requiring no distributed coordination. Through careful algorithmic and system design that enables remastering outside the boundaries of transactions and using metadata-only operations, we ensure that dynamic mastering is efficient (Section III-B).

Remastering is necessary only if a site does not master all of the data items that a transaction will update. For example, in Figure 1c, a subsequent transaction T_2 also updates a and b, and therefore executes without remastering, *amortizing* the first transaction's remastering costs. Unlike the single-master architecture (Figure 1a), dynamic mastering allows T_3 to execute at a different site, thereby *distributing the write load* through multiple one-site executions. Thus, it is important that the site selector *adaptively* decides where to remaster data to balance load and minimize future remastering — objectives our remastering strategy takes into account (Section IV).

1) System Model: To perform dynamic mastering efficiently, we consider a model in which data is fully replicated so that every site has a copy of every data item, allowing flexibility in mastership placement. A transaction provides write-set information, using reconnaissance queries [14], [17]–[19] if necessary, which allows mastering the write-set at a single-site. The dynamic mastering system guarantees snapshot isolation (SI) as is common in distributed replicated databases [20]–[22]; additionally, strong-session snapshot isolation (SSSI) is guaranteed on top of SI.

III. Dynamic Mastering Protocol

Having presented an overview of remastering and its benefits, we now detail our dynamic mastering protocol and its implementation.

A. Maintaining Consistent Replicas

The dynamic mastering protocol exploits lazily maintained replicas to transfer ownership of data items efficiently. Lazily-replicated systems execute update transactions on the master copies of data, and apply updates asynchronously to replicas as *refresh transactions*.

We now describe how the dynamic mastering protocol guarantees snapshot isolation (SI) that is popularly used by

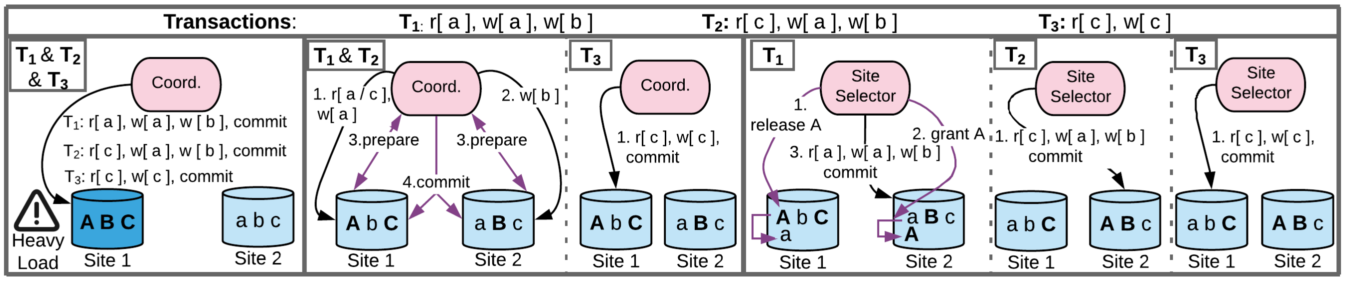

Transactions:
T1: r[a], w[a], w[b]
T2: r[c], w[a], w[b]
T3: r[c], w[c]
T1 & T2 & T3
Coord.
T1: r[a], w[a], w [b], commit
T2: r[c], w[a], w [b], commit
T3: r[c], w[c], commit
Heavy Load
A B C
a b c
Site 1
Site 2
T1 & T2
1. r[a / c], w[a]
2. w[b]
3.prepare
3.prepare
4.commit
A b C
a B c
T3
1. r[c], w[c], commit
T1
Site Selector
1. release A
2. grant A
3. r[a], w[a], w[b] commit
A b C a
a B c A
T2
1. r[c], w[a], w[b] commit
a b C
A B c
T3
1. r[c], w[c], commit
(a) Single-Master (b) Multi-Master (c) Dynamic Mastering

Fig. 1: An example illustrating the benefits of dynamic mastering over single-master and multi-master replicated architectures. Uppercase bolded letters represent master copies of data items, with read-only replicas distributed to the other sites. Single-master bottlenecks the master site as all transactions execute there. Multi-master requires a distributed commit protocol for both transactions T_1 and T_2 as the write set is distributed. In dynamic mastering, a is remastered before T_1 executes, allowing T_1 and T_2 to execute as single-site transactions at Site 2. The load is distributed by executing T_3 at Site 1.

replicated database systems [20]–[22], and later outline the provision of session guarantees on top of SI. An SI system must assign every transaction T a begin timestamp such that T sees the updates made by transactions with earlier commit timestamps. SI systems must also ensure that two transactions with overlapping lifespans cannot both successfully update the same data item [23]. Consequently, for every update transaction T, its corresponding refresh transaction $R(T)$ is applied to replicas in an order that ensures transactions observe a consistent snapshot of data. Ensuring that transactions respect SI without unnecessary blocking is challenging in a dynamically mastered system as unlike traditional single-master architectures, updates may originate from any site, and masters can change dynamically.

To apply refresh transactions in a consistent order, we track each site's state using *version vectors*. In a dynamic mastering system with m sites, each site maintains an m-dimensional vector of integers known as a *site version vector*, denoted $svv_i[\,]$ for the i^{th} site (site S_i), where $1 \le i \le m$. The j-th index of site S_i's version vector, $svv_i[j]$, indicates the number of refresh transactions that S_i has applied for transactions originating at site S_j. Therefore, whenever site S_i applies the updates of a refresh transaction originating at site S_j, S_i increments $svv_i[j]$. Similarly, $svv_i[i]$ denotes the number of locally committed update transactions at site S_i.

We assign update transactions an m-dimensional transaction version vector, $tvv[\,]$ that acts as a commit timestamp and ensure that updates are applied in an order consistent with this commit timestamp across sites. When an update transaction T begins executing at site S_i, it records S_i's site version vector $svv_i[\,]$ as T's begin timestamp ($tvv_{B(T)}[\,]$). During commit, T copies $tvv_{B(T)}[\,]$ to tvv_T, increments $svv_i[i]$ and copies that value to $tvv_T[i]$. Thus, $tvv_T[\,]$ (the commit timestamp) reflects T's position in S_i's sequence of committed update transactions, while $tvv_{B(T)}[\,]$ (the begin timestamp) represents the updates visible to T when it executed.

SI does not prevent *transaction inversion* [22], i.e., under SI, a client may not see its previous updates in subsequent transactions. SSSI eliminates this undesirable behaviour by adding session freshness rules [22] and is a flexible and popular solution [7], [24]–[26]. To enforce the client session-freshness guarantee, the system tracks each client's state using a client version vector $cvv_c[\,]$ and ensures that a client's transaction executes on data that is at least as fresh as the state last seen by the client. Specifically, transactions respect the following *freshness rules*: when a client c with an m-dimensional client session version vector $cvv_c[\,]$ accesses data from a site S_i with site version vector $svv_i[\,]$, c can execute when $svv_i[k] \ge cvv_c[k], \forall k \in (1, \ldots, m)$. After the client accesses the site, it updates its version vector to $svv_i[\,]$.

The transaction version vector and site version vector indicate when a site can apply a refresh transaction. Given a transaction T that commits at site S_i, a replica site S_j applies T's refresh transaction $R(T)$ only after S_j commits all transactions whose committed updates were read or updated by T. Formally, we say that T *depends* on T' if T reads or writes a data item updated by T'. That is, $R(T)$ cannot execute and commit at S_j until S_j commits all transactions that T depends on. The transaction version vector ($tvv_T[\,]$) encapsulates the transactions that T depends on, while the site version vector ($svv_j[\,]$) indicates the updates committed at S_j. Hence, site S_j blocks the application of $R(T)$ until the following *update application rule* holds (Equation 1):

$$\Big(\bigwedge_{k \neq i} svv_j[k] \ge tvv_T[k]\Big) \wedge \big(svv_j[i] = (tvv_T[i] - 1)\big) \quad (1)$$

As an example of the update application rule, consider the three-sited system in Figure 2. In the first two steps, transaction T_1 updates a data item and commits locally at site S_1, which increments the site version vector from $[0,0,0]$ to $[1,0,0]$. Next, T_1's refresh transactions, $R(T_1)$, begin applying the updates to sites S_2 and S_3. $R(T_1)$ commits at S_3, and sets $svv_3[\,]$ to $[1,0,0]$ (Step 4), but site S_2 has not yet committed $R(T_1)$ (Step 5). In Step 6, transaction T_2 begins after $R(T_1)$ and commits at site S_3, and therefore sets $svv_3[\,]$ to $[1,0,1]$, capturing that T_2 depends on T_1. The update application rule blocks S_2 from applying $R(T_2)$ until $R(T_1)$ commits (Step 7). Without blocking the application of T_2, it would be possible for T_2's updates to be visible at site S_2 before T_1's updates

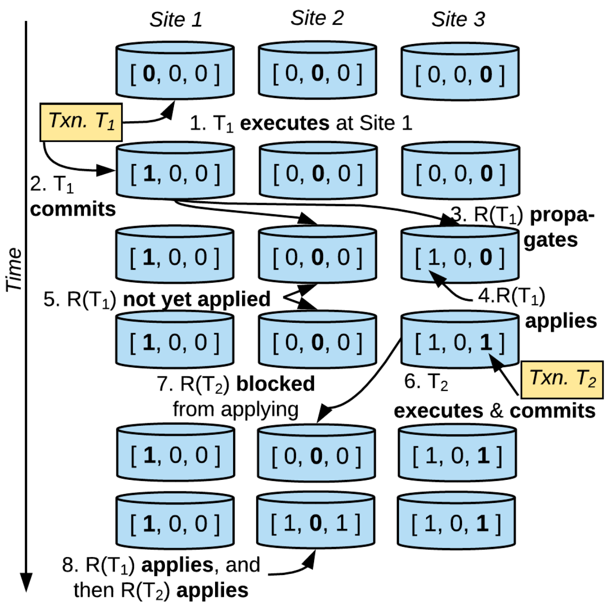

Site 1
Site 2
Site 3
[0, 0, 0]
Txn. T_1
1. T_1 executes at Site 1
[1, 0, 0]
2. T_1 commits
3. $R(T_1)$ propagates
4. $R(T_1)$ applies
5. $R(T_1)$ not yet applied
Time
[1, 0, 1]
Txn. T_2
6. T_2 executes & commits
7. $R(T_2)$ blocked from applying
8. $R(T_1)$ applies, and then $R(T_2)$ applies

Fig. 2: An example showing how commits and update propagation affect site version vectors.

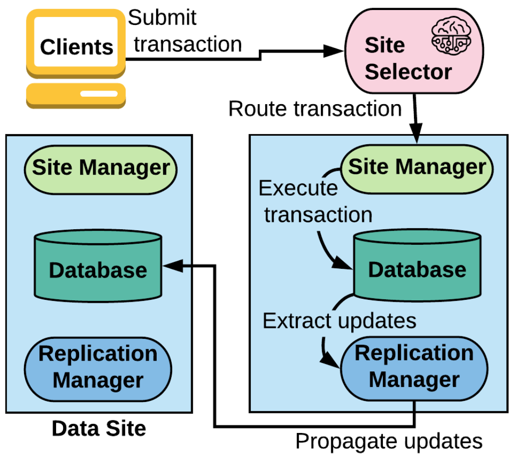


Fig. 3: Generic dynamic mastering architecture.

have been applied, despite the fact that T_2 depends on T_1. Finally, after site S_2 applies and commits $R(T_1)$, it applies $R(T_2)$, which results in $svv_2[\,]$ being set to $[1, 0, 1]$ and ensures a globally consistent order of update application.

B. Transaction Execution and Remastering

The dynamic mastering protocol is implemented with the architecture shown in Figure 3 consisting of a site selector and data sites composed of a site manager, database system, and replication manager. Clients submit transactions to the site selector, which remasters data items if necessary and routes transactions to an appropriate data site. The site managers at each data site interact with a database system to maintain version vectors, apply propagated updates as refresh transactions, and handle remastering requests. The database system processes transactions and sends transactional updates to a replication manager, which forwards them to the other sites for application as refresh transactions.

Clients and the system components interact with each other using remote procedure calls (RPCs). Clients issue transactions by sending a `begin_transaction` request containing the transaction's write set to the site selector, which decides on the execution site for the transaction using the routing and remastering strategies discussed in Section IV. If necessary, the site selector transfers the mastership of relevant data items to the execution site via remastering.

To perform remastering (Algorithm 1), the site selector issues parallel `release` RPCs to each of the site managers that hold master copies of data items to be remastered (Line 7). When a site manager receives a `release` message, it waits for any ongoing transactions writing the data to finish before releasing mastership of the items and responding to the site selector with the data site's version vector. Immediately after a `release` request completes, the site selector issues a `grant` RPC to the site that will execute the transaction (Line 8). This data site waits until updates to the data item from the releasing site have been applied to the point of the release. The data site then takes mastership of the granted items and returns a version vector indicating the site's version at the time it took ownership. After all necessary `grant` requests complete, the site selector computes the element-wise max of the version vectors that indicates the minimum version that the transaction must execute on the destination site (Line 9). Finally, the site selector notifies the client of the site that will execute its transaction and this minimum version vector. `release` and `grant` are executed as transactions, and consequently a data item waits to be remastered while being updated.

Parallel execution of `release` and `grant` operations greatly speed up remastering, reducing waiting time for clients. Clients begin executing as soon as their write set is remastered, benefiting from the remastering initiated by clients with common write sets. A client then submits transactions directly to the database system. Client `commit`/`abort` operations are submitted to the site manager. Note that since data is fully replicated, clients need not synchronize with each other unless they are waiting for a `release` or `grant` request. Further, read-only transactions may execute at any site in the system without synchronization across sites.

A key performance *advantage* is that coordination through remastering takes place outside the boundaries of a transaction, and therefore does not block running transactions. Once a transaction begins at a site, it executes without any coordination with any other sites, allowing it to commit or abort unilaterally. By contrast, in a multi-master architecture, changes made by a multi-site transaction are not visible to other transactions because the database is uncertain about the global outcome of the distributed transaction. Thus, distributed transactions block local transactions further increasing transaction latency [13]. To illustrate this benefit, consider a transaction T_4 that updates B concurrently with T_1 in our example from Figure 1. T_4's update to B can occur while A is being remastered, unlike during 2PC in multi-master that blocks T_4 while the global outcome of T_1 is unknown.

Thus, remastering operations (i) change only mastership

Algorithm 1 Remastering Protocol

Input: Transaction T's write set w
Output: The site S that will execute the update transaction T, and version vector out_vv indicating T's begin version

```
1:  $S$ = determine_best_site($w$) // execution site
2:  $data\_item\_locs$ = group_by_master_loc($w$)
3:  $out\_vv = \{0\} \times m$ //zero_vector
4:  // In parallel:
5:  for ($site\ S_d$, $data\_item\ d$) $\in data\_item\_locs$ do
6:      if $S_d \neq S$ then
7:          // $S_d$ currently masters $d$
8:          $rel\_vv$ = send_release($S_d, d$)
9:          $grant\_vv$ = send_grant($S$, $d$, $rel\_vv$)
10:         $out\_vv$ = elementwise_max($out\_vv, grant\_vv$)
11:     end if
12: end for
13: return ($out\_vv$, $S$)
```

location metadata, (ii) occur outside the boundaries of transactions, (iii) do not physically copy master data items, and (iv) allow one-site transaction execution once a write set is localized. These features make our dynamic mastering protocol *lightweight* and *efficient*, resulting in significant **performance advantages** over existing approaches (Section VI).

IV. Site Selector Strategies

In previous sections, we described the dynamic mastering architecture and the importance of adaptive remastering decisions. Our system, DynaMast, implements this dynamic mastering architecture and efficiently supports it using comprehensive transaction routing and remastering strategies. We will now describe these strategies in detail, deferring descriptions of their implementation to Section V.

Transaction routing and remastering decisions play a key role in system performance. Routing and remastering strategies that do not consider load balance, data freshness, and access patterns for data items can place excessive load on sites, increase latency waiting for updates, and cause a ping-pong effect by repeatedly remastering the same data between nodes. Thus, DynaMast uses comprehensive strategies and *adaptive* models to make transaction routing and remastering decisions.

A. Write Routing and Remastering

When the site selector receives a write transaction request from client c, it first determines whether the request requires remastering. If all items the transaction wishes to update are currently mastered at one of the sites then the site selector routes the transaction there for local execution. However, if these items' master copies are distributed across multiple sites, the site selector co-locates the items via remastering before transaction execution. DynaMast makes remastering decisions prudently: data is remastered only when necessary, employing strategies that choose a destination site that minimizes the amount of remastering in the future.

Our remastering strategies consider load balance, site freshness, and data access patterns. We use a linear model that captures and quantifies these factors as input features and outputs a score that represents an estimate of the expected benefits of remastering to a site. Concretely, DynaMast computes a score for each site, indicating the benefits of remastering the transaction's write set there and then remasters these data items to the site that obtained the highest score.

1) Balancing Load: Workloads frequently contain access skew, which if left unaddressed can result in resource underutilization and performance bottlenecks [27]. Consequently, our remastering strategy balances data item mastership allocation among the sites according to the write frequency of the items, which in turn balances the load.

When evaluating a candidate site S as a destination for remastering, we consider both the *current* write load balance and the *projected* load balance if we were to remaster to S the items that the transaction wishes to update. For a system mastership allocation X, we express the write balance as the *distance from perfect write load balancing* (where every one of the m sites processes the same volume of write requests):

$$f_{balance\_dist}(X) = \sqrt{\sum_{i=1}^{m} \left(\frac{1}{m} - freq(X_i) \right)^2}$$

where $freq(X_i) \in [0, 1]$ indicates the fraction of write requests that would be routed to site i under mastership allocation X. If each site receives the same fraction of writes ($\frac{1}{m}$), then $f_{balance\_dist}(X) = 0$; larger values indicate larger imbalances in write load.

We use $f_{balance\_dist}$ to consider the change in load balance if we were to remaster the items a transaction T wishes to update to a candidate site S. Let B be the current mastership allocation and $A(S)$ be the allocation resulting from remastering T's write set to S. The change in write load balance is computed as:

$$f_{\triangle balance}(S) = f_{balance\_dist}(B) - f_{balance\_dist}(A(S)) \quad (2)$$

A positive value for $f_{\triangle balance}(S)$ indicates that the load would be more balanced after remastering to site S; a negative value indicates that the load would be less balanced.

Although $f_{\triangle balance}(S)$ gives an indication of a remastering operation's improvement (or worsening) in terms of write balance, it does not consider how balanced the system *currently is*. If the current system is balanced, then unbalancing it slightly, in exchange for less future remastering, may yield better performance. However, for a system that is already quite imbalanced, re-balancing it is important. We incorporate this information into a scaling factor $f_{balance\_rate}(S)$ that reinforces the importance of balance in routing decisions:

$$f_{balance\_rate}(S) = \max\left(f_{balance\_dist}(B), f_{balance\_dist}(A(S))\right) \quad (3)$$

We combine the change in write load balance, $f_{\triangle balance}$, with the balance rate, $f_{balance\_rate}$, to yield an overall balance factor. This factor considers both the magnitude of change in write load balance and the importance of correcting it:

$$f_{balance}(S) = f_{\triangle balance}(S) \cdot \exp\left(f_{balance\_rate}(S)\right) \quad (4)$$

2) Estimating Remastering Time: After a candidate site S is chosen as the remastering destination for a transaction, the `grant` request blocks until S applies the refresh transactions for all of the remastered items. Additionally, the transaction may block at S to satisfy session-freshness requirements. Thus, if S lags in applying updates, the time to remaster data and therefore execute the transaction increases.

Our strategies estimate the number of updates that a candidate site S needs to apply before a transaction can execute by computing the dimension-wise maximum of version vectors for each site S_i from which data will be remastered and client c's version vector ($cvv_c[\,]$). We subtract this vector from the current version vector of S and perform a dimension-wise sum to count the number of necessary updates, expressed as:

$$f_{refresh\_delay}(S) = \left\| \max\left(cvv_c[\,], \max_i \left(svv_i[\,] \right) \right) - svv_S[\,] \right\|_1 \tag{5}$$

3) Co-locating Correlated Data: Data items are often correlated; a particular item may be frequently accessed with other items according to relationships in the data [11], [28], [29]. Thus, we consider the effects of remastering data on current and subsequent transactions. Our strategies remaster data items that are frequently written together to one site, which optimizes for current and subsequent transactions with one remastering operation. The goal of co-accessed data items sharing the same master site is similar to the data partitioning problem [11], [29], solutions to which typically model data access as a graph and perform computationally expensive graph partitioning to decide on master placement. Instead, our site selector strategy uses a heuristic that promotes mastership co-location based on data access correlations.

We consider two types of data access correlations: data items frequently written together within a transaction (intra-transaction access correlations) and items indicative of future transactions' write sets (inter-transaction access correlations). In the former case, we wish to keep data items frequently accessed together mastered at a single site to avoid remastering for subsequent transactions (ping-pong effect). In the latter case, we anticipate future transactions' write sets and preemptively remaster these items to a single site. Doing so avoids waiting on refresh transactions to meet session requirements when a client sends transactions to different sites. Considering both of these cases enables DynaMast to rapidly co-locate master copies of items that clients commonly access together.

To decide on master co-location, DynaMast exploits information about intra-transactional data item accesses. For a given data item d_1, DynaMast tracks the probability that a client will access d_1 with another data item d_2 in a transaction as $P(d_2|d_1)$. For a transaction with write set w that necessitates remastering and a candidate remastering site S, we compute the *intra-transaction localization factor* as:

$$f_{intra\_txn}(S) = \sum_{d_1 \in w} \sum_{d_2} P(d_2|d_1) \times \texttt{single\_sited}(S, \{d_1, d_2\}) \tag{6}$$

where `single_sited` returns 1 if remastering the write set to S would place the master copies of both data items at the same site, -1 if it would split the master copies of d_1 and d_2 apart, and 0 otherwise (no change in co-location). Thus, `single_sited` encourages remastering data items to sites such that future transactions would not require remastering. We normalize the benefit that remastering these items may have by the likelihood of data item co-access. Consequently, a positive f_{intra_txn} score for site S indicates that remastering the transaction's write set to S will improve data item placements overall and reduce future remastering.

DynaMast also tracks inter-transactional access correlations, which occur when a client submits a transaction that accesses item d_2 within a short time interval of accessing a data item d_1. We configure this interval, Δt, based on inter-transactional arrival times and denote the probability of this inter-transactional access as $P(d_2|d_1; T \leq \Delta t)$. For a transaction with write set w and candidate remastering site S, we compute the *inter-transaction localization factor* similarly to Equation 7, but normalize with inter-transactional likelihood:

$$f_{inter\_txn}(S) = \sum_{d_1 \in w} \sum_{d_2} P(d_2|d_1; T \leq \Delta t) \times \texttt{single\_sited}(S, \{d_1, d_2\}) \tag{7}$$

$f_{inter\_txn}(S)$ quantifies the effects of remastering the current transaction's write set to candidate site S with respect to accesses to data items in the future.

4) Putting It All Together: Each of the previously described factors come together to form a comprehensive model that determines the benefits of remastering at a candidate site. When combined, features complement each other and enable the site selector to find good master copy placements.

$$f_{benefit}(s) = w_{balance} \cdot f_{balance}(s) + w_{delay} \cdot f_{refresh\_delay}(s) + w_{intra\_txn} \cdot f_{intra\_txn}(s) + w_{inter\_txn} \cdot f_{inter\_txn}(s) \tag{8}$$

We combine the scores for site S in Equations 4 through 7 using a weighted linear model (Equation 8), and remaster data to the site that obtains the highest score.

B. Read Routing

Site load and timely update propagation affect read-only transaction performance as clients wait for sites to apply updates present in their session. Thus, DynaMast routes read-only transactions to sites that satisfy the client's session-based freshness guarantee. DynaMast randomly chooses a site that satisfies this guarantee, which both minimizes blocking and spreads load among sites.

V. THE DYNAMAST SYSTEM

Our replicated database system, DynaMast, implements the dynamic mastering architecture (Section III) and consists of a site selector and data sites containing a site manager, database system and replication manager. The site selector uses the remastering strategies from Section IV. System components communicate via the Apache Thrift RPC library [30].

A. Data Sites

A data site is responsible for executing client transactions. DynaMast integrates the site manager, database system and replication manager into a single component, which improves system performance by avoiding concurrency control redundancy between the site manager and the database system while minimizing logging/replication overheads. The replication manager propagates updates among data sites through writes to a durable *log* (Section V-A2) that also serves as a persistent redo log (Section V-C). For fault tolerance and to scale update propagation independently of the data sites, we use Apache Kafka [31] to store our logs and transmit updates.

1) Data Storage and Concurrency Control: The data site stores records belonging to each relation in a row-oriented in-memory table using the primary key of each record as an index. Our system supports reading from snapshots of data using multi-version concurrency control (MVCC), similar to Microsoft's Hekaton engine [32], [33], to exploit SSSI [22]. The database stores multiple versions (by default four, as determined empirically) of every record, which we call *versioned records*, that are created when a transaction updates a record. Transactions read the versioned record that corresponds to a specific snapshot so that concurrent writes do not block reads [33]. To avoid transactional aborts on write-write conflicts, DynaMast uses locks to mutually exclude writes to records, which is simple and lightweight (Section VI-B7).

2) Update Propagation: Recall from the update application rule (Section III-A) that when an update transaction T commits at site S_i, the site version vector index $svv_i[i]$ is atomically incremented to determine commit order and the transaction is assigned a commit timestamp (transaction version vector) $tvv_T[\,]$. Site S_i's replication manager serializes $tvv_T[\,]$ and T's updates and writes them to S_i's log. Each replication manager subscribes to updates from logs at other sites. When another site S_j receives T's propagated update(s) from S_i's log, S_j's replication manager deserializes the update, follows the update application rules from Section III-A, and applies T's updates as a refresh transaction by creating new versioned records. Finally, the replication manager makes the updates visible by setting $svv_j[i]$ to $tvv_T[i]$.

B. Site Selector

The site selector is responsible for routing transactions to sites for execution and deciding if, where and when to remaster data items using the strategies detailed in Section IV.

To make a remastering decision, the site selector must know the site that contains the master copy of the data item. To reduce the overhead of this metadata, the site selector supports grouping of data items into partitions[1], tracking master location on a per partition basis, and remastering data items in partition groups. For each partition group, DynaMast stores partition information that contains the current master location and a readers-writer lock.

To route a transaction, the site selector looks up the master location of each data item in the transaction's write set in a concurrent hash-table containing partition information. The site selector acquires each accessed partition's lock in shared read mode. If one site masters all partitions, then the site selector routes the transaction there and unlocks the partition information. Otherwise, the site selector makes a remastering decision and dynamically remasters the corresponding partitions to a single site. To do so, the site selector upgrades each partition information lock to exclusive write mode, which prevents concurrent remastering of a partition. Then, the site selector makes a remastering decision using vectorized operations that consider each site as a destination for remastering in parallel. Once the site selector chooses a destination site for the transaction, it remasters the necessary partitions using parallel `release` and `grant` operations, updates the master location in the partition and downgrades its lock to read mode. When the site masters the necessary partitions, the transaction begins executing, and the site selector releases all locks.

The site selector builds and maintains statistics such as data item access frequency and data item co-access likelihood for its strategies (Section IV) to effectively remaster data. Thus, the partition information also contains a counter that indicates the number of accesses to the partition and counts to track intra- and inter-transactions co-access frequencies. The site selector captures these statistics by adaptively sampling [34] transaction write sets and recording sampled transactions, and each transaction executed within a time window Δt (Equation 7) of it — submitted by the same client — in a transaction history queue. From these sampled write sets, the site selector determines partition level access and co-access frequencies, which it uses to make its remastering decisions. Finally, DynaMast expires samples from the transaction history queue by decrementing any associated access counts to adapt to changing workloads.

C. Fault Tolerance and Recovery

As is common in in-memory database systems [10], [19], DynaMast uses redo logging. On commit, DynaMast writes updates to the persistent Kafka log as a redo log to provide fault tolerance, and to propagate updates. Any data site recovers independently by initializing state from an existing replica and replaying redo logs from the positions indicated by the site version vector. DynaMast also logs `grant` and `release` operations to the redo log. Thus, if any site manager or site selector fails, on recovery it reconstructs the data item mastership state from the sequence of `release` and `grant` operations in the redo logs.

VI. Performance Evaluation

We evaluate the performance of DynaMast to answer the following questions:

- Can DynaMast efficiently remaster data and perform well on write-intensive workloads?
- Does DynaMast maintain its performance advantages when write transactions become more complex?
- Do DynaMast's remastering strategies ameliorate the effects of access skew?
- Does DynaMast adapt to changing workloads?

[1]By default, the site selector groups sequential data items into equally sized partitions [27] though clients can supply their own grouping.

- How frequent is remastering, and what are its overheads?
- How sensitive is DynaMast to remastering strategy hyperparameter values?

A. *Experimental Setup*

1) Evaluated Systems: To conduct an apples-to-apples comparison, we implement all alternative designs within the DynaMast framework. Hence, all systems share the same site manager, storage system, multi-version concurrency control scheme, and isolation level (strong-session snapshot isolation). This allows us to directly measure and attribute DynaMast's performance to the effectiveness of our dynamic mastering protocol and site selector strategies.

DynaMast: we implemented the dynamic mastering protocol and site selector strategies as the DynaMast system described in Section V. In each experiment, DynaMast has no fixed initial data placement as we rely on, and evaluate, its remastering strategies to distribute master copies among the sites.

Partition-Store: is the partitioned multi-master database system that we implemented in DynaMast. Partition-store uses table-specific partitioning (e.g. range, hash) to assign partitions to data sites, but does not replicate data except for static read-only tables. By using the offline tool Schism [11], we favoured partition-store to have superior partitioning for the OLTP workloads that we benchmark against. Partition-store uses the popular 2PC protocol to coordinate distributed transactions.

Multi-master: we implemented a replicated multi-master database system by augmenting partition-store to lazily maintain replicas. Thus, the multi-master system allows read-only transactions to execute at any site. As each data item has one master copy, updates to a data item occur only on the data item's master copy.

Single-Master: we leveraged DynaMast's adaptibility to design a single-master system in which all write transactions execute at a single (master) site while lazily maintaining read-only replicas at other sites. Single-master is superior to using a centralized system because the single-master system routes read-only transactions to (read-only) replicas, thereby reducing the load on the master.

LEAP: like DynaMast, guarantees single-site transaction execution but bases its architecture on a partitioned multi-master database without replication [14]. To guarantee single-site execution, LEAP localizes data in a transaction's read and write sets to the site where the transaction executes. To perform this data localization, LEAP does data shipping, copying data from the old master to the new master. We implement LEAP by modifying our partition-store implementation.

2) Benchmark Workloads: In our experiments, each data site executes on a 12-core machine with 32 GB RAM. We also deploy a site selector machine and two machines that run Apache Kafka to ensure that there are enough resources available to provide timely and efficient update propagation. A 10Gbit/s network connects machines. All results are averages of at least five, 5-minute OLTPBench [35] runs with 95% confidence intervals shown as bars around the means.

Given the ubiquity of multi-data item transactions [35], [36] and workload access patterns [14], [18] present in a broad class of OLTP workloads, we incorporated these realistic characteristics into the **YCSB** workload. We use YCSB's scan transaction that reads from 200 to 1000 sequentially ordered keys, and enhance the read-modify-write (RMW) transaction to update three keys. These modifications induce access correlations and multi-partition transactions, resulting in multi-site (distributed) transactions for multi-master and partition-store, remastering for DynaMast, and data-shipping for LEAP. Each experiment uses four data sites containing an initial database size of 5 GB that grows to 30 GB of data by the end of the run, thereby taking up most of the available memory.

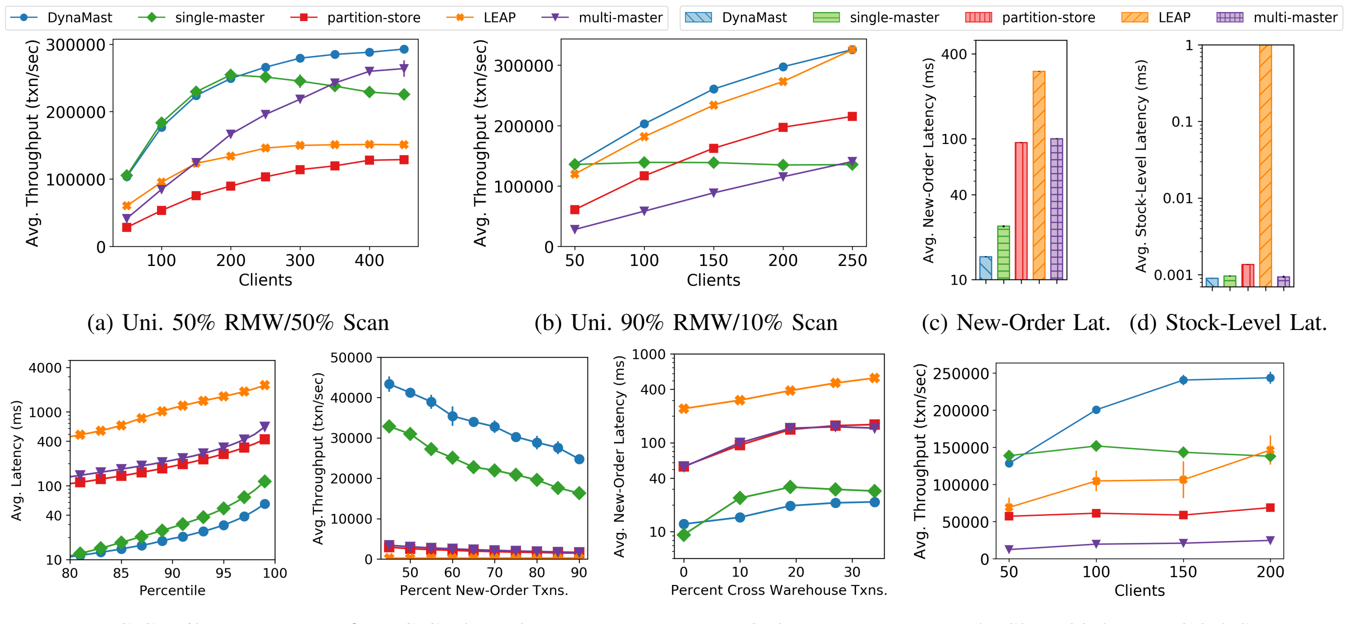
Our **TPC-C** workload evaluation contains three transaction types: *New-Order*, *Payment* and *Stock-Level* that represent two update intensive transactions and a read-only transaction, respectively, and make up the bulk of both the workload and distributed transactions. By default, we use eight data sites, 350 concurrent clients and a 45% New-Order, 45% Payment, 10% Stock-Level mix that matches the default update and read-only transaction mix in the TPC-C benchmark. Our TPC-C database has 10 warehouses and 100,000 items that grows to more than 20 GB of in-memory data per site by the end of an experiment run. Having more than this number of warehouses outstrips the physical memory of a data site machine.

B. *Results*

We compared DynaMast against partition-store, multi-master, single-master, and LEAP to answer the evaluation questions we posed at the beginning of Section VI.

1) Write-Intensive Workloads: Schism [11] reports that the partitioning strategy that minimizes the number of distributed transactions is range-partitioning. Thus, we assigned partition-store and multi-master a range-based partitioning scheme. DynaMast does not have a fixed partition placement and must learn access patterns to place partitions accordingly. As shown in Figure 4a, DynaMast outperforms the other systems, improving transaction throughput by $2.3\times$ over partition-store and $1.3\times$ over single-master. Partition-store performs poorly compared to the other systems due to additional round-trips during transaction processing. While LEAP's transaction localization improves performance over partition-store by 20%, DynaMast delivers double the throughput of LEAP. DynaMast executes the scan operations at replicas without the need for remastering while LEAP incurs data transfer costs to localize both read-only and update transactions.

Single-master offloads scan transactions to replicas. However, as the number of clients increases in Figure 4a, the single-master site bottlenecks as it becomes saturated with update transactions. Like single-master, multi-master's replication allows scans to run at any site, which improves performance compared to partition-store. Multi-master avoids the single-site bottleneck as it distributes writes. However, its multi-site write transactions incur synchronization costs that DynaMast eliminates. Consequently, DynaMast's remastering strategies



(a) Uni. 50% RMW/50% Scan (b) Uni. 90% RMW/10% Scan (c) New-Order Lat. (d) Stock-Level Lat.

(e) TPC-C Tail Lat. (f) TPC-C Throughput (g) New-Order Lat. (h) Skew 90% RMW/10% Scan

Fig. 4: Experiment results for DynaMast, partition-store, multi-master, single-master and LEAP using YCSB and TPC-C.

avoid the pitfalls of both single and multi-master; they ensure master copies of data are distributed evenly across the sites without executing distributed transactions, resulting in better resource utilization and thus superior performance.

Next, we increased the proportion of RMW transactions to 90% (Figure 4b), which increases the number of transactions that require remastering and increases contention. DynaMast continues to outperform the other systems by delivering almost $2.5\times$ more throughput. Multi-master has lower throughput compared to partition-store because there are fewer scans in the workload to leverage replicas. Increasing the proportion of update transactions in the workload saturates the single-master site rapidly. DynaMast and partition-store experience an improvement in transaction throughput because scan transactions access more keys and take longer to execute than the uniform workload's RMW transactions that have reduced resource contention. DynaMast is effective in executing read-only transactions at replicas and remastering write transactions. LEAP's performance is lower than DynaMast's as the workload requires LEAP to localize the read-only transactions that DynaMast executes at replicas without any remastering.

2) Complex Write Transactions: We studied the effect of a workload with more complex write transactions by using TPC-C. The New-Order transaction writes dozens of keys as part of its execution, increasing the challenge of efficiently and effectively placing data via remastering. TPC-C is not fully-partitionable due to cross-warehouse New-Order and Payment transactions but Schism confirms that the well-known partitioning by warehouse strategy minimizes the number of distributed transactions. Thus, we partition partition-store and multi-master by warehouse but force DynaMast to learn partition placements.

We first study the differences in New-Order transaction latency among DynaMast and its comparators (Figure 4c). On average, DynaMast reduces the time taken to complete the New-Order transaction by a hefty 40% when compared to single-master. This large reduction in latency comes from DynaMast's ability to process New-Order transactions at all data sites, thereby spreading the load across the replicated system unlike that of single master. As shown in Figure 4e, the largest difference in transaction latency between DynaMast and single master exists in 10% of transactions that suffer the most from load effects. Specifically, DynaMast reduces the 90^{th} and 99^{th} percentile tail latency by 30% and 50% respectively compared to single-master.

DynaMast reduces average New-Order latency by 85% when compared to partition-store and multi-master, both of which perform similarly. DynaMast achieves this reduction by running the New-Order transaction at a single site and remastering to avoid the cost of multi-site transaction execution, which partition-store and multi-master must incur for every cross-warehouse transaction. Consequently, Figure 4e shows that both partition-store and multi-master have significantly higher $(10\times)$ 90^{th} percentile latencies compared to DynaMast.

DynaMast reduces New-Order latency by 96% over LEAP, which has no routing strategies and thus continually transfers data between sites to avoid distributed synchronization, unlike DynaMast's adaptive strategies that limit remastering. LEAP's localization efforts result in high transfer overheads and contention, manifesting in a 99^{th} percentile latency that is $40\times$ higher than DynaMast's (Figure 4e).

DynaMast has low latency for the Stock-Level transaction (Figure 4d) because efficient update propagation rarely causes transactions to wait for updates, and multi-version concurrency control means that updates do not block read-only transactions. As single-master and multi-master also benefit from these optimizations, they have similar latency to DynaMast. Partition-store's average latency is higher because the Stock-Level transaction can depend on stock from multiple warehouses, necessitating a multi-site transaction. Although multi-site read-only transactions do not require distributed coordination, they are subject to straggler effects, increasing the probability of incurring higher latency as they must wait for all requests to complete. Thus, the slowest site's response time determines their performance. By contrast, LEAP, which lacks replicas, has orders of magnitude higher Stock-Level latency than DynaMast as it must localize read-only transactions.

Figure 4f shows how throughput varies with the percentage of New-Order transactions in the workload. When New-Order transactions dominate the workload, DynaMast delivers more than $15\times$ the throughput of partition-store and multi-master, which suffer from multiple round trips and high tail latencies. Similarly, DynaMast delivers $20\times$ the throughput of LEAP that lacks adaptive master transfer strategies and consequently continually moves data to avoid distributed coordination. DynaMast's significantly lower New-Order transaction latency results in throughput $1.64\times$ that of single-master.

3) Decreasing Transaction Access Locality: We study the effect of increasing the ratio of cross-warehouse New-Order transactions on New-Order average transaction latency (Figure 4g). DynaMast and LEAP never execute multi-site transactions so as cross-warehouse transactions increase, remastering increases in DynaMast and more data shipping occurs in LEAP. While the best partitioning remains warehouse-based, cross-warehouse transactions induce more distributed transactions in partition-store and multi-master.

DynaMast reduces New-Order transaction latency by an average of 87% over partition-store and multi-master when one-third of New-Order transactions cross warehouses. Partition-store and multi-master's New-Order latency increases almost $3\times$ over having no cross warehouse transactions to when one-third of the transactions cross warehouses, which results in an increase of only $1.75\times$ in DynaMast. LEAP increases the New-Order latency by more than $2.2\times$ as more distributed transactions necessitate more data transfers while DynaMast brings its design advantages that include routing strategies and remastering to bear, delivering better performance than LEAP.

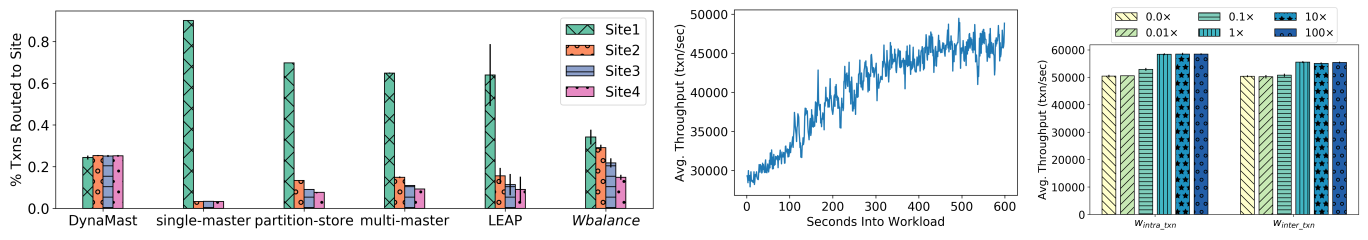
Partition-store and multi-master's latency increase because cross warehouse transactions also slow down single warehouse transactions [13]. As the number of cross-warehouse transactions increases, DynaMast recognizes that being a more dominant single-master system can bring performance benefits, and therefore reacts by mastering more data items at one site. However, DynaMast knowingly avoids routing all New-Order transactions to a single site to avoid placing excessive load on it. These techniques allow DynaMast to significantly reduce New-Order latency by 25% over single-master.

4) Skewed Workloads: We evaluated DynaMast's ability to balance load in the presence of skew via remastering with a YCSB-based Zipfian 90%/10% RMW/scan workload (Figure 4h). DynaMast significantly outperforms its comparators, improving throughput over multi-master by $10\times$, partition-store by $4\times$, single-master by $1.8\times$, and LEAP by $1.6\times$. Partition-store's performance suffers as it cannot distribute heavily-accessed partitions to multiple sites (Figure 5a), resulting in increased transaction latency due to resource contention. Multi-master suffers the same fate while additionally having to propagate updates, which further degrades performance. LEAP shows better throughput than partition-store as it executes transactions at one site but suffers from co-location of hot (skewed) partitions. Resource contention degrades performance for single-master as all update transactions must execute at the single master site. DynaMast mitigates the performance issues of its competitors by spreading updates to master data partitions over all sites in the replicated system evenly, resulting in balanced load and superior performance.

5) Learning Changing Workloads: Access patterns in workloads often change [27] resulting in degraded performance when data mastership/allocation is fixed. A key feature of DynaMast is its ability to adapt to these workload changes by localizing transactions. To demonstrate this adaptive capability, we conducted a YCSB experiment with mastership allocated manually using range partitioning but with the workload utilizing randomized partition access. Hence, DynaMast must learn new correlations and remaster data to maintain performance. We deployed 100 clients running 100% RMW transactions accessing data with skew. This challenges DynaMast with *high contention* and the remastering of *nearly every* data item. As Figure 5b shows, DynaMast rises to this challenge, continuously improving performance over the measurement interval resulting in a throughput increase of $1.6\times$ from when the workload change was initiated. This improvement showcases DynaMast's ability to learn new relationships between data partitions and its strategies, leveraging this information effectively to localize transactions via remastering.

6) Remastering Analysis: Recall that DynaMast makes remastering decisions by employing a linear model (Equation 8) that contains four hyperparameters ($w_{balance}$, w_{delay}, w_{intra_txn}, w_{inter_txn}). To determine the effects of these parameters on the site selector's master placement decisions and subsequently on performance, we performed sensitivity experiments using a skewed YCSB workload. We varied each parameter from its default value (normalized to 1) by scaling one or two orders of magnitude up, and down. We also set each parameter, in turn, to 0 to determine the effect of removing the feature associated with that parameter from the site selector's strategy. When every parameter is non-zero, throughput remains at 8% of the maximal throughput, demonstrating DynaMast's robustness to variation in parameter values.

When $w_{balance}$ is 0, throughput drops by nearly 40% as DynaMast increasingly masters data at one master site since no other feature encourages load balance. Figure 5a shows the effects on transaction routing when $w_{balance}$ is scaled to 0.01 of its default: 34% of the requests go to the most frequently accessed site while 13% of requests go to the least frequently accessed site, compared to even routing (25%) by default.



(a) Skew Transaction Routing (b) Adaptivity Over Time (c) Co-Access Sensitivity

Fig. 5: Transaction routing for a skewed workload, adaptivity results, and a sensitivity analysis of site selector strategies.

The w_{intra_txn} and w_{inter_txn} hyperparameters complement each other; when one is 0, the other promotes co-location of co-accessed data items. To further examine these effects, we induce workload change, as in Section VI-B5, so that learning and understanding data item access patterns is of utmost importance. Figure 5c shows throughput increasing as w_{intra_txn} increases from 0 to the relative value of 1 used in experiments. This 16% improvement in throughput demonstrates that the site selector captures the intra-transactional co-access patterns, using them to co-locate the master copies of co-accessed data items. We observe a similar trend of throughput increasing by 10% when we vary the inter-transactional co-access parameter (w_{inter_txn}).

7) Performance Breakdown: We designed DynaMast to be a system that delivers significant performance benefits with low overhead. Measurements of latencies for routing decisions, wait time for pending updates and transaction commit time are low. DynaMast's comprehensive strategies that learn access correlations are effective at minimizing remastering as less than 3% of transactions require remastering. Given these small latencies overall, network latencies proportionally account for 40% of latency, while transaction logic accounts for 40%.

8) Additional Results: Appendices D, F, E and G contain more experimental results. Using the SmallBank workload, we examine the effect of short transactions on DynaMast and show tail latency reductions of 4 to 40×. DynaMast exhibits *excellent* **scalability** with near-linear scaling of throughput that increases by more than 3× as the number of data sites is scaled from 4 to 16 nodes. Moreover, as database sizes increase, DynaMast continues to maintain its throughput.

VII. Related Work

DynaMast is the first *transactional* system that *both* adaptively places data by considering data access patterns and guarantees single site transaction execution via remastering.

Shared-data architectures decouple transaction processing from data storage [18], [37], [38]. These architectures rely on 2PL-like protocols over the network to guarantee atomicity for transactions or perform over the network data transfers to execute operations on the transaction processing node [14]. By contrast, DynaMast maintains the replicated multi-master architecture while supporting one-site transaction execution, foregoing communication with other nodes during execution.

TAPIR [39] and Janus [40] address the overheads of distributed transaction processing by coupling the transaction consistency protocol with the data replication protocol that is necessary for fault tolerance. However, these systems — unlike DynaMast — do not guarantee single site transaction execution, statically assign master copies of data to nodes and do not support mastership changes.

Systems reduce transaction latency by replicating data locally into caches [41]–[43]. However, to ensure consistency these systems must either update or invalidate the caches, which require distributed coordination and increase update transaction latency. By contrast, DynaMast lazily updates replicas, executes transactions locally, and uses comprehensive cost-based remastering strategies to learn application workloads that balance the load and minimize future remastering in the replicated system.

Coordination avoidance [44] exploits application invariants [45] and commutative data types [20], [46] to avoid distributed transactions, by asynchronous merging of diverging updates. However, not all workloads support such invariants and thus require distributed transactions. Speculative execution [13], [47], [48] make globally uncommitted changes visible to other transactions early. Regardless of the operation, DynaMast guarantees one-site transaction execution to all transactions.

Deterministic databases [17], [49] eliminate distributed communication by grouping all transactions submitted to the system within an epoch and then executing them as a batch, which increases transaction latency. STAR [19] replicates data into both a single-master and partitioned multi-master format, then groups transactions into batches and divides them into either the single-master or partitioned multi-master execution batch. Unlike such systems, DynaMast executes transactions as they arrive, and remasters on-the-fly only when necessary.

Advances in low-latency remote memory access [50]–[52] and programmable network switches [53] allow systems to improve throughput in distributed database systems. Such hardware allows efficient remote data access, but still require expensive distributed protocols to coordinate and acquire locks (e.g., using distributed 2PL) through RDMA operations. By contrast, DynaMast improves transaction processing by guaranteeing single-site transaction execution without relying on specialized hardware technologies.

Repartitioning systems [11], [27], [29], [54]–[59] modify data placement to reduce the number of distributed transactions. However, distributed transactions remain unless the workload is perfectly partitionable, whereas DynaMast eschews distributed transactions entirely using lightweight metadata-only remastering operations.

VIII. Conclusion

We presented DynaMast, a multi-master replicated database system that guarantees one-site transaction execution through dynamic mastering. As shown experimentally, DynaMast's novel remastering protocol and adaptive strategies are lightweight, ensure balanced load among sites, and minimize remastering as part of future transactions. Consequently, DynaMast eschews multi-master's expensive distributed coordination and avoids the single-master site bottleneck. These design strengths allow DynaMast to improve performance by up to $15\times$ over prior replicated system designs.

References

[1] S. Wu and B. Kemme, "Postgres-r (si): Combining replica control with concurrency control based on snapshot isolation," in *ICDE*, 2005.

[2] A. Verbitski *et al.*, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," 2017.

[3] "Mysql: Primary-secondary replication," https://dev.mysql.com/doc/refman/8.0/en/group-replication-primary-secondary-replication.html, 2019, accessed: 2019-02-01.

[4] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," 2011.

[5] N. Bronson *et al.*, "TAO: Facebooks distributed data store for the social graph," 2013.

[6] "Huawei relational database service," https://support.huaweicloud.com/en-us/productdesc-rds/rds-productdesc.pdf, 2019, accessed: 2019-06-01.

[7] J. Goldstein *et al.*, "Mtcache: Mid-tier database caching for sql server," *ICDE*, 2004.

[8] Q. Luo *et al.*, "Middle-tier database caching for e-business," in *SIGMOD*, 2002.

[9] A. Pavlo *et al.*, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," *SIGMOD*, 2012.

[10] M. Stonebraker *et al.*, "The end of an architectural era: (it's time for a complete rewrite)," *VLDB*, 2007.

[11] C. Curino *et al.*, "Schism: a Workload-Driven Approach to Database Replication and Partitioning," *PVLDB*, 2010.

[12] R. Harding *et al.*, "An evaluation of distributed concurrency control," *PVLDB*, 2017.

[13] E. P. Jones *et al.*, "Low overhead concurrency control for partitioned main memory databases," 2010.

[14] Q. Lin *et al.*, "Towards a Non-2PC Transaction Management in Distributed Database Systems," *SIGMOD*, 2016.

[15] P. Chairunnanda *et al.*, "Confluxdb: Multi-master replication for partitioned snapshot isolation databases," *PVLDB*, 2014.

[16] A. Silberschatz *et al.*, *Database system concepts*. McGraw-Hill New York, 2019, vol. 7.

[17] A. Thomson *et al.*, "Calvin: fast distributed transactions for partitioned database systems," *SIGMOD*, 2012.

[18] S. Das *et al.*, "G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud," *SoCC*, 2010.

[19] Y. Lu *et al.*, "Star: Scaling transactions through asymmetrical replication," *PVLDB*, 2019.

[20] Y. Sovran *et al.*, "Transactional storage for geo-replicated systems," 2011.

[21] C. Binnig *et al.*, "Distributed snapshot isolation: global transactions pay globally, local transactions pay locally," *VLDBJ*, 2014.

[22] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," *VLDB*, 2006.

[23] H. Berenson *et al.*, "A critique of ansi sql isolation levels," in *SIGMOD*, 1995.

[24] B. Cooper *et al.*, "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB*, 2008.

[25] K. Krikellas *et al.*, "Strongly consistent replication for a bargain," 2010.

[26] M. A. Bornea *et al.*, "One-copy serializability with snapshot isolation under the hood," in *ICDE*, 2011.

[27] R. Taft *et al.*, "E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems," *PVLDB*, 2014.

[28] I. T. Bowman and K. Salem, "Optimization of query streams using semantic prefetching," *TODS*, 2005.

[29] M. Serafini *et al.*, "Clay: Fine-Grained Adaptive Partitioning for General Database Schemas," *PVLDB*, 2016.

[30] M. Slee *et al.*, "Thrift: Scalable cross-language services implementation," 2007.

[31] J. Kreps *et al.*, "Kafka: A distributed messaging system for log processing," 2011.

[32] P.-Å. Larson *et al.*, "High-performance concurrency control mechanisms for main-memory databases," *PVLDB*, 2011.

[33] C. Diaconu *et al.*, "Hekaton: Sql server's memory-optimized oltp engine," in *SIGMOD*, 2013.

[34] P. Bailis *et al.*, "Macrobase: Prioritizing attention in fast data," in *SIGMOD*, 2017.

[35] D. E. Difallah *et al.*, "Oltp-bench: An extensible testbed for benchmarking relational databases," *PVLDB*, 2013.

[36] "The Transaction Processing Council. TPC-C Benchmark (Revision 5.11)." http://www.tpc.org/tpcc/, February 2010.

[37] S. Loesing *et al.*, "On the design and scalability of distributed shared-data databases," 2015.

[38] P. A. Bernstein *et al.*, "Hyder-a transactional record manager for shared flash." in *CIDR*, 2011.

[39] I. Zhang *et al.*, "Building consistent transactions with inconsistent replication," *TOCS*, 2018.

[40] S. Mu *et al.*, "Consolidating concurrency control and consensus for commits under conflicts," in *OSDI*, 2016.

[41] X. Yu *et al.*, "Sundial: harmonizing concurrency control and caching in a distributed oltp database management system," *PVLDB*, 2018.

[42] H. A. Mahmoud *et al.*, "Maat: Effective and scalable coordination of distributed transactions in the cloud," *PVLDB*, 2014.

[43] "Nuodb architecture," http://go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf, 2017, accessed: 2018-12-07.

[44] P. Bailis *et al.*, "Coordination Avoidance in Consistent Database Systems," *PVLDB*, 2015.

[45] J. A. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination." in *ATC*, 2012.

[46] C. Li *et al.*, "Making geo-replicated systems fast as possible, consistent when necessary." 2012.

[47] R. Gupta *et al.*, "Revisiting commit processing in distributed database systems," *SIGMOD*, 1997.

[48] P. K. Reddy and M. Kitsuregawa, "Speculative locking protocols to improve performance for distributed database systems," *TKDE*, 2004.

[49] K. Ren *et al.*, "Slog: Serializable, low-latency, geo-replicated transactions," *PVLDB*, 2019.

[50] E. Zamanian *et al.*, "The end of a myth: Distributed transactions can scale," *PVLDB*, 2017.

[51] A. Dragojević *et al.*, "No compromises: distributed transactions with consistency, availability, and performance," in *SOSP*, 2015.

[52] T. Wang *et al.*, "Query fresh: log shipping on steroids," *PVLDB*, 2017.

[53] J. Li *et al.*, "Just say no to paxos overhead: Replacing consensus with network ordering," in *OSDI*, 2016.

[54] A. J. Elmore *et al.*, "Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases," *SIGMOD*, 2015.

[55] A. Shanbhag *et al.*, "A robust partitioning scheme for ad-hoc query workloads," in *SoCC*, 2017.

[56] M. S. Ardekani and D. B. Terry, "A self-configurable geo-replicated cloud storage system," 2014.

[57] M. Serafini *et al.*, "Accordion: Elastic scalability for database systems supporting distributed transactions," *PVLDB*, 2014.

[58] N. Mukherjee *et al.*, "Distributed architecture of oracle database in-memory," *PVLDB*, 2015.

[59] T. Rabl and H.-A. Jacobsen, "Query centric partitioning and allocation for partially replicated database systems," 2017.

APPENDIX

A. Dynamic Mastering Snapshot Isolation Level Proof Sketch

We provide a proof that DynaMast provides snapshot isolation (SI).

Our proof relies on three conditions that hold within the DynaMast system. First, each underlying database system guarantees that transactions execute under SI. Therefore, writes to the same data items cannot occur concurrently. Additionally, each site provides a commit order of updates represented by atomically updating the site version vector. Second, the site managers and site selector enforce that update transactions can write to only the data items mastered at the site of transaction execution. The site manager guarantees that updates occur only to data items granted ownership to the site by the site selector. Furthermore, the site manager will not `release` ownership of data items while transactions update the items. The site selector uses mutual exclusion in its remastering protocol to guarantee that it sends only a single `grant` message per data item (Section III-B). The third and final condition is that all refresh transactions eventually and reliably propagate to other sites in the order sent by the replication manager. Our implementation creates distinct Kafka logs for updates from each site, which provides the necessary ordering requirements and message delivery guarantees [31].

To show that DynaMast provides SI, we prove the following lemma and theorem:

Lemma 1. *A transaction T_1 must see the updates made by a transaction T_2 that has a commit timestamp smaller than T_1's begin timestamp.*

Proof. As described in Section III-A, when a transaction T_2 commits, it updates the site version vector. Hence if T_1 has a larger begin timestamp than T_2's commit timestamp, it must read T_2's update to the site version vector. Given such a begin timestamp, the MVCC protocol described in Section V-A1 guarantees that T_1 will read T_2's updates to records, as T_2's committed versioned records will have a version number smaller than or equal to T_1's begin timestamp. $\square$

Theorem 1. *If two transactions T_1 and T_2 have overlapping begin and commit timestamps then T_1 and T_2 can commit only if T_1 and T_2 write different data items.*

Proof. The transaction version vectors capture the begin and commit timestamps of the transaction. Therefore, T_1 has begin and commit timestamps $tvv_{B(T_1)}[\,]$ and $tvv_{T_1}[\,]$ respectively. Similarly T_2 has begin and commit timestamps $tvv_{B(T_2)}[\,]$ and $tvv_{T_2}[\,]$ respectively.

We assume, per the theorem, that both transactions T_1 and T_2 write data items, that is neither transaction is a read. Therefore, from the algorithm description in Section III-A we know that $tvv_{T_1}[\,]$ and $tvv_{T_2}[\,]$ are unique.

Recall from Section III-A that a transactions begin and commit timestamps differ only in the i-th position if site S_i is the site that executed the transaction. Therefore, we consider two cases: when T_1 and T_2 both execute at site S_i, and when T_1 and T_2 execute at sites S_1 and S_2 respectively, without loss of generality.

a) Case 1: In the first case, if both transactions execute at site S_i, then the underlying database system provides SI as stated by our first condition within the DynaMast system. Recall from Section V-A1 that our database system implementation guarantees SI by first locking the data items in the write set, which ensures that conflicting updates cannot occur concurrently. If T_1 acquires write locks after T_2 commits, then T_2's updates to the site version vector will be present in $tvv_{B(T_1)}$, which the system updates after lock acquisition. Thus, the property that T_1 and T_2 cannot update the same data item and have overlapping timestamps, holds in the local site scenario, so the theorem holds.

b) Case 2: We now argue, by way of contradiction, that it is not possible for T_1 and T_2 to execute at different sites, update the same data item d, and have overlapping begin and commit timestamps. Suppose, without loss of generality that T_1's begin timestamp is before T_2's, that is $tvv_{B(T_1)}[\,] < tvv_{B(T_2)}[\,]$.[2] Then for T_1 and T_2 to overlap, $tvv_{T_1}[\,] \geq tvv_{B(T_2)}[\,]$ must hold.

If T_1 and T_2 both update the same data item d on sites S_1 and S_2 respectively, then the site selector remastered d from S_1 to S_2 as $tvv_{B(T_1)} < tvv_{B(T_2)}$. As described in Section III-B, the `grant` request at S_1 will not complete until transactions that update d complete, and `release` does not return until all refresh transactions to d complete at S_2. The site selector does not allow transactions that update d to begin while the remastering protocol executes. The update propagation algorithm guarantees that the replication manager will apply all aforementioned updates. Therefore, before T_2 begins its transaction, S_2's site version vector reflects the update from T_1's commit, as well as a remastering operation that increments the site version vector. Thus $svv_2[\,] > tvv_{T_1}[\,]$.

However, the system sets T_2's begin timestamp $tvv_{B(T_2)}[\,]$ to at least $svv_2[\,]$, which is a contradiction because $tvv_{T_1}[\,] < svv_2[\,] \leq tvv_{B(T_2)}[\,]$ and $tvv_{T_1}[\,] \geq tvv_{B(T_2)}[\,]$ cannot both hold. Therefore these transactions must have disjoint write sets, which satisfies the theorem. $\square$

Lemma 1 and Theorem 1 together provide the requirements of SI, therefore DynaMast satisfies SI.

B. Dynamic Mastering with Session Based Snapshot Isolation

Recall from Section III-A that in addition to providing snapshot isolation (SI), the system also provides strong-session snapshot isolation (SSSI) [22]. To provide SSSI, the system enforces a *freshness rule* that ensures that a client's transaction executes on data that is at least as fresh as the state last seen by the client. Importantly, these *freshness rules* protect clients from reading data that is older than what they last read. Generally, clients see data that is more up-to-date than what they last observed because our replication scheme does not unnecessarily delay update propagation.

[2]We use the definition that a vector $v_1[\,]$ is less than another vector $v_2[\,]$, ($v_1[\,] < v_2[\,]$), if $v_1[i] < v_2[i]$ for all positions i.

1) Dynamic Mastering Strong Session Snapshot Isolation Level Proof: We now prove that our update propagation protocol together with our session-based freshness scheme enforce the ordering guarantees required to provide SSSI.

We first prove the session requirements of SSSI:

Theorem 2. *If two transactions T_1 and T_2 belong to the same session, and the commit of T_1 precedes the start of T_2 then T_2's begin timestamp is greater than T_1's commit timestamp, that is T_2 observes any state observed or created by T_1.*

Proof. In DynaMast, the begin timestamp of a transaction is at least the last commit timestamp of the previous transaction in the session. Therefore, if $tvv_{B(T_2)}[\,]$ is the begin timestamp of T_2, $tvv_{T_1}[\,]$ is the commit timestamp of T_1, and $cvv[\,]$ is the client session vector, then $tvv_{T_1}[\,] = cvv[\,] \leq tvv_{B(T_1)}[\,]$.

The blocking rules described in Appendix B guarantee that for any site of execution, T_2 will not begin the transaction until the sites version vector is at least $cvv[\,]$. Consequently, T_2 will execute and observe any state reflected in $cvv[\,]$, as required by SSSI. □

Given that DynaMast provides SI, a pre-requisite to providing SSSI, and the session requirements of SSSI, DynaMast guarantees SSSI.

C. YCSB Workload Details

To introduce workload access patterns in YCSB, we divided the key space into partitions that contain 100 contiguous keys, that are ordered based on the partition ID. In particular, partitions are more likely to be co-accessed with some partitions than others due to relationships among the data items they contain. We introduce these patterns by correlating partitions in ranges. Note that in every experiment, DynaMast has *no* a priori knowledge of these access correlations; it models client workloads and *discovers* them.

Multi-partition scan transactions start at a base partition ID drawn according to the access distribution (uniform or skew), reading all keys in the next k partitions where k is drawn uniformly between 2-10, resulting in scans of length 200 to 1000 keys. For example, in Figure 6a, the scan transaction begins at the 36^{th} partition (key 3600) and scans 3 partitions (until partition 38, key 3899).

Multi-partition read-modify-writes update three keys from a set of partitions with IDs near each other or correlated partitions. Partitions are drawn by selecting a base partition ID drawn according to the access distribution (uniform or skew ($\rho = 0.75$)), after which two keys are selected from neighboring partitions. The neighboring partitions are selected using a Bernoulli distribution centered at the base partition with a probability of success $p = 0.5$ and five trials. That is, if we sample the Bernoulli distribution and get three successes, then we will draw the next key from the base partition, whereas if we get one success, then we will draw a key from two partitions before the base partition. Using the example in Figure 6a, we select base partition 36 using the access distribution. Afterward, we sample the Bernoulli distribution twice, getting 1 and 5 successes, respectively. As the center of the distribution is three successes, these samples produce partitions 34 and 38. We then select keys from within these partitions using a uniform distribution, yielding (3472, 3601, 3890) as the write-set for the transaction. This approach induces probabilistic range-based partitions, which increases the difficulty for DynaMast to learn these patterns.

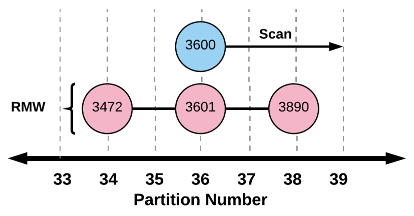
In our YCSB workloads, we introduced access locality, by having clients perform up to 1000 transactions against a set of correlated partitions, which corresponds to roughly 1 second worth of client activity. We empirically determined that increasing or decreasing this affinity by an order of magnitude does not affect throughput by more than 2%. The number of clients is fixed for a given experiment, with clients leaving after the affinity-period and other clients joining to replace them.

For the time-varying adaptivity experiment (Figure 5b), we changed client access patterns from the default range-based access correlations. To do so, we randomize the correlations by shuffling the sorted partition IDs to produce a new partition ID order. Afterwards, with this new order, correlated partitions are selected using the same neighbor partition algorithm described earlier in this section. Additionally, this experiment uses a client affinity to 25 transactions against a set of correlated partitions, after which another client replaces them with different partition correlations, which increases the number of access correlations within the experiment.

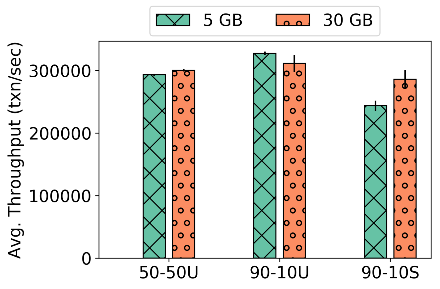
D. Performance Overhead of DynaMast

In Section VI-B7 we provided a brief discussion of the overheads of DynaMast, we now expand on these results. Figure 7 plots a breakdown of DynaMast's average transaction execution time (Figure 7) during a Uniform 50/50 RMW/Scan YCSB experiment. We divide latency into six categories: the time that it takes the site selector to lock and identify the location of data items, which accounts for 10% of the overall average response time; the time taken to make a routing decision — including remastering — that takes less than 1% of the overall time as DynaMast amortizes the cost of remastering across many transactions; the amount of time that requests spend in the network between system components, which is more than 40% of the time; the actual execution time of the database stored procedure (actual transaction logic) accounts for 45% of overall latency; the time to begin a transactions, including lock acquisition at the data site and waiting for any session state, which takes less than 1% of the overall time; and the time to commit a transaction that takes just over 1% of the overall time.

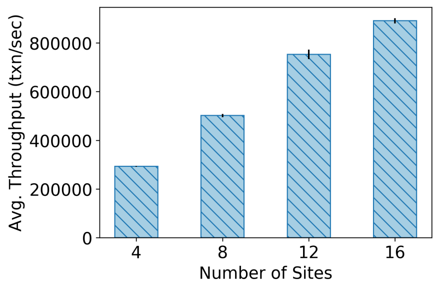
Recall that transaction routing accounts for less than 1% of the overall transaction time due to the amortization of remastering across many transactions. DynaMast achieves this low overhead because it's strategies aim to minimize future remastering by modelling inter- and intra-transactional data access correlations. Consequently, less than 1% of transactions in the YCSB and SmallBank workloads and less than 3% in TPC-C require remastering. We additionally measured the network overhead of remastering and DynaMast as a whole.



(a) Workload Generation



(b) Data Size Scalability



(c) Scaling Across Data Sites

Fig. 6: YCSB Workload Details and Scalability Experiments. (6a) illustrates how keys for RMW and scan transactions are selected, (6b) shows scalability results for larger database sizes, and (6c) highlights DynaMast's scaling capabilities with more data sites.

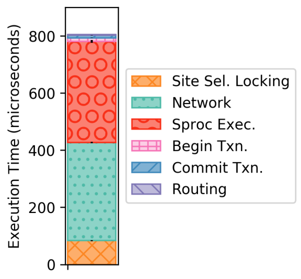


Fig. 7: A breakdown of transactional latencies.

In a YCSB workload that generated an average of 43 MB/s of stored procedure arguments, propagating refresh transactions consumed 155 MB/s of network traffic — traffic necessary in any replicated system. Remastering requests accounted for a meager 3 MB/s of network traffic. These results indicate that DynaMast adds minimal overhead to transaction execution, while significantly outperforming the other systems for different workload characteristics.

E. Scalability Results

To show that DynaMast can support larger database sizes, we evaluated its performance on our YCSB workloads using an initial database size of 30 GB. Over the course of the experiment, the database size grows to 120 GB given that the system keeps at least 4 versions of each record. Consequently, we added memory to the database machines to bring them to 128 GB of RAM. No other aspects of the system were changed for these experiments.

Figure 6b shows DynaMast's throughput for the YCSB workloads with initial database sizes of 5 GB and 30 GB that grow to occupy nearly all of the memory of the data site machines. We observe that there is little variation in performance as the database size increases for the uniform 50/50 (50-50U) and 90/10 RMW/Scan workloads (90-10U), though DynaMast does experience a slight performance degradation on the write-intensive workload due to increased data tracking and management overheads at the site selector and increased remastering. DynaMast's performance on the skewed workload (90-10S) increases because the access skew is spread across more items, and therefore decreases contention. These experiments show that DynaMast's adaptive remastering strategies and underlying infrastructure continue to perform well as database size grows.

To assess DynaMast's ability to scale across an increasing number of data sites, we evaluated DynaMast using 4, 8, 12 and 16 data sites using our uniform YCSB workload with a 50% RMW and 50% scan mix. We used a balanced read-write workload and a 5 GB database size to reduce write contention on individual partitions and attribute DynaMast's scaling capabilities to effective resource utilization.

Figure 6c shows a maximum throughput comparison for DynaMast as the number of data sites increase. We observe that DynaMast improves throughput by more than 3× as the number of sites grows by a factor of 4. DynaMast achieves this near-linear scalability because it can effectively distribute requests among sites and therefore leverage their resources to improve performance. As the number of sites increases, the rate of increase in throughput slows, a consequence of maintaining full replicas at each data site by applying transactional updates. Finally, even as nearly 900,000 transactions per second proceed through the system, the site selector is not a bottleneck.

F. Effect of Short Transactions

To stress our transaction protocol, we next evaluate DynaMast using a workload that contains short transactions. In this workload, unlike TPC-C and YCSB, transactions access at most two records, which are the minimum necessary for different sites to master data accessed in the transaction, and trigger remastering in DynaMast, 2PC in partition-store and multi-master, or data shipping in LEAP. Such a workload places a different burden on systems than the heavier TPC-C transactions as the underlying transaction protocol dominates transaction execution time, not the actual transaction logic. To do so, we use the SmallBank workload, which models a banking application where users have checking and savings accounts and can transfer money between accounts, or check account balances. In the SmallBank transaction mix, there exist three types of transactions. First, single-row update transactions that account for 45% of the workload, including the *DepositChecking* transaction that adds money to a users' checking account. Second, two-row update transactions that comprise 40% of the workload mix, as in the *SendPayment* transaction, which atomically transfers money between two accounts. Third, the read-only *Balance* transaction that reads two rows and returns the sum of these rows — a users' checking and savings accounts — and occurs 15% of the time.

As shown in Figure 8a, DynaMast has the highest throughput in the SmallBank workload, when compared to partition-store (by 15%), multi-master (by 10%), single-master (by 40%) and LEAP (more than 600%). To understand DynaMast's improvement in throughput, we examined the distribution of transaction latencies for the three transaction types within the workload, and present their tail latencies in Figures 8b, 8c and 8d. Comparing DynaMast with single-master, we can observe the load effect of routing all updates to a single-site: more than $7\times$ higher tail latencies for update transactions (Figures 8b and 8c), whereas DynaMast dissipates the update load among sites. By contrast, read-only transactions (Figure 8d) run at replicas for single-master and therefore have similar latencies to DynaMast.

LEAP initially has slightly lower single-row update latencies than DynaMast (Figure 8c), as DynaMast requires system resources to maintain replicas asynchronously. However, LEAP suffers from high tail latencies for these single-row transactions, as they must wait for data migration that is necessary for multi-row transactions to complete. As LEAP does not have smart routing strategies, LEAP suffers from frequent and expensive data migration, which increases multi-row transaction latency by nearly 40 $\times$ that of DynaMast (Figures 8b and 8d). As with LEAP, partition-store initially has a similar single-row transaction latency to DynaMast (Figure 8c); however, the requirements of the uncertain phase during distributed transaction processing force blocking — even for single-row transactions — which increases tail latency. At the tail of multi-row transactions (Figures 8b and 8d), which require the expensive two-phase commit for partition-store, we observe that DynaMast has latency that is a quarter of partition-store. Multi-master exhibits similar trends to partition-store for update transactions, as both systems incur distributed transactions, however, multi-master must propagate updates which increases tail latency slightly when compared to partition-store. For read transactions, multi-master can leverage the existence of replicas and execute at a single site, which reduces transaction latency to levels closer to that of single-master and DynaMast. This reduction in read-transaction latency compared to partition-store contributes to multi-master's higher throughput.

In summary, we find that DynaMast significantly reduces the tail latency of transactions in SmallBank, thereby demonstrating the benefits of the dynamic mastering protocol and our transaction routing strategies.

G. Additional TPC-C Results

We now provide additional experimental results for the TPC-C workload.

In the TPC-C benchmark, the *Payment* transaction is an update transaction, which records a payment by a customer. Similar to the New-Order transaction, 15% of the time, the Payment transaction updates a remote warehouse and district to simulate a customer paying for a remote order. However, unlike the New-Order transaction, the Payment transaction is much lighter as it updates only four rows by inserting a history of the payment and incrementing the payment totals for the relevant customer, district and warehouse.

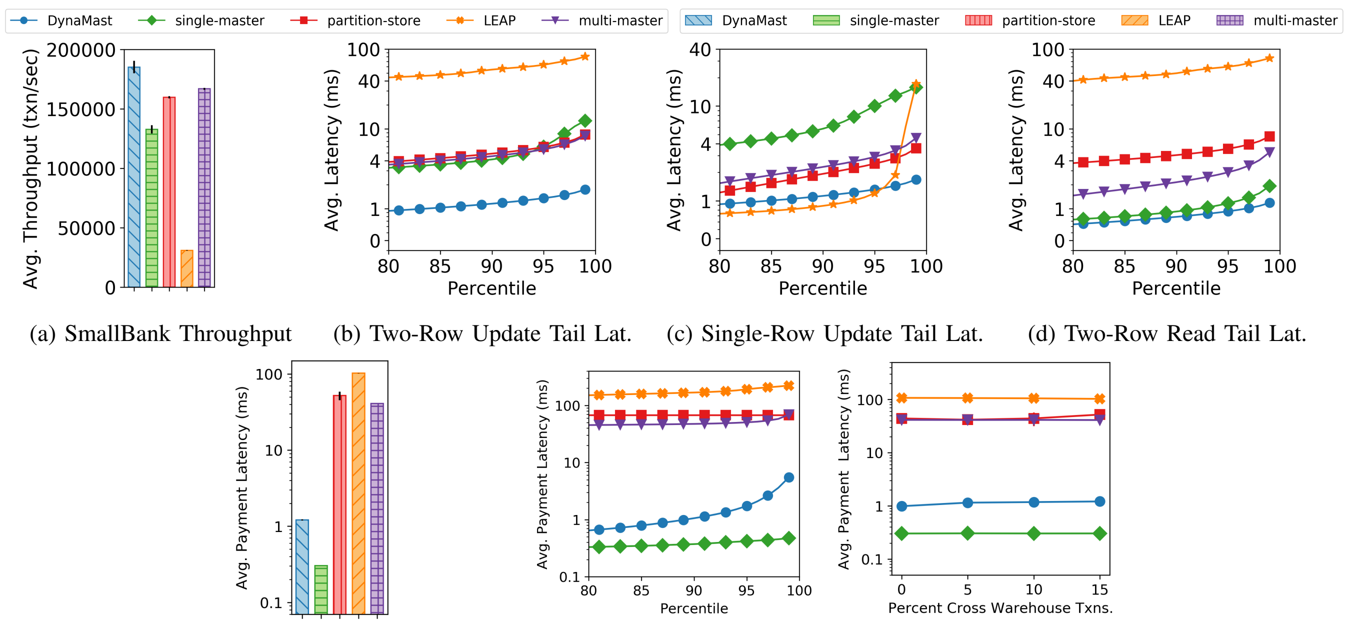
Figure 8 presents the experimental results for the Payment transaction for the TPC-C workload, with a 15% remote warehouse by default. As shown in Figure 8e, single-master has the lowest average latency at 0.3 ms, and DynaMast a mere 1.2 ms. However, as shown in Figure 4f DynaMast has higher overall throughput for the TPC-C workload overall. Consequently, DynaMast trades off a small increase in Payment transaction latency for improvements in the workload overall. There are two primary causes for this trade off in performance. First, routing all Payment transactions to a single-master does not place a heavy load on this node, when compared to the New-Order transaction. Hence, the single-master does not suffer from load effects. Second, as the workload is not perfectly partitionable, DynaMast must perform some remastering to execute transactions at a single site, an operation not necessary for single-master. Figure 8f highlights the cost of this remastering on the Payment transaction, as DynaMast only differs from single-master significantly in the slowest 10% of transactions. Although DynaMast could master all data items at a single-node, doing so would significantly increase the latency of the New-Order transaction, as shown in Figure 4c.

As with the New-Order transaction, we observe that LEAP, partition-store and multi-master experience orders of magnitude higher transaction latency for the Payment transaction, and DynaMast reduces Payment latency by 99%, 97% and 96% over these competitors, respectively (Figure 8e).

Figure 8g presents the average latency of the payment transaction as the rate of cross warehouse Payment transactions increase. Observe that latency increases just a mere 0.2 ms for DynaMast whereas latency for partition-store and multi-master increase by nearly 10 ms as the number of cross warehouse Payment transactions increases from 0 to the default 15%. As in the experiment with cross warehouse New-Order transactions, this experiment demonstrates that DynaMast adds little overhead to transaction execution, and has effective master placement strategies when compared to partition-store. Finally, observe that single-master experiences little change in Payment transaction latency as the number of cross-warehouse transactions increase because the lightweight transactions do not increase contention as was the case for the New-Order transaction.

H. Hyperparameter Selection

As described in Section IV, DynaMast employs a linear model (Equation 8) to make remastering decisions. Recall that the model employs four weights ($w_{balance}$, w_{delay}, w_{intra_txn}, w_{inter_txn}) that control the relative importance of each features present in the model. We first discuss the effect that each



(a) SmallBank Throughput (b) Two-Row Update Tail Lat. (c) Single-Row Update Tail Lat. (d) Two-Row Read Tail Lat.

(e) TPC-C Payment Lat. (f) Payment Tail Lat. (g) Payment Lat.

Fig. 8: Additional experimental results for the SmallBank and TPC-C workloads. (8a) presents the maximum throughput in SmallBank. (8b, 8c and 8d) show the tail latency for SmallBank's three transaction classses. (8e and 8f and 8g) examine the average and tail latency of the TPC-C Payment transaction. (8g) displays the TPC-C Payment latency as the percentage of cross warehouse transactions increase.

hyperparameter has on the model, and then describe how we selected the hyperparameters for our workloads.

Generally, the load balance weight, $w_{balance}$, plays the most important role in performance as it balances transactions among sites in the system. If this weight is too small, then many master copies of data items may be placed on a single node, which can result in underutilization of the distributed system and poor performance due to resource contention. Setting this value too high may result in excessive remastering because DynaMast will strive to keep data item accesses globally balanced despite access correlations.

We set the weights that control the intra-transaction access correlations (w_{intra_txn}) and the inter-transaction access correlations (w_{inter_txn}) based on the frequency of these patterns in the workload. If the workload contains no correlations of a particular type, setting the weight to zero avoids remastering based on spurious correlations. Otherwise, the weights should be set according to their prevalence and importance in the workload. Given that these features work together to localize data, the sum of their weights should be considered when comparing feature importance against $w_{balance}$ and w_{delay}. We use w_{delay} to avoid remastering to a site that has fallen behind in applying updates.

Generally, we set w_{intra_txn} and w_{inter_txn} based on the existence of these in the workload. Next, we set $w_{balance}$ in response to the sum of w_{intra_txn} and w_{inter_txn}, indicating whether locality of transactions should take priority over skew in cases where balance is imperfect (and to what degree), or vice versa. For YCSB, we set $w_{balance} = 1000000$, w_{intra_txn} = 3, w_{inter_txn} = 0, and w_{delay} = 0.5. The YCSB workload should have master copies of items placed by ranges, and load balance plays the biggest role in system performance. Consequently, we chose a large value for $w_{balance}$ to enforce balance, with w_{intra_txn} as second priority. We set w_{inter_txn} to 0 because w_{intra_txn} already captures partition relationships. For the SmallBank workload, we use the same weights as the YCSB workload but decreased the load balance weight $w_{balance}$ to 1. We lowered this weight, as the shorter transactions and smaller write sets place less load on the individual data sites, so considering data access patterns within a transaction are comparatively more important than perfect load balance.

For TPC-C, we set $w_{balance} = 0.01$, $w_{intra\_txn} = w_{inter\_txn} = 0.88$, $w_{delay} = 0.05$. We first set $w_{intra\_txn} = w_{inter\_txn}$ to values that were close to the probability that a transaction is entirely within a warehouse (90%). Because the workload is not perfectly partitionable, we set $w_{balance}$ to a small non-zero value, which ensures that the system considers load balance.

I. Scaling the Site Selector

Although we have described the site selector as a single-machine component, the site selector can be distributed if greater scalability is desired. We now present a distributed site selector design that maintains correctness and exploits our transaction routing decisions. The site selector tracks the location of the master copy of each data item and maintains statistics for its routing decisions. Since remastering is infrequent (Section VI-B7), a single-master site-selector with multiple replicas is appropriate — updates are infrequent as empirically remastering is rare. Clients may route their transactions to either the master site-selector or its replicas. The master site-selector acts like our standalone site-selector, following the same algorithms described in Sections III and IV.

When a replica site-selector receives a request, it tries to handle the routing decisions locally before falling back to the master site-selector if remastering is required; read-only transaction routing does not change. If the request is an update transaction, then the replica site-selector locally looks up master locations of the data items in the transaction's write set. If the replica site-selector determines that the transaction's write set has distributed master copies, then it routes the transaction to the master site-selector. Otherwise, if a single site masters all of the items, then the transaction is routed to that site. However, as a replica site-selector may have stale master location metadata, the site manager must abort the transaction if it no longer masters a data item. An aborted transaction is always resubmitted to the master site-selector, which performs remastering if necessary.

Given that the master site-selector performs all remastering, this distributed site-selector design provides the same correctness guarantees as in the standalone design. As shown empirically, remastering is infrequent; therefore it is unlikely for replica site-selectors to be stale, which reduces the likelihood of aborted client requests.