DynaMast: Dynamic Mastering for Replicated Databases

Author Names Blinded for Review

ABSTRACT

Ever-increasing application demands make distributed database system deployments ubiquitous. However, maintaining data consistency and transactional atomicity in these systems cause system performance limitations arising from the costly overhead of distributed transactions or bottlenecks on a single-master site. We propose DynaMast, a lazily replicated, multi-master distributed database system that eliminates distributed transactions and their overheads by guaranteeing single site transaction execution. DynaMast achieves this by dynamically transferring the mastership of data, or remastering, among sites using lightweight metadata-only operations. Moreover, DynaMast leverages remastering to intelligently place master copies to balance load and minimize future remastering. Using popular benchmark workloads, we demonstrate that DynaMast delivers superior performance over the single-master multi-replica and partitioned multi-master distributed database system architectures.

ACM Reference Format:
Author Names Blinded for Review. 2019. DynaMast: Dynamic Mastering for Replicated Databases. In *Proceedings of ACM SIGMOD Conference (SIGMOD'20).* ACM, New York, NY, USA, Article 4, 22 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

The continuous growth in data processing demands has led to a renewed focus on scalable distributed database architectures. These architectures can be divided into two categories: single-master systems that place master copies of data on a single site (node), and multi-master systems that distribute the mastership of data among multiple sites (nodes) in the system. Both of these approaches perform well on workloads for which they were designed but suffer from considerable performance degradation otherwise.

Single-master multi-replica architectures are common [2, 3, 12, 32, 58, 61] due to their simplicity and effectiveness in servicing read-intensive workloads. By placing the writable (master) copies of all data items at the single master site, all update transactions execute and commit locally at that site without expensive distributed coordination. This architecture enjoys good scalability for read-intensive workloads because read operations can be served from any site. However, as the update workload scales up, the single master site becomes a bottleneck, which results in poor system performance [27, 40, 46].

To alleviate this single site bottleneck, multi-master systems distribute master copies of data items among sites. When transactions access data located at a single site, the system's throughput scales with the number of sites. However, transactions that update data at multiple sites must use an expensive distributed commit protocol, such as two-phase commit (2PC), to ensure atomicity. It is well-known that such distributed transactions significantly degrade performance and scalability due to multiple rounds of messages and blocking [13, 29, 31, 38].

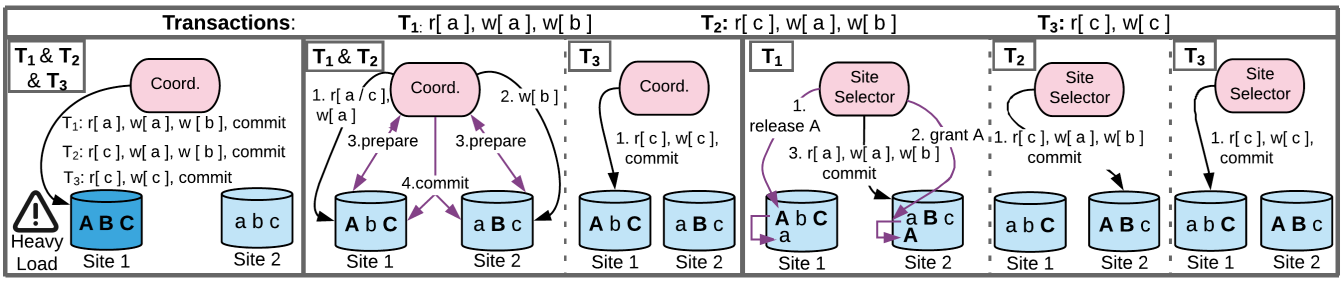
To reduce the number of distributed transactions, some approaches focus on monitoring workloads and periodically co-locating items co-accessed in transactions [1, 5, 14, 18, 25, 51, 55, 56]. However, these approaches suffer from two fundamental limitations: (i) many workloads — such as the popular TPC-C benchmark [4] — contain a significant number of distributed transactions that cannot be avoided, even when using the best-known data placement strategy [18], and (ii) when workload access patterns change, data placements must be adjusted to reduce costly distributed coordination, necessitating a workload-observation period before re-allocation occurs.

These pitfalls have given rise to methods [19, 38] that *physically move* data among sites before transactions start, *localizing* the master copies of data items to avoid distributed transactions altogether. Unfortunately, these physical data transfers worsen transaction response times because large amounts of data must be transferred over the network before transaction execution. Moreover, these approaches do not consider workload access patterns that result in unnecessary ping-pong data transfers among sites and load imbalance.

Unlike prior work, we present a distributed database system that provides the benefits of multi-master systems but ensures that transactions always execute at one site, which eliminates distributed transactions and thus their performance disadvantages. This is challenging to achieve because it requires: (i) *dynamic transfer of data mastership* among all

SIGMOD'20, June 2020, Portland, Oregon USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4



(a) Single Master **(b) 2PC** **(c) Dynamic Mastering Protocol**

Figure 1: An example of the benefits of the dynamic mastering protocol when compared to single master and two-phase commit (2PC). Uppercase bolded letters represent a master copy of a data item, with read-only replicas distributed to the other sites. Single master routes all transactions to the master. 2PC is necessary for both transactions T_1 and T_2 as the write set is distributed. By contrast, with the dynamic mastering protocol, before T_1 executes A is remastered, and then T_1 and T_2 execute as single site transactions at Site 2. Consequently, load is distributed as T_3 executes at Site 1.

sites in the distributed system while maintaining transactional consistency, and (ii) deciding the locations (sites) at which to master data items taking into account workload access patterns.

We address these challenges, including the deficiencies of prior approaches, through the design and implementation of **DynaMast**, a scalable, replicated, distributed database system that *adaptively models workload patterns* and exploits them to remaster data items intelligently. This mastership transfer effectively (i) uses lightweight metadata operations that exploit the presence of replicas for efficiency, (ii) balances load, and (iii) curtails remastering as part of future transactions. These design decisions enable DynaMast to significantly reduce transaction latency compared to both the single-master and partitioned multi-master architectures resulting in up to a 15× improvement in throughput.

The contributions of this paper are four-fold:

(1) We propose a novel replication protocol that efficiently supports dynamic mastership transfer to eliminate distributed commits while maintaining well-understood and established transactional semantics. (Section 3)
(2) We present remastering strategies that learn workload data-access patterns and exploit them to remaster data intelligently. Our strategies transfer the mastership of data among sites to minimize future remastering, which in turn reduces transaction processing latency. (Section 4)
(3) We develop DynaMast, a distributed, in-memory database system that provides low latency transaction execution through the use of the dynamic mastering protocol and remastering strategies. (Section 5)
(4) We empirically compare DynaMast against the single-master and partitioned multi-master architectures, demonstrating DynaMast's superiority on a range of workloads. (Section 6)

We discuss related work in Section 7 and conclude in Section 8. Additionally, this technical report contains detailed proofs, additional experiments, and further details on DynaMast.

2 BACKGROUND

The performance of both single-master and multi-master database systems suffer if the workload deviates from their ideal workloads. For example, write-intensive workloads overload a single-master system. Multi-master systems can distribute the write load among sites but suffer if the workload contains distributed transactions requiring communication and coordination necessary to commit atomically. By contrast, our dynamic mastering protocol provides the benefits of both architectures. It *guarantees* single-site transaction execution by intelligently transferring mastership of data items among sites, which eliminates distributed transactions.

2.1 Limitations of Single Master and Two-Phase Commit Protocols

Replicated single-master systems route all update transactions to a single site, which eliminates distributed transactions but can overload that site. Figure 1a illustrates this problem by example; a client submits three update transactions, all of which execute on the master copies (indicated by bold, uppercase letters) at the single-master (site 1). Consequently, site 1 is under heavy load as it cannot offload update transactions to the read-only replica (site 2).

By contrast, in Figure 1b, the replicated multi-master system distributes updates among sites to balance the load. In

the first step of transaction execution with 2PC (Figure 1b), the transaction (T_1) reads and writes data item a at site 1, afterward writing data item b at site 2 (Step 2). In the third step, the transaction coordinator starts the distributed commit process. In the first phase of 2PC, the transaction coordinator sends a `prepare` message to all of the involved sites (Step 3). When these sites receive a `prepare` message, they respond with a message indicating if they can commit the transaction's operations. When the coordinator receives all of the responses to the `prepare` message, 2PC enters the second phase. If all of the sites indicate that they can commit, the coordinator sends a `global commit` message to each site and each site commits the transaction and make its changes visible (Step 4). If any site indicates that it cannot commit, the coordinator instead issues a `global abort` message.

Importantly, a site cannot commit the transaction until it receives the coordinator's decision. During this *uncertain phase*, to achieve consistency, the sites must block conflicting transactions in case the coordinator aborts the transaction. This blocking and the overhead of the two rounds of communication increase the latency of distributed transactions [31]. Finally, observe that all transactions with distributed write sets, such as T_2 in Figure 1b, must execute with 2PC. However, transactions with single site write sets, such as T_3, do not require 2PC.

2.2 Dynamic Mastering

Figure 1c shows how transactions T_1, T_2 and T_3 execute using our proposed dynamic mastering protocol. First, a site selector determines that either data item a or b must be remastered for single-site transaction execution. Without loss of generality, the site selector 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 waits for any pending operations on a to complete and then releases its mastership (Step 1). Next, the site selector issues a `grant` message for a to site 2, informing site 2 that it is now the master of a (Step 2). In Step 3, T_1 executes at site 2, which can now apply the operations and commit locally without distributed coordination. Critical design decisions, such as remastering outside the boundaries of transactions and metadata only operations, ensure dynamic mastering is efficient (Section 3.2).

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), T_3 executes at a different site, thereby *distributing the write load*. Thus it is important that the site selector intelligently decide where to remaster data to balance load and minimize future remastering — objectives our remastering strategy takes into account (Section 4).

3 DYNAMIC MASTERING PROTOCOL

In the previous section, we presented an overview of remastering and its benefits. In this section, we present details of our dynamic mastering protocol and its implementation.

3.1 Maintaining Consistent Replicas

The dynamic mastering protocol exploits lazily maintained replicas to transfer ownership of data items efficiently. In lazily-replicated systems, update transactions execute on the master copies of data, and their updates are applied asynchronously to the replicas as *refresh transactions*.[1] In this paper, we focus on leveraging the dynamic mastering protocol while guaranteeing snapshot isolation (SI), which is popularly used by distributed database systems [9, 20, 53].

An SI system must assign every transaction T a begin timestamp such that T sees the updates made by transactions with earlier commit timestamps. The SI system must also ensure that if two transactions have overlapping lifespans and update the same data item, only one transaction can successfully commit [8]. 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.

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 $1 \leq i \leq m$. 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, $tvv[\,]$, which acts as a commit timestamp and ensures a consistent order of update application 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 ($tvv_{B(T)}[\,]$). When T commits, it copies $tvv_{B(T)}[\,]$ to tvv_T, increments $svv_i[i]$ and copies that value to $tvv_T[i]$. Hence, $tvv_T[\,]$ reflects the updates that were visible to T when it began executing and the position of T's commit in site S_i's sequence of committed update transactions.

The transaction version vector and site version vector indicate when a site can apply a refresh transaction. Given

[1] We use update-shipping rather than statement-replication to increase update application parallelism.

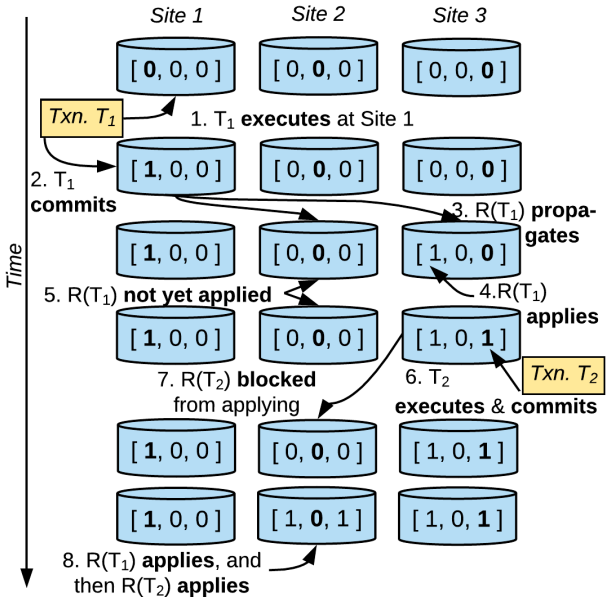


Figure 2: An example showing how commits and update propagation affect site version vectors.

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 have been read, or updated, by T. Formally, we say that T *depends* on T' if T reads or writes a data item that T' updated. 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 which updates have committed at S_j. Hence, site S_j blocks the application of $R(T)$ until the following *update application rule* holds (Equation 1):

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

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). Note that 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 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 consistent order of update application.

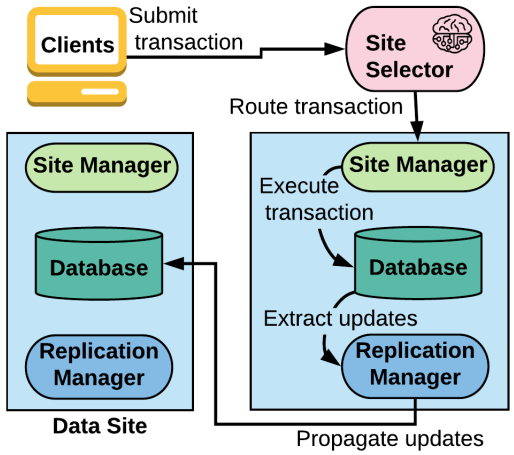


Figure 3: Generic dynamic mastering architecture.

3.2 Transaction Execution and Remastering

Figure 3 shows the architecture we use to implement the dynamic mastering protocol, which consists 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 manage 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 where the transaction will execute using the routing and remastering strategies discussed in Section 4. 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 until any ongoing transactions writing the data finish before releasing mastership of the items and responding to the site selector with the data site's version vector after the `release`. Immediately after a `release` request completes, the site selector issues a `grant` RPC to the site that will execute the transaction (Line 8). The data site receiving the `grant` waits until the 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

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 start version

```
1:  S ← determine_best_site(w) // execution site
2:  data_item_locs ← group_by_master_location(w)
3:  out_vv = {0} × m //zero_vector
4:  // In parallel:
5:  for (site S_d, data_item d) ∈ data_item_locs do
6:    if S_d ≠ S then
7:      // S_d currently masters d
8:      rel_vv ← send_release(S_d, d)
9:      completion_vv ← send_grant(S, d, rel_vv)
10:     out_vv = elementwise_max(out_vv, completion_vv)
11:   end if
12: end for
13: return (out_vv, S)
```

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. The `release` and `grant` operate as transactions, and consequently a data item waits to be remastered while being updated. In Appendix A we prove that Algorithm 1 guarantees that dynamic mastering preserves SI.

The parallel execution of `release` and `grant` operations reduce the time for clients waiting to update data being remastered. Clients begin executing as soon as their write set is remastered, benefiting from the remastering initiated by clients with intersecting write sets. The client then submits transactions directly to the database system. To complete the transaction, the client submits a `commit` or `abort` RPC 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, which allows it to commit or abort unilaterally. By contrast, during 2PC's uncertain phase, changes made by a distributed transaction are not visible to other transactions because the database is uncertain about the global outcome of the distributed transaction. Consequently, the distributed transaction blocks local transactions, which further increases transaction latency [31].

Remastering operations change only mastership location metadata, occur outside the boundaries of transactions, do not physically copy master data items, and allow single-site transaction execution once a write set is localized. These performance advantages make our dynamic mastering protocol *lightweight* and *efficient*.

3.3 Client Consistency

The dynamic mastering protocol and update application rule together provide SI. However, SI does not prevent *transaction inversion* — under SI a client may not see its previous updates in subsequent transactions. We eliminate this undesirable behaviour by adding session freshness rules to support strong session snapshot isolation (SSSI) [20]. SSSI is stronger than SI as a client will see all of its previous updates regardless of where the client executes queries; hence, numerous distributed database systems use SSSI [10, 16, 27, 34].

To enforce the client session-freshness guarantee, the system tracks each client's state using version vectors and ensures that a client's transaction executes on data that is at least as fresh as the state last seen by the client. In particular, the transaction respects the following *freshness rules*: For a client c with an m-dimensional client session version vector $cvv_c[\ ]$, when the client accesses data from a site S_i with site version vector $svv_i[\ ]$, c can execute when $svv_i[k] \geq cvv_c[k], \forall k \in (1, \ldots, m)$. After the client accesses the site, it updates its version vector to $svv_i[\ ]$.

In Appendix A , we prove that our update propagation protocol together with our session-based freshness scheme enforce the ordering guarantees required to provide SSSI.

4 SITE SELECTOR STRATEGIES

In the previous sections, we described the dynamic mastering architecture and the importance of intelligent 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 5.

Transaction routing and remastering decisions play an important role in system performance. Naïve strategies that fail to consider system balance, data freshness, and access patterns for data items when routing requests and remastering can place excessive load on sites, increase latency while waiting for data to be refreshed, and repeatedly remaster the same data between nodes. Therefore, DynaMast uses comprehensive strategies that leverage *adaptive* models to decide how to route transactions and remaster data.

4.1 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 of the 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, then the site selector must co-locate the items via remastering before transaction execution. Since remastering has upfront costs, we avoid remastering unless it is necessary and employ 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. DynaMast then remasters these data items to the site that obtained the highest score. We defer discussion of statistics collection and maintenance to Section 5.

4.1.1 Balancing Load. Workloads frequently contain access skew, which if left unaddressed can result in resource underutilization and performance bottlenecks [55]. Consequently, the first feature in our remastering strategy balances data item mastership allocation among the sites according to their write frequency, 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. Observe that if each site receives the same frequency 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:

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

A positive value for $f_{\Delta balance}(S)$ 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 $f_{\Delta balance}(S)$ 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 largely balanced then unbalancing it slightly, in exchange for less future remastering, may yield better overall performance. However, for a system that is already quite imbalanced, re-balancing it is important. Therefore, we determine how imbalanced the system is currently ($f_{balance\_dist}(B)$) and how imbalanced it would be after remastering to S ($f_{balance\_dist}(A(S))$). We incorporate this information into a scaling factor, $f_{balance\_rate}(S)$, which 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, $f_{\Delta balance}$, 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:

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

4.1.2 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. To do so, we compute the dimension-wise maximum of version vectors from each site S_i from which data will be remastered, and client c's version vector ($cvv_c[\,]$). We subtract this vector from S's current version vector and perform a dimension-wise sum to count the number of necessary updates. We express this estimate as:

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

4.1.3 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, 51]. Consequently, it is important that 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.

We consider two types of data correlations: data items that are frequently written together within a transaction (intra-transaction access correlations) and items that are indicative of future transactions' write sets (inter-transaction access correlations). In the former case, we wish to keep data items that are frequently accessed together mastered at a single site to avoid remastering for subsequent transactions. In the latter case, we anticipate upcoming transactions' write sets and preemptively remaster these items to a single site. In doing so, we avoid 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 the 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:

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

where `single_sited` returns 1 if remastering the write set of the transaction 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 will not require additional remastering. We normalize the benefit that remastering items 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 or obviate remastering for future transactions, while a negative score indicates that a larger proportion of subsequent transactions will require remastering before execution.

DynaMast also tracks inter-transactional access correlations. These occur when a client submits a transaction that accesses item d_2 within a short time interval of a transaction that accesses 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 a candidate remastering site S, we compute the *inter-transaction localization factor* similarly to Equation 7, but normalize with inter-transactional likelihood:

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

$f_{inter\_txn}(S)$ quantifies the item placement effects of remastering the current transaction's write set to candidate site S with respect to accesses to data items in the future. For ease of understanding, we have shown Equations 6 and 7 as sums over all data items d_2; for efficiency we sum over only those data items with non-zero access probabilities given d_1.

4.1.4 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.

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

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. We describe the effects of these weights and features in further detail in Appendix F .

4.2 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 chooses any of the sites that satisfy this guarantee at random, which both minimizes blocking and spreads load among sites.

5 THE DYNAMAST SYSTEM

Our distributed database system, DynaMast, implements the dynamic mastering architecture from Section 3 and consists of a site selector and data sites. Each data site comprises a site manager, database system and replication manager. The site selector uses the remastering and routing strategies from Section 4. We now detail the design of DynaMast's components; we evaluate the system in Section 6.

5.1 Client Interface

To begin a transaction, clients supply their transaction's write sets to the site selector.[2] The site selector uses this information to dynamically remaster any required data items to an appropriate site. A client then executes database transactions in DynaMast using stored procedures, which reduces communication between the client and the data site [38, 54, 57]. To end a transaction, a client sends either an abort, which

[2]Prior work [19, 38, 54, 57] demonstrates that write sets can generally be determined from transaction arguments; otherwise, DynaMast uses reconnaissance queries [57] to infer this information.

rolls back the changes or a commit, which makes the changes visible to all clients (Section 5.2). As a network optimization, DynaMast can send the begin transaction, stored procedure arguments, and end transaction messages as a single RPC.

5.2 Data Sites

A data site is responsible for executing client transactions. At a data site, the site manager interacts with the underlying database system to handle remastering requests and manage version vectors, while the replication manager propagates and applies updates. DynaMast integrates the site manager, database system and replication manager into a single component, which improves system performance. By integrating these components, we reduce concurrency control redundancy between the site manager and the database system while minimizing logging/replication overheads. The replication manager propagates updates among data sites by writing to a durable *log*, which also acts as a persistent redo log (Section 5.2.2). For fault tolerance and to scale update propagation independently of the data sites, we use Apache Kafka to store our logs and transmit updates. In Appendix G we outline how DynaMast provides fault tolerance.

5.2.1 Data Storage and Concurrency Control. The site manager stores records belonging to each relation in a row-oriented in-memory table using the primary key of each record as an index. Our database system supports reading from a snapshot of data using multi-version concurrency control (MVCC), similar to Microsoft's Hekaton engine [21, 35], to exploit the strong session SI level. Specifically, the database stores multiple versions — by default four[3] — of every record, which we call *versioned records*. When a transaction updates a record, the database creates a new versioned record. Transactions read the versioned record that corresponds to a specific snapshot so that concurrent writes do not block reads [21]. To avoid transactional aborts on write-write conflicts, DynaMast uses locks to mutually exclude writes to records, which is a lightweight and simple approach (Section 6.2.5).

When a client c submits a transaction T to site S_i for execution, c waits until the site satisfies its session by consulting the client's and site's version vectors (Section 3.3). The site then assigns the transaction a begin timestamp equal to the site's version vector $svv_i[\ ]$. Update transactions create new versioned records, which contain a version number v, and a site identifier i to indicate that a transaction at S_i created the version. On commit, $svv_i[i]$ is atomically incremented and then assigned to v. Hence, v represents T's position in the commit order at the site mastering the record.

When reading records, a transaction with begin timestamp $tvv_{B(T)}[\ ]$ must identify versioned records that are visible in its snapshot. For each record the transaction reads, it finds the most recently committed versioned record with the largest version number v and associated site identifier i such that $v \leq tvv_{B(T)}[i]$. Since v indicates the master site's version at the time of commit, $tvv_{B(T)}[i]$ determines whether that versioned record is visible in the transaction's snapshot.

5.2.2 Update Propagation. Recall from the update application rule (Section 3.1) 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. The transaction is then assigned a commit timestamp (transaction version vector) $tvv_T[\ ]$ based on the site version vector $svv_i[\ ]$. 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 3.1, and then applies T's updates as a refresh transaction by creating new versioned records with site identifier i and version $tvv_T[i]$. Finally, the replication manager makes the updates visible by setting $svv_j[i]$ to $tvv_T[i]$.

5.3 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 4.

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[4], 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 by querying a concurrent hash-table containing partition information. As it accesses each partition information, the site selector acquires the partition 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 must make a remastering decision and dynamically remaster 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 its remastering decision using vectorized single-instruction multiple-data (SIMD) operations that

[3]We use four versions because we empirically determined that it provides high performance without consuming excessive memory.

[4]By default, the site selector groups sequential data items into equally sized partitions [55] though clients can supply their own grouping.

consider each site as a destination for remastering in parallel. Given a destination site for the transaction, the site selector remasters the necessary partitions using the release and grant operations in parallel. When the site selector has remastered a partition, it updates the master location in the partition and downgrades its lock to read mode. Once all partitions are mastered at the same site, the transaction begins executing at the site, and the site selector releases all partition information locks.

The site selector builds and maintains statistics such as data item access frequency and the likelihood of data item co-access for its strategies (Section 4) to effectively remaster data. Consequently, 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. To efficiently capture these statistics, the site selector samples transaction write sets.[5] The sampled transaction, and each transaction executed within a time window Δt (Equation 7) of it — submitted by the same client — have their write sets recorded in a transaction history queue.

From these sampled write sets, the site selector determines partition level access and co-access frequencies. In particular, for each partition corresponding to the data items in a sampled write set, the site selector increments the partition information access count. Additionally, the site selector increments the intra- and inter-transaction co-access counts that correspond to partitions accessed intra- and inter-transactionally from the sampled transaction. From these access counts, the site selector quickly derives the access and co-access frequencies necessary to make its remastering decisions. Finally, DynaMast expires samples from the transaction history queue by decrementing any associated access counts to ensure it adapts to changing workloads.

In the next section, we discuss strategy computation times and tracking overheads. Our optimizations — grouping data items into partitions, and vectorized routing strategies — produce a site selector that efficiently handles concurrent clients as the workload scales up. We describe the design of a distributed site selector for even greater scalability in Appendix H .

6 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?
- How frequent is remastering, and what are its overheads?

Appendices C, D, and F.1 include experiments that answer the following additional questions:

- How well does DynaMast scale across increasing numbers of machines and larger data sets?
- How does DynaMast perform when transactions are short, and the transaction protocol dominates latency?
- How sensitive is DynaMast to remastering strategy hyperparameter values?

6.1 Experimental Setup

6.1.1 Workloads. **YCSB:** Given the ubiquity of workload access patterns [19, 38], we extended YCSB to include transactional access correlations. We also included support for multi-partition transactions, which results in distributed transactions for partition-store, remastering for DynaMast, and data-shipping for LEAP. Partitions contain 100 contiguous keys and are ordered based on their partition IDs. We extended the scan transaction in YCSB to read all of the keys from two to 10 sequentially ordered partitions based on their IDs, and extended the read-modify-write (RMW) transaction to update three keys from partitions with IDs near each other, hereafter referred to as correlated partitions.[6] Clients perform up to 1000 transactions against a set of correlated partitions, which corresponds to roughly 1 second worth of client activity.[7] The number of clients is fixed for a given experiment, with clients leaving after the affinity-period and other clients joining to replace them. 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 the majority of available memory.[8]

TPC-C: We focused our evaluation on three TPC-C transaction types: *NewOrder*, *Payment* and *StockLevel* as they represent two update intensive transactions, and a read-only transaction, respectively, and make up the bulk of both the workload and distributed transactions. In the NewOrder transaction, customers order items from warehouses, thereby performing updates to reduce the stock of the item. The read-only StockLevel transaction determines the set of recently ordered stock items below a specified threshold. The Payment transaction records a payment by a customer and updates the relevant warehouse and customer information.

[5] We use adaptable damped reservoir sampling [7] because it is efficient and enables DynaMast to adapt to changing workloads.

[6] Further workload details can be found in Appendix B .

[7] We empirically determined that increasing or decreasing this affinity by an order of magnitude does not affect throughput by more than 2%.

[8] Recall that DynaMast keeps at least four versions of each record for MVCC.

As 90% of NewOrder transactions and 85% of Payment transactions issued by a client are local to a single warehouse, we group data into partitions based on warehouse IDs. By default, we use 350 concurrent clients and a 45% NewOrder, 45% Payment, 10% StockLevel mix as this 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, which corresponds to roughly 1 GB of data that grows to more than 20 GB of in-memory data per site by the end of an experiment run. Note that having more than this number of warehouses outstrips the physical memory of a data site machine. These experiments use 8 machines (data sites).

SmallBank: Additionally, in Appendix D we use the SmallBank benchmark that simulates a banking system to examine the effect of short transactions on DynaMast as short transactions result in the transaction protocol playing a larger role in overall latency.

6.1.2 Evaluated Systems. To conduct an apples-to-apples comparison, we implement all alternative designs within the DynaMast framework. That is, each system shares the same underlying site manager, storage system, and multi-version concurrency control scheme configured to provide strong session SI. Thus, when evaluating the systems, we can directly attribute DynaMast's performance to the effectiveness of our dynamic mastering protocol and site selector strategies. Each system uses stored procedures to eliminate extra round-trips between clients and the data sites.

DynaMast: We implemented the dynamic mastering protocol and site selector strategies as the DynaMast system described in Section 5. In each experiment, DynaMast has no fixed initial data placement as we rely on its remastering strategies to distribute master copies among the sites.

Partition-Store: We implemented a partitioned database system in DynaMast, hereafter known as partition-store, that uses the 2PC protocol to coordinate distributed transactions. Partition-store uses table-specific partitioning (e.g. range, hash) to assign partitions to data sites, and the site selector routes transactions to the data sites that contain the partitions accessed in the transaction. To determine this partitioning, we use Schism [18], an offline tool that provides the best-known partitioning for OLTP workloads. A client uses 2PC to coordinate a transaction that updates data at multiple data sites. We favor partition-store by allowing it to perform as many actions in parallel as possible during 2PC. In particular, partition-store executes the prepare messages, commit messages, and actual stored procedure execution in parallel on the involved machines. Partition-store replicates read-only tables to forego propagation of updates.

Single-Master: We designated one data site as the single master site that executes all write transactions. Single-master uses asynchronous (lazy) update propagation to push updates to the replica sites that execute read-only transactions. Read transactions are split across the replicas, which improves load balance. Note that single-master is superior to using a single centralized site because the single-master system routes read-only transactions to (read-only) replicas, thereby reducing the load on the master.

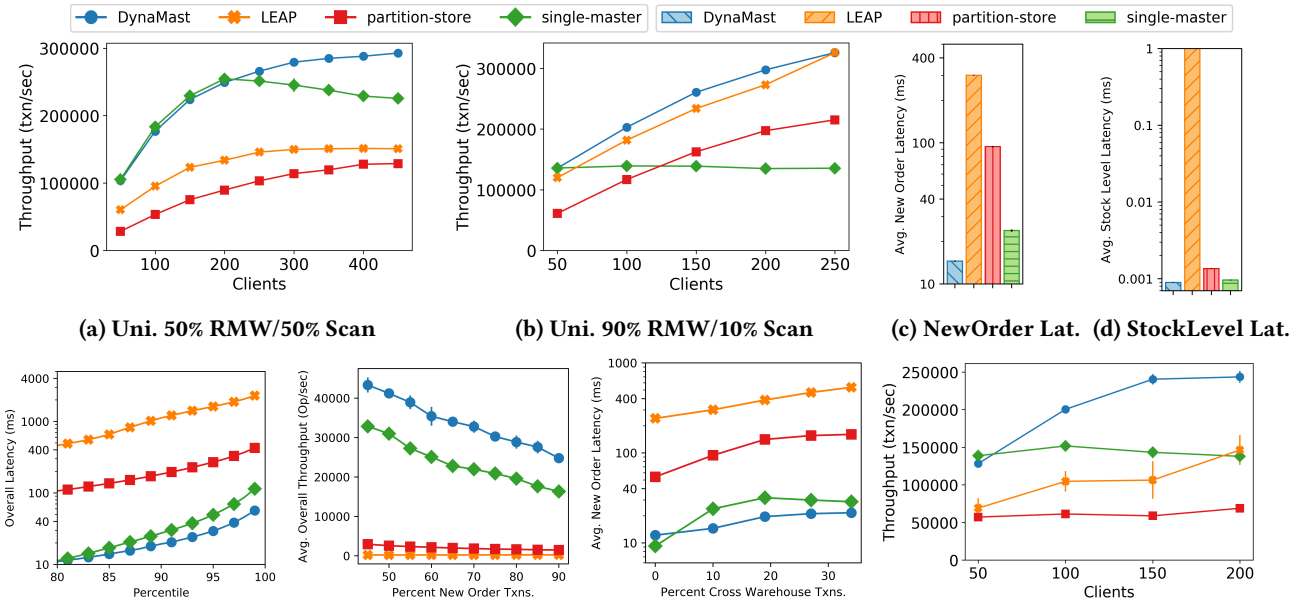
LEAP: To support the LEAP transaction localization scheme, we modified partition-store by implementing LEAP's ownership transfer protocol that reads the most recent data items from the old master and writes them to the new master [38]. As with partition-store, LEAP does not replicate data and therefore does not write data to a Kafka log or propagate transaction updates to other machines.

In our experiments, each data site executes on a 12-core machine with 32 GB of RAM. We additionally 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. All of the system components are connected using a 10 Gbit/s network. All of our results are averages of at least five, 5-minute OLTP-Bench [22] runs with 95% confidence intervals shown as bars around the means.

6.2 Results

We compare DynaMast against partition-store, single-master, and LEAP to answer our research questions from Section 6.

6.2.1 Impact of Write-Intensive Workloads. Since write transactions require remastering, we start by considering the effects of a write-intensive workload, varying the mix of write transactions to stress DynaMast further. We first evaluated the systems using YCSB with a uniform access pattern and a 50% RMW, 50% scan (read-only) transaction workload (Figure 4a). Schism [18] reports that the partitioning strategy that minimizes the number of distributed transactions is range-partitioning. Thus, we assigned partition-store a range-based partitioning scheme. DynaMast does not have a fixed partition placement and must learn access patterns to place partitions accordingly. We observe that DynaMast significantly outperforms the other systems, improving transaction throughput by 2.3× over partition-store and 1.3× over single-master. Partition-store performs poorly compared to the other systems due to additional round-trips during transaction processing. Although range-partitioning in partition-store reduces the number of distributed transactions, partition-store's inability to provide localized transaction processing results in communication and network latency overheads. LEAP's transaction localization improves the performance of partition-store by 20%. However, DynaMast achieves nearly 2× the throughput of LEAP. DynaMast executes the scan operations at replicas without the need for remastering, while LEAP needs to transfer data to localize



(a) Uni. 50% RMW/50% Scan (b) Uni. 90% RMW/10% Scan (c) NewOrder Lat. (d) StockLevel Lat.

(e) TPC-C Lat. CDF (f) TPC-C Throughput (g) NewOrder Lat. (h) Skew 90% RMW/10% Scan

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

both read-only and update transactions.

Single-master offloads scan transactions to the replicas, thereby avoiding distributed transactions without overloading the master site. As the number of clients increases, however, the single-master site becomes saturated with update transactions and performance degrades. DynaMast's remastering strategies avoid both of these pitfalls; they ensure that the master copies of data are distributed evenly across the sites without incurring distributed transactions, resulting in better resource utilization and thus better performance.

Next, we increased the proportion of RMW transactions to 90% (Figure 4b). Doing so increases the number of transactions that require remastering and increases contention. Again, we see that DynaMast outperforms single-master and partition-store in throughput by 2.5× and 1.3×, respectively. As with the last experiment, partition-store's distributed transactions and single master's master site bottleneck hamper performance and limit throughput. However, increasing the proportion of update transactions in the workload saturates the single-master site more 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. While 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 still causes LEAP to localize 10% of transactions that are read-only. By contrast, DynaMast executes read-only transactions at replicas, without remastering.

6.2.2 Impact of Complex Write Transactions. Next, we examined the effect of a workload with more complex write transactions by using TPC-C. The NewOrder transaction writes dozens of keys as part of its execution, which increases the challenge of efficiently and effectively placing data via remastering. Although TPC-C is not fully-partitionable due to cross-warehouse NewOrder and Payment transactions, Schism confirms that the well-known partitioning by warehouse strategy minimizes the number of distributed transactions. Thus, we partition partition-store by warehouse, but DynaMast must learn appropriate partition placements.

We first examine the differences in NewOrder transaction latency among DynaMast and its comparators (Figure 4c). On average, DynaMast reduces the time taken to complete the NewOrder transaction by 40% when compared to single-master. This significant reduction in latency occurs because DynaMast processes NewOrder transactions at all the data sites thereby spreading the load across all of the sites, and not just at a single master site that suffers from heavy load as in the single-master system. As shown in Figure 4e, the largest difference in transaction latency between DynaMast and single master exists in the slowest 10% of transactions that suffer 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 NewOrder latency by 85% when compared to partition-store. DynaMast achieves this

reduction because it always runs the NewOrder transaction at a single site, and therefore does not incur the cost of the multi-site 2PC. Indeed, as shown by the tail latencies in Figure 4e, partition-store has a significantly higher (10×) 90^{th} percentile latency compared to DynaMast.

Compared to LEAP, DynaMast reduces NewOrder latency by 96%. LEAP has no routing strategies and thus continually transfers data between the sites to avoid 2PC, unlike DynaMast's intelligent strategies that limit remastering. LEAP's localization efforts result in high transfer overheads and contention, which manifests in a 99^{th} percentile latency that is 40× higher than DynaMast's as shown in Figure 4e.

DynaMast has low average latency for the StockLevel transaction (Figure 4d) because efficient update propagation rarely causes transactions to wait for updates, in addition to the use of multi-version concurrency control that does not block read-only transactions. As single-master also benefits from these optimizations, it has a similar latency to DynaMast. Partition-store's average latency is higher because the StockLevel transaction may depend on stock from multiple warehouses, which necessitates a multi-site transaction. Although read-only multi-site transactions do not require 2PC, they are subject to straggler effects as they must wait for all responses to complete. The slowest site's response time determines read-only transactions' latency, resulting in higher average latency. By contrast, LEAP has orders of magnitude higher StockLevel latency than DynaMast as it must localize read-only transactions as it cannot exploit replicas.

Figure 4f shows how throughput varies with the percentage of NewOrder transactions in the workload. When NewOrder transactions dominate the workload, DynaMast delivers more than 15× the throughput of partition-store, a consequence of partition-store's high cost of transactions resulting from multiple round trips and high tail latencies. Similarly, DynaMast has 20× the throughput of LEAP; LEAP lacks intelligent master transfer strategies and consequently continually moves data to avoid 2PC. DynaMast's significantly lower NewOrder transactions latency compared to single-master results in throughput 1.64× that of single-master.

In Appendix E , we show that DynaMast reduces Payment transaction latency by 99% and 97% of LEAP and partition-store's latency respectively; when compared to single-master DynaMast achieves higher overall throughput as it routes updates to multiple sites, which reduces latency for the heavy NewOrder transaction, a trade-off that increases Payment latency by a mere 0.9 ms.

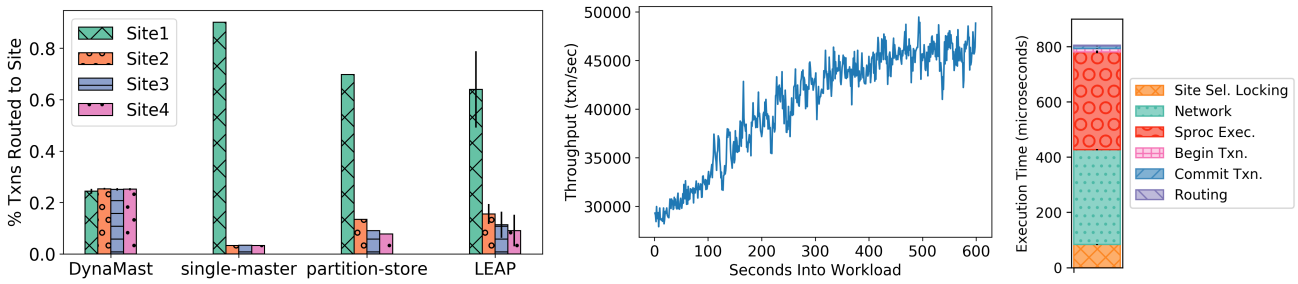
Decreasing Transaction Access Locality: We study the effect of increasing the ratio of cross-warehouse NewOrder transactions on NewOrder average transaction latency (Figure 4g). Although the best partitioning remains warehouse-based, these cross-warehouse transactions induce more distributed transactions in partition-store. Recall that 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.

Compared to partition-store, DynaMast reduces NewOrder transaction latency by 87% on average when one-third of NewOrder transactions cross warehouses. Partition-store's average NewOrder latency increases by almost 3× from when there are no cross warehouse transactions to when one-third of the transactions cross warehouses, compared to an increase of just 1.75× in DynaMast. Similar to partition-store, LEAP increases the NewOrder latency by more than 2.2× as more distributed transactions necessitate further data transfers, demonstrating the benefit of DynaMast's strategies and lightweight remastering over LEAP.

Partition-store's latency increases because cross warehouse transactions also slow down single warehouse transactions [31]. Recall that 2PC requires holding locks during the uncertain phase to prevent uncommitted changes from being visible to other transactions, which therefore also blocks local transactions from executing. As the number of cross-warehouse transactions increases, DynaMast increasingly masters more data items at one site. However, DynaMast still reduces NewOrder latency by 25% when compared to single-master. Because DynaMast does not route all NewOrder transactions to a single site, DynaMast does not suffer from the same heavy load effects as single-master.

6.2.3 Impact of 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).[9] DynaMast significantly outperforms its comparators, and improves throughput over the range-based partition-store by 4×, single-master by 1.8×, and LEAP by 1.6× LEAP. Partition-store's performance suffers because range partitioning places the most heavily-accessed partitions at one site (Figure 5a), resulting in longer transaction execution times due to resource contention. LEAP improves throughput compared to partition-store as it guarantees transactions execute at one site, but also suffers from excessive co-location of partitions at one site. Resource contention is also present for the single-master approach as all of the update transactions must be executed at the master site. By contrast, DynaMast spreads frequently updated partitions over sites, which balances load and improves performance.

[9]For brevity, we omit a skewed 50%/50% RMW/scan workload, which has similar performance trends.



(a) Skew Transaction Routing **(b) DynaMast Adaptivity Over Time** **(c) Transaction Execution**

Figure 5: Routing graphs for a skewed workload, adaptivity results, and a breakdown of transactional latencies.

6.2.4 Impact of Changing Workloads. Access patterns in workloads often change [55], which results in degraded performance for a fixed mastership allocation. A key feature of DynaMast is its ability to adapt to these workload changes by localizing transactions. To demonstrate this capability, we conducted a YCSB experiment where we placed master copies of data using range partitioning, but use a workload with randomized partition correlations. Hence, DynaMast must learn these new correlations and remaster data accordingly. The experiment used 100 clients running 100% RMW transactions, with a skewed data access pattern. Each client executes 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 and challenges DynaMast with high load, contention, and an initial remastering of nearly every data item. As shown in Figure 5b, DynaMast continuously improves throughput over the measurement interval; by the end of the experiment, throughput increases by 1.6× compared to the starting point. This improvement shows that DynaMast learns new relationships between partitions and its strategies leverage the learned information effectively to localize transactions via remastering.

6.2.5 Performance Breakdown. The previous experiments demonstrate DynaMast's superior performance over single-master, partition-store and LEAP for different workloads. To show that DynaMast achieves these performance improvements without causing significant overheads in transaction processing or routing, we plotted a breakdown of DynaMast's average transaction execution time (Figure 5c) during a Uniform 50/50 RMW/Scan YCSB experiment. Network latencies between clients and system components account for 40% of transaction execution time, which highlights the importance of localizing transactions in distributed databases and avoiding multi-round distributed commit protocols. The time to execute the actual transaction logic at a data site accounts for 45% of the latency. The breakdown shows that on average, routing decisions, wait time for updates to arrive to begin transaction execution, and commit time are small relative to network and stored procedure execution latency. Thus, DynaMast adds minimal overhead to transaction execution time while significantly outperforming the other systems for different workload characteristics.

We next assess how effective DynaMast's strategies are at minimizing remastering. DynaMast remasters data according to its learned access correlations, resulting in a single remastering operation localizing many transactions in the workload. Consequently, less than 1% of transactions in the YCSB workloads and less than 3% in TPC-C require remastering. Finally, we measured the network overhead of DynaMast. 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.

6.2.6 Additional Results. The appendices contain further experimental results. Using the SmallBank workload, we examine the effect of short transactions on DynaMast and show tail latency reductions of 4 to 40× (Appendix D). We demonstrate DynaMast's robustness to hyperparameter settings — throughput varies by at most 8% as hyperparameters change (Appendix F.1). Finally, we show DynaMast's excellent scalability (Appendix C), as more memory becomes available, DynaMast maintains its throughput on larger database sizes. Similarly, as DynaMast adds more data sites, growing from 4 to 16 nodes, throughput scales near-linearly and increases by more than 3×.

7 RELATED WORK

DynaMast is the first transactional system that both intelligently places data by considering data access patterns and eliminates distributed transactions through remastering.

Shared-data architectures decouple transaction processing from data storage [39] by forwarding transaction reads and writes to a shared-storage subsystem supported by a fast network. This approach requires the shared-storage system

to support atomicity and consistency guarantees for transactional data accesses. Despite presenting a unified storage interface, data access to remote storage nodes incurs network communication overhead. To provide atomicity and consistency for multi-key operations in a shared-data environment, G-Store uses a 2PL-like protocol [19]. L-Store [38] extends the application of these transactional properties beyond the lifespan of a single transaction, which reduces the overhead of future transaction execution. By contrast, DynaMast manages database replicas with lazy replication, and its remastering protocol guarantees that transactions execute on a local copy of the data without network communication necessary in shared-data systems. Furthermore, DynaMast considers data access correlations and load balance across the distributed system in its remastering decisions.

Coordination avoidance [6] exploits application invariants [17] and commutative data types [36, 53] to avoid distributed transactions, by merging diverging updates asynchronously. However, not all workloads support these invariants, so they require distributed transactions. Regardless of the operation, DynaMast guarantees single site transaction execution.

To mitigate the blocking effect of 2PC, some systems [28, 31, 48] use speculative execution to make globally uncommitted changes visible. Until a global commit arrives, transactions that view these changes are forced to block. In DynaMast, once a transaction begins executing, it does not require nor rely on distributed state information.

Deterministic databases [57] 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. Unlike such systems, DynaMast executes transactions as they arrive and remasters on-the-fly only when necessary.

Sundial [62] introduces data caches into a partitioned system to minimize the latency of distributed transactions using *logical leases* as the basis of both the cache coherence protocol and its optimistic concurrency control scheme. Similarly, MaaT [42] uses logical timestamps to change the commit order among transactions through explicit coordination. In contrast to these systems, DynaMast uses its remastering strategies to learn the application's workload and minimize ownership transfer through remastering of data items.

Some systems exploit advances in hardware such as low-latency remote memory accesses [15, 23, 60, 63], programmable network switches [37], hardware transactional memory [15], or non-volatile memory [23, 60] to improve throughput in distributed database systems. DynaMast improves transaction processing without requiring specialized, cost-prohibitive, hardware technologies that are not widely available [49].

TAPIR [64] and Janus [43] address the overheads of distributed transaction processing by coupling the transaction consistency protocol with the replication protocol necessary for fault tolerance. However, these systems — unlike DynaMast — do not guarantee single site transaction execution and thus require a distributed commit protocol. Furthermore, TAPIR and Janus statically assign master copies of data to nodes and do not support dynamic mastership changes.

NuoDB [1] executes transaction updates on locally cached copies of data partitions and propagates updates to remote caches. To ensure transactions do not conflict, NuoDB requires transactions to communicate with any leaders of each updated partition in the transaction, hence distributed coordination can occur for transactions. NuoDB changes partition leaders if a site or partition becomes unavailable while DynaMast remasters data items on a per transaction basis using a comprehensive cost-based strategy.

Repartitioning systems[5, 14, 18, 25, 44, 50, 52, 55] periodically change data placement as workloads change to reduce the number of distributed transactions. However, distributed transactions remain unless the workload is perfectly partitionable, whereas DynaMast eschews distributed transactions entirely using metadata-only remastering operations.

DynaMast's site selector strategies build upon previous work in modelling database workloads, and data access patterns [11, 24, 26, 41, 45, 56], to estimate resource usage, predictively execute queries and select indexes. However, DynaMast uses its workload model to decide where to execute transactions and whether or not to remaster data items.

Rabl and Jacobsen propose partially replicated, partitioned databases to avoid distributed read-only transactions [47]. Using a priori knowledge of query classes, they statically partition data to guarantee single-site read-only transactions, but actively replicate writes using a distributed commit protocol. By contrast, DynaMast learns and uses intelligent master data placement to avoid distributed transactions altogether.

In a single-master environment, KuaFu [30] applies refresh transactions in parallel at replicas and orders them by reconstructed transaction dependencies to ensure consistency. By contrast, DynaMast remasters data items among sites and uses version vectors to order refresh transactions.

8 CONCLUSION

We presented DynaMast, a replicated, multi-master distributed database system that guarantees single site transaction execution through dynamic mastering. As shown experimentally, DynaMast's novel remastering protocol and intelligent strategies are lightweight, ensure balanced load among sites, and minimize remastering as part of future transactions. Consequently, DynaMast eschews expensive distributed commit protocols and avoids the single-master site bottleneck. These design strengths allow DynaMast to improve performance by up to 2.5× and 15× over the popular single-master and partitioned multi-master database systems, respectively.

REFERENCES

[1] 2017. NuoDB Architecture. http://go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf. Accessed: 2018-12-07.

[2] 2019. Huawei Relational Database Service. https://support.huaweicloud.com/en-us/productdesc-rds/rds-productdesc.pdf. Accessed: 2019-06-01.

[3] 2019. MySQL: Primary-Secondary Replication. https://dev.mysql.com/doc/refman/8.0/en/group-replication-primary-secondary-replication.html. Accessed: 2019-02-01.

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

[5] Masoud Saeida Ardekani and Douglas B Terry. 2014. A self-configurable geo-replicated cloud storage system. *OSDI*.

[6] Peter Bailis et al. 2015. Coordination Avoidance in Consistent Database Systems. *PVLDB* 4 (2015).

[7] Peter Bailis et al. 2017. Macrobase: Prioritizing attention in fast data. In *SIGMOD*.

[8] Hal Berenson et al. 1995. A critique of ANSI SQL isolation levels. In *SIGMOD*.

[9] Carsten Binnig et al. 2014. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *VLDBJ* 23 (2014).

[10] Mihaela A Bornea et al. 2011. One-copy serializability with snapshot isolation under the hood. In *ICDE*.

[11] Ivan T. Bowman and Kenneth Salem. 2005. Optimization of Query Streams Using Semantic Prefetching. *TODS* 30 (2005).

[12] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}: FacebookâĂŹs Distributed Data Store for the Social Graph. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 49–60.

[13] Prima Chairunnanda, Khuzaima Daudjee, and Tamer Ozsu. 2014. ConfluxDB: Multi-master Replication for Partitioned Snapshot Isolation Databases. *PVLDB* 11 (2014).

[14] Wilson Wai Shun Chan et al. 2008. Techniques for multiple window resource remastering among nodes of a cluster. US Patent 7,379,952.

[15] Haibo Chen et al. 2017. Fast in-memory transaction processing using RDMA and HTM. *TOCS* 35 (2017).

[16] Brian F. Cooper et al. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB* (2008).

[17] James A Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination.. In *ATC*.

[18] Carlo Curino et al. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB* 1-2 (2010).

[19] Sudipto Das et al. 2010. G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. *SoCC* (2010).

[20] Khuzaima Daudjee and Kenneth Salem. 2006. Lazy database replication with snapshot isolation. *VLDB* (2006).

[21] Cristian Diaconu et al. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*.

[22] Djellel Eddine Difallah et al. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB* 7 (2013).

[23] Aleksandar Dragojević et al. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*.

[24] Naiqiao Du et al. 2009. Towards workflow-driven database system workload modeling. *DBTest*.

[25] Aaron J. Elmore et al. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. *SIGMOD* (2015).

[26] Brad Glasbergen et al. 2018. Apollo: Learning Query Correlations for Predictive Caching in Geo-Distributed Systems. In *EDBT*.

[27] Jonathon Goldstein et al. 2004. MTCache: Mid-Tier Database Caching for SQL Server. *ICDE* (2004).

[28] Ramesh Gupta et al. 1997. Revisiting commit processing in distributed database systems. *SIGMOD* (1997).

[29] Rachael Harding et al. 2017. An evaluation of distributed concurrency control. *PVLDB* 10 (2017).

[30] Chuntao Hong et al. 2013. KuaFu: Closing the parallelism gap in database replication. *ICDE*.

[31] Evan PC Jones et al. 2010. Low overhead concurrency control for partitioned main memory databases. *SIGMOD*.

[32] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *ICDE*.

[33] Jay Kreps et al. 2011. Kafka: A distributed messaging system for log processing. *NetDB*.

[34] Konstantinos Krikellas et al. 2010. Strongly consistent replication for a bargain. *ICDE*.

[35] Per-Åke Larson et al. 2011. High-performance concurrency control mechanisms for main-memory databases. *PVLDB* (2011).

[36] Cheng Li et al. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. *OSDI*.

[37] Jialin Li et al. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*.

[38] Qian Lin et al. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. *SIGMOD* (2016).

[39] Simon Loesing et al. 2015. On the design and scalability of distributed shared-data databases. *SIGMOD*.

[40] Qiong Luo et al. 2002. Middle-tier Database Caching for e-Business. In *SIGMOD*.

[41] Lin Ma et al. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. *SIGMOD*.

[42] Hatem A Mahmoud et al. 2014. Maat: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB* 7 (2014).

[43] Shuai Mu et al. 2016. Consolidating concurrency control and consensus for commits under conflicts. In *OSDI*.

[44] Niloy Mukherjee and othersj. 2015. Distributed architecture of Oracle database in-memory. *PVLDB* (2015).

[45] Andrew Pavlo et al. 2011. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB* 5, 2 (2011).

[46] Andrew Pavlo et al. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. *SIGMOD* (2012).

[47] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. *SIGMOD*.

[48] P Krishna Reddy and Masaru Kitsuregawa. 2004. Speculative locking protocols to improve performance for distributed database systems. *TKDE* (2004).

[49] Arjun Roy et al. 2015. Inside the social network's (datacenter) network. In *SIGCOMM*.

[50] Marco Serafini et al. 2014. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB* 7 (2014).

[51] Marco Serafini et al. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *PVLDB* 4 (2016).

[52] Anil Shanbhag et al. 2017. A robust partitioning scheme for ad-hoc query workloads. In *SoCC*.

[53] Yair Sovran et al. 2011. Transactional storage for geo-replicated systems. *SOSP*.

[54] Michael Stonebraker et al. 2007. The end of an Architectural Era: (It's Time for a Complete Rewrite). *VLDB* (2007).

[55] Rebecca Taft et al. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *PVLDB* 3 (2014).

[56] Rebecca Taft et al. 2018. P-Store: An Elastic Database System with Predictive Provisioning. *SIGMOD*.

[57] Alexander Thomson et al. 2012. Calvin: fast distributed transactions for partitioned database systems. *SIGMOD* (2012).
[58] Alexandre Verbitski et al. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. *SIGMOD*.
[59] Guozhang Wang et al. 2015. Building a replicated logging system with Apache Kafka. *PVLDB* 12 (2015).
[60] Tianzheng Wang et al. 2017. Query fresh: log shipping on steroids. *PVLDB* 11 (2017).
[61] Shuqing Wu and Bettina Kemme. 2005. Postgres-R (SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*.
[62] Xiangyao Yu et al. 2018. Sundial: harmonizing concurrency control and caching in a distributed OLTP database management system. *PVLDB* 11 (2018).
[63] Erfan Zamanian et al. 2017. The end of a myth: Distributed transactions can scale. *PVLDB* 10 (2017).
[64] Irene Zhang et al. 2018. Building consistent transactions with inconsistent replication. *TOCS* (2018).

A STRONG SESSION SNAPSHOT ISOLATION PROOF SKETCH

We provide a proof sketch that shows that DynaMast guarantees Strong Session Snapshot Isolation (SSSI) [20]. A prerequisite to providing SSSI is guaranteeing Snapshot Isolation (SI), which we prove first.

Our proof sketch 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 3.2). 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 [33].

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 3.1, 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 5.2.1 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. □

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 3.1 we know that $tvv_{T_1}[\,]$ and $tvv_{T_2}[\,]$ are unique.

Recall from Section 3.1 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.

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 5.2.1 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.

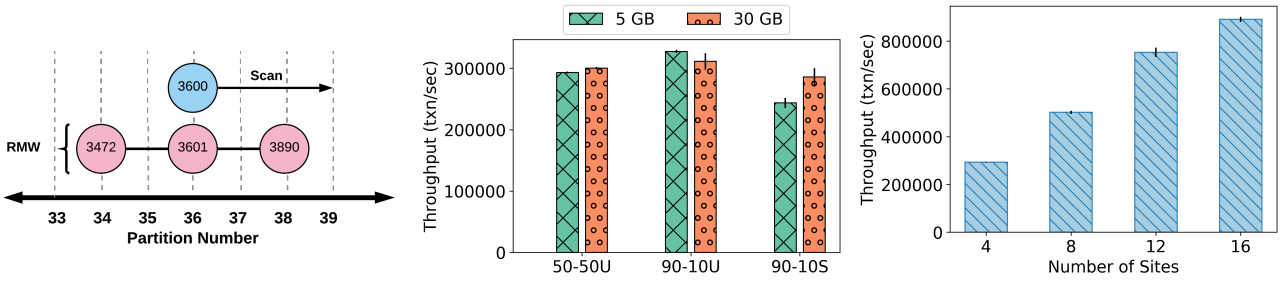
Case 2: We now argue, by way of contradiction, that it is not possible for T_1 and T_2 to execute at different sites, update the same data item d, and have overlapping begin and commit timestamps. Suppose, without loss of generality that T_1's begin timestamp is before T_2's, that is $tvv_{B(T_1)}[\,] < tvv_{B(T_2)}[\,]$.[10] 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 3.2, 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. □

Lemma 1 and Theorem 1 together provide the requirements of SI, therefore DynaMast satisfies SI. We now prove the session requirements of SSSI:

[10] 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.



(a) Workload Generation (b) Data Size Scalability (c) Scaling Across Data Sites

Figure 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.

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 Section 3.3 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 and the session requirements of SSSI, DynaMast guarantees SSSI.

B YCSB WORKLOAD DETAILS

Our YCSB workload induces access correlations among partitions to emulate client access patterns [19, 38]. 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. 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 have a base partition ID drawn according to the access distribution (uniform or skew ($\rho = 0.75$)), after which 2 keys are selected from neighboring partitions. The neighboring partitions are selected using a Bernoulli distribution centered at the base partition with probability of success $p = 0.5$ and 5 trials. That is, if we sample the Bernoulli distribution and get 3 successes, we will draw the next key from the base partition, whereas if we get 1 success 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 3 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.

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.

C 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 intelligent 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.

D 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, 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 user's 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 user's checking and savings accounts — and occurs 15% of the time.

As shown in Figure 7a, DynaMast has the highest throughput in the SmallBank workload, when compared to partition-store (by 15%), 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 7b, 7c and 7d. Comparing DynaMast with single-master, we can observe the load effect of routing all updates to a single-site: more than 7× higher tail latencies for update transactions (Figures 7b and 7c), whereas DynaMast dissipates the update load among sites. By contrast, read-only transactions (Figure 7d) run at replicas for single-master and therefore have similar latencies to DynaMast.

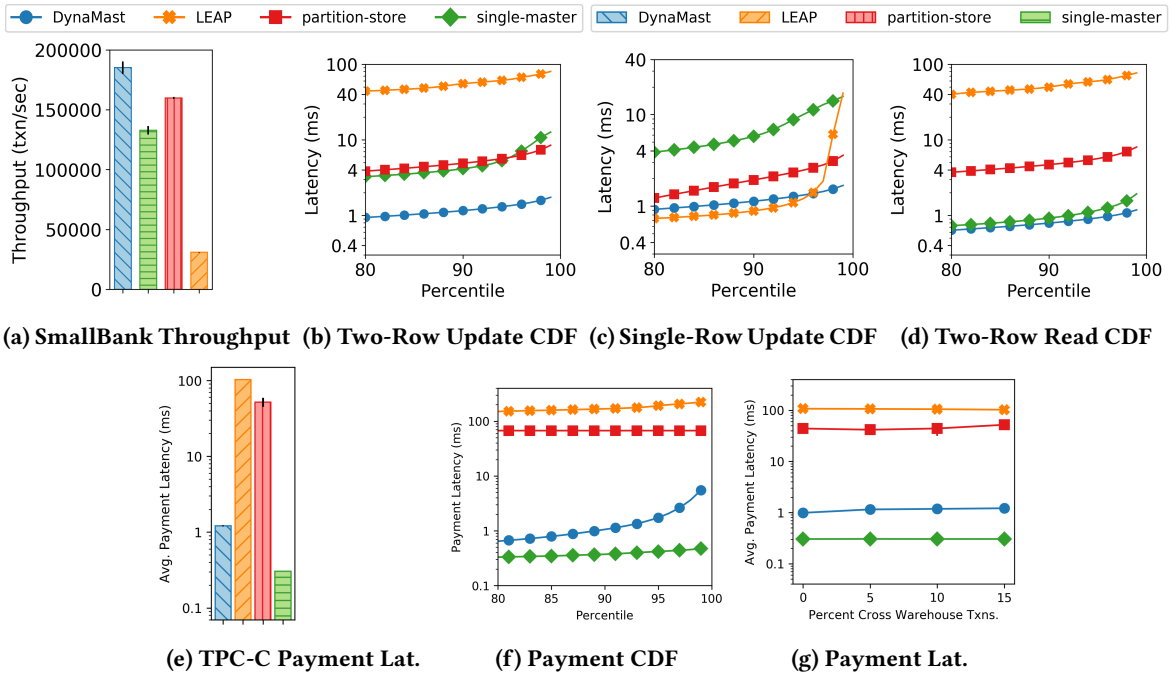
LEAP initially has slightly lower single-row update latencies than DynaMast (Figure 7c), as DynaMast requires system resources to maintain replicas asynchronously. However, LEAP suffers from high tail latencies for these single-row transactions, as they must wait for data migration that is necessary 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 × that of DynaMast (Figures 7b and 7d). As with LEAP, partition-store initially has a similar single-row transaction latency to DynaMast (Figure 7c); 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 7b and 7d), which require the expensive two-phase commit for partition-store, we observe that DynaMast has latency that is a quarter of partition-store.

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.

E 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 NewOrder 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 NewOrder transaction, the Payment transaction is much lighter as it updates only four rows by inserting a



(a) SmallBank Throughput **(b) Two-Row Update CDF** **(c) Single-Row Update CDF** **(d) Two-Row Read CDF**

(e) TPC-C Payment Lat. **(f) Payment CDF** **(g) Payment Lat.**

Figure 7: Additional experimental results for the SmallBank and TPC-C workloads. (7a) presents the maximum throughput in SmallBank. (7b, 7c and 7d) show the tail latency for SmallBank's three transaction classses. (7e and 7f and 7g) examine the average and tail latency of the TPC-C Payment transaction. (7g) displays the TPC-C Payment latency as the percentage of cross warehouse transactions increase.

history of the payment and incrementing the payment totals for the relevant customer, district and warehouse.

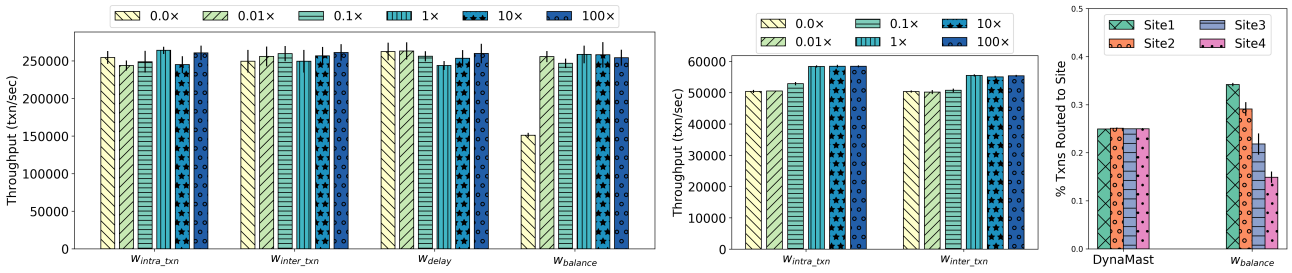
Figure 7 presents the experimental results for the Payment transaction for the TPC-C workload, with a 15% remote warehouse by default. As shown in Figure 7e, single-master has the lowest average latency at 0.3 ms, and DynaMast a mere 1.2 ms. However, as shown in Figure 4f DynaMast has higher overall 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 NewOrder 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 7f 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 NewOrder transaction, as shown in Figure 4c.

As with the NewOrder transaction, we observe that LEAP and partition-store experience orders of magnitude higher transaction latency for the Payment transaction, and DynaMast reduces Payment latency by 99% and 97% over its competitors, respectively (Figure 7e).

Figure 7g 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 increases 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 NewOrder 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 light-weight transactions do not increase contention as was the case for the NewOrder transaction.

F HYPERPARAMETER SELECTION

As described in Section 4, 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



(a) YCSB Sensitivity Experiments **(b) Co-Access Sensitivity** **(c) Routing Sensitivity**

Figure 8: Sensitivity Experiments. (8a) shows the throughput of DynaMast using a skewed YCSB workload with a 50% RMW, 50% Scan mix, as default parameters are scaled indepedently by orders of magnitude. (8b) illustrates the sensitivity of the intra and inter-transactional parameters on a changed workload. (8c) examines how the balance parameter effects transaction routing.

in the model. We first discuss the effect that each 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.

F.1 Effects of Hyperparameter Settings

In Figure 8a we experimentally vary each of the hyperparameters using a skewed YCSB workload, that contains a 50% RMW and 50% scan mix. For each hyperparameter we vary from our default parameter — normalized to 1.0 — by scaling by one or two orders of magnitude up, denoted as 10.0 and 100.0, and down, denoted as 0.1 and 0.01.[11] Finally, we set each parameter to 0, to determine the effect of removing the feature from the site selectors strategy.

Figure 8a demonstrates that DynaMast's performance is robust to variation in hyperparameters. When every parameter is non-zero, throughput remains at 8% of the maximal throughput. However, when $w_{balance}$ is 0, throughput drops by nearly 40%. This behaviour arises as $w_{balance}$ is the only hyperparameter that aims to keep the load balanced in the system, hence when we set it to 0, DynaMast increasingly masters data at a one master site. By contrast, the w_{intra_txn} and w_{inter_txn} hyperparameters complement each other and hence when one is 0, the other promotes co-location of co-accessed data items. Similarly, w_{delay} encourages re-mastering data items to sites with fresh replicas, hence if no site is lagging

[11] Note for w_{inter_txn} which is 0 by default we use 1.0 as the base parameter.

in applying updates, the hyperparameter has little effect.

To further examine the effect of the w_{intra_txn} and w_{inter_txn} hyperparameters, we performed sensitivity experiments after a workload change, as in Section 6.2.4. After a workload change, the ability of the site selector to learn and understand data item access patterns is of utmost importance. To further understand the effects of these parameters, we set the balance hyperparameter to a relatively small number ($w_{balance} = 0.01$).

Figure 8b shows that throughput increases as w_{intra_txn} increases from 0 to a relative value of 1 that we use for our experiments. This 16% improvement in throughput demonstrates that the site selector captures the intra- transactional co-access patterns and uses them to co-locate the master copies of co-accessed data items. As the hyperparameter value continues to increase, throughput does not increase as the site selector captures the access patterns sufficiently. When we vary the inter-transactional co-access parameter (w_{inter_txn}), we observe a similar trend as throughput increases by 10%.

The distribution of requests for the experiments in Figure 8b is shown in Figure 8c. Compared to the even baseline routing in DynaMast, a small $w_{balance}$ value results in a mild access skew among sites. In particular, 35% of the requests go to the most frequently accessed site while 13% of requests go to the least frequently accessed site, which demonstrates the effect that the $w_{balance}$ parameter has on transaction routing.

G FAULT TOLERANCE

DynaMast and the dynamic mastering protocol are fault tolerant. To provide fault tolerance, DynaMast ensures that transaction updates and commits are durable, that is, they persist in the presence of faults. DynaMast provides transactional durability by writing transaction updates to the Kafka log, as part of the commit. Kafka guarantees the persistence of the log by replicating it to a configurable number of followers, which correspond to the number of faults that the system can tolerate [59]. When a crash occurs, DynaMast consults the Kafka log to determine the state of the system and recover.

Similarly, to ensure remastering is fault tolerant, DynaMast writes `release` and `grant` operations to the log. Consequently, if the site selector or site manager fail, DynaMast reconstructs the data item mastership state from the sequence of `release` and `grant` messages in the Kafka log. Therefore, after a failure, the site selector reissues `grant` requests for data items that do not have a master. As an optimization, if the site managers fail (but not the site selector), then the site selector provides mastership information to the site managers directly (and vice versa).

H 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 6.2.5), 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 3 and 4.

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.