MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems

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

ABSTRACT

Distributed database systems are widely used to meet the demands of storing and managing computation-heavy workloads. To boost performance and minimize resource and data contention, these systems require selecting a *distributed physical design* that determines *where* to place data, and *which* data items to *replicate* and *partition*. Deciding on a physical design is difficult as each choice poses a trade-off in the design space, and a poor choice can significantly degrade performance. Current design decisions are typically static and cannot adapt to workload changes or are unable to combine multiple design choices such as data replication *and* data partitioning integrally. This paper presents **MorphoSys**, a distributed database system that dynamically chooses, and alters, its physical design based on the workload. MorphoSys makes integrated design decisions for all of the data partitioning, replication and placement decisions on-the-fly using a learned cost model. MorphoSys provides efficient transaction execution in the face of design changes via a novel concurrency control and update propagation scheme. Our experimental evaluation, using several benchmark workloads and state-of-the-art comparison systems, shows that MorphoSys delivers excellent system performance through effective and efficient physical designs.

PVLDB Reference Format:
Michael Abebe, Brad Glasbergen, Khuzaima Daudjee. MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems. *PVLDB*, 13(13): 3573-3587, 2020.
DOI: https://doi.org/10.14778/3424573.3424578

1. INTRODUCTION

Modern database systems store and manage vast amounts of data [39, 57] by replicating and partitioning these data to distribute their transactional processing over multiple sites (nodes). The data replication and partitioning schemes chosen for a distributed database form its *physical design*.

Constructing a distributed physical design includes making the following key decisions (i) what the data partitions should be, (ii) which data partitions to replicate, and (iii) where (at which site (machine)) to place the master (updateable) copy of a partition, and where to place any replicas (secondary) copies of partitions. These decisions, in turn, determine the sites where transactions execute.

One well-known distributed physical design [30, 58] distributes partitioned data such that transactions execute at different sites, thereby spreading both the update and read load over multiple sites in the distributed system. To achieve even load distribution, careful partitioning of the data must occur, resulting in a partitioning of the workload among the sites in the distributed system. A performance concern with transactions that access, and in particular update, data at multiple sites is that they require costly synchronization and coordination across the sites where they execute to guarantee transactional atomicity and consistency [22, 29, 36].

Another popular technique to achieve physical distribution is to replicate data to multiple sites [62, 66]. Replication allows transactions to execute on data copies or replicas. A challenge with data replication is that updates to one copy of the data cause other copies to become stale [20]. Moreover, replicas risk being inconsistent in the face of concurrent updates across the distributed system.

Placing an excessive number of data partitions (master or replica) on the same site places excessive load at that site while consuming larger amounts of storage. As in-memory databases become common to reduce latency, it is important to use storage efficiently to reduce cost-to-performance ratios [18]. Thus, dynamic data replication can use memory storage judiciously by deciding what, and where, to replicate selectively instead of (fully) replicating all data everywhere.

In addition to the transaction coordination and consistency issues that occur with data replication and partitioning, there is a need to avoid hot spots caused by workload imbalance. This imbalance can be of two types: (i) when particular data are frequently accessed causing excessive load on sites hosting the data [10, 61], and (ii) when workload hotspots shift or emerge [25, 60]. These types of load imbalances emanate from workloads contending for physical compute resources, e.g., CPU, or data resources such as locks.

Offline tools [10, 47, 48, 55, 63, 67] can help administrators make design decisions. However, these tools generate *offline, static* physical designs that cannot adapt to workload changes or are not suitable if extensive information about the workload, e.g., access patterns, is not available *a priori*. Moreover, such tools do not eliminate distributed coordination costs incurred by transactions that access data across multiple sites. Thus, static distributed physical designs fall short in delivering good performance in the presence of hotspots, changing workloads or when workload information is not available a priori to inform physical design decisions.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3424573.3424578

We present **MorphoSys**, a distributed database system that we have designed and built that makes decisions *automatically* on how to partition data, what to replicate, and where to place these partitioned and replicated data. MorphoSys learns from workload locality characteristics what the partitions should be, and where to place master and replica partitions, *dynamically.* This dynamism frees the system from having to know workload access patterns a priori, allowing MorphoSys to *change* or metamorphosize the distributed physical design on-the-fly. Moreover, to avoid expensive multi-site transaction coordination, MorphoSys dynamically alters its physical design to co-locate co-accessed data and to guarantee single-site transaction execution.

Remarkably, once MorphoSys starts executing, it requires no administrator intervention. Unlike prior approaches [53, 64], MorphoSys adapts its physical design continuously and iteratively, adjusting *both* its replication and partitioning schemes to cater to the workload. MorphoSys uses a *learned* cost model based on workload observations to decide how, and when, to alter its physical design.

The main contributions of this paper include:

- An architecture for a distributed database system that dynamically constructs physical designs using a set of physical design change operators (Section 3).
- A novel concurrency control algorithm and update propagation protocol to support efficient execution of transactions and physical design changes (Section 4).
- A cost model that drives physical design decisions to improve transaction processing performance by learning and exploiting workload patterns (Section 5).
- An extensive evaluation that establishes MorphoSys' efficacy to deliver superior performance over prior approaches. (Section 7).

2. RELATED WORK

MorphoSys is the first and only *transactional* (ACID) distributed database system that *dynamically* generates distributed physical designs that encompass all three schemes of (i) data replication, (ii) data partitioning, and (iii) master data location in an *integrated* approach. None of the related work to-date can achieve more than one of the three aforementioned schemes simultaneously in a dynamic fashion.

Dynamic replication adjusts what, and where, to replicate for a workload. The adaptive replication algorithm [64] used by non-transactional storage systems [24, 40], and key-value stores [42] optimizes replica placement given workload descriptions and operation cost estimates. These systems replicate on only a per site basis rather than on a per data item or partition basis; by contrast, MorphoSys is transactional with ACID semantics, considers the whole workload while making replication decisions for individual data partitions and guarantees single-site execution.

Dynamic partitioning systems [16,53,54,60,61] dynamically change the grouping of data into partitions to account for access skew. These systems rely on the expensive two-phase commit protocol (2PC) to coordinate multi-site transaction execution [22,29]. By contrast, MorphoSys guarantees single-site transactions by using dynamic physical design operators that change the location of both master data and their replicas. Clay [53] makes data partition decisions periodically after observing the workload over a large interval before generating a revised data partitioning scheme. MorphoSys iteratively changes partitions at the transaction-level granularity, considering every transaction as an opportunity to adapt its design to the workload.

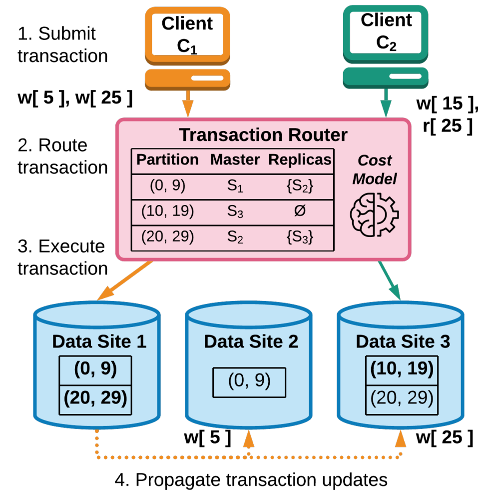


Figure 1: MorphoSys System Architecture. Transaction router selects data sites at which client transactions are executed. Replicas are lazily maintained through updates propagated from data sites.

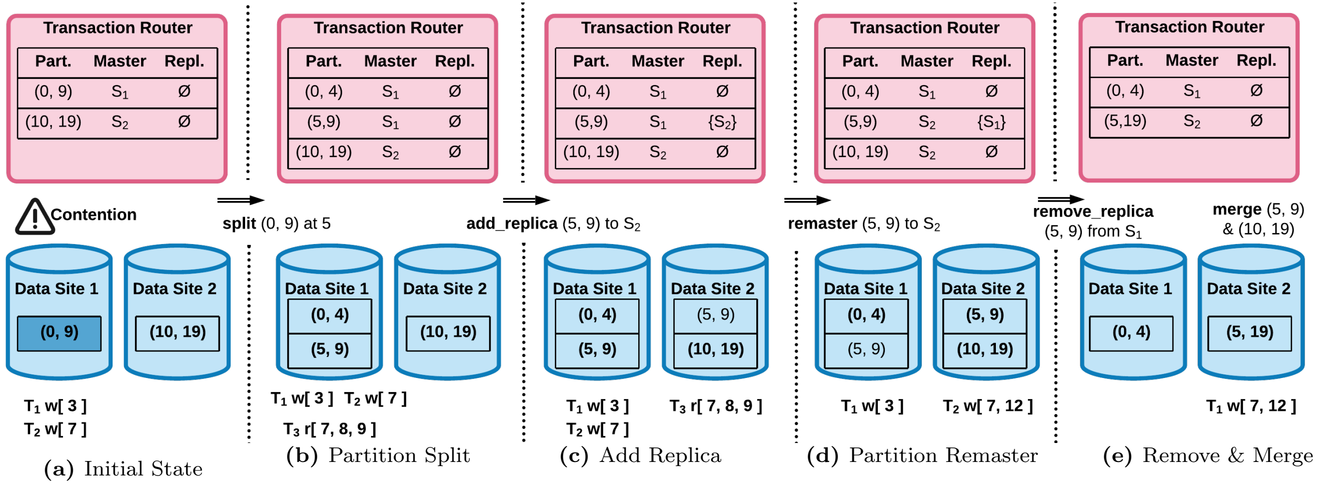
Dynamic mastering guarantees single-site transaction execution by co-locating master data at one site [3,36,38,50]. MorphoSys employs dynamic mastering for this purpose and leverages replicas to change mastership efficiently, unlike, STAR [38] and DynaMast [3] that require full data replication. All prior dynamic mastering systems require static *a priori* grouping of data into partitions while MorphoSys dynamically partitions data to mitigate the effects of contention. STAR and SLOG [50] employ batched transaction execution while MorphoSys executes transactions as they arrive to reduce latency.

Offline tools exist that recommend how to group data items together into partitions [10], what to replicate where [48], or how to assign partition mastership [10, 55, 67]. A system must then implement and apply these decisions. By contrast, MorphoSys makes all of these physical design decisions together intrinsically within the system in an online fashion, allowing it to adjust to workload changes on-the-fly.

Orthogonal database physical design decisions include index recommendation [11], automatic index generation [28, 37, 43], selecting views to materialize [21], determining the storage layout of data [4], and elastically adding database sites [59, 65]. Such decisions are complementary and can co-exist with MorphoSys' physical design space.

3. MorphoSys OVERVIEW

MorphoSys' distributed architecture consists of *data sites* that store data and execute transactions on the data. MorphoSys stores data in an OLTP optimized row-oriented format. Each row represents a data item identified by a *row id* that is either application-defined or generated from the primary key of the table and encodes any primary-key foreign-key constraints. MorphoSys groups data items together into *partitions* and each data item belongs to one partition at a time. For every partition, MorphoSys decides the site that stores the master copy of the partition and the sites, if any, that lazily replicate the partition. A change in any of these decisions results in a physical design change. MorphoSys acts on these decisions, using physical design change opera-



(a) Initial State **(b)** Partition Split **(c)** Add Replica **(d)** Partition Remaster **(e)** Remove & Merge

Figure 2: An illustration of physical design change operations in MorphoSys.

tions (Section 3.1) to dynamically change the system's data replication, partitioning and mastering schemes.

Clients submitting transactions to the database system do not need to know the data sites that store the relevant data items for their transactions, and where the transaction will execute. Instead, the *transaction router* considers the current design in routing transactions to data sites. MorphoSys uses read and write set information in transactions submitted by clients to plan and execute design changes needed for single-site execution and to improve system performance.

Figure 1 shows MorphoSys' architecture, with partitions identified as contiguous ranges of *row id*'s (e.g., $(0, 9)$), master copies indicated in bold (e.g., data site 1 masters partition $(0, 9)$), and replicas not in bold (e.g., data site 2 replicates partition $(0, 9)$). Note that MorphoSys *partially* replicates data, thereby replicating selected data partitions to selected sites. Figure 1 also provides an overview of transaction execution: client C_1 submits a transaction to update data items 5 and 25 to the transaction router (Step 1), which routes the transaction to data site 1 (Step 2) where the transaction executes (Step 3). MorphoSys lazily propagates the transaction updates to data sites 2 and 3, which contain replicas of partition $(0, 9)$ and $(20, 29)$, respectively (Step 4). MorphoSys concurrently executes client C_2's transaction at data site 3, which updates the master copy of partition $(10, 19)$ and reads from the replica of partition $(20, 29)$.

The transaction router ensures single-site transaction execution by routing transactions to a site that stores the master copies of partitions containing data items in the write set, and (master or replica) copies of partitions containing data items in the read set. If no site satisfies this requirement, then the transaction router alters the system's physical design on-the-fly by dynamically adding or removing replicas of partitions, changing a partition's granularity, or changing the mastership of partitions. Thus, the transaction router plays a key role in system performance: it must effectively decide on physical designs to distribute transactions among sites without requiring design changes before executing every transaction. MorphoSys meets these goals by capturing workload patterns and quantifying the expected benefit of design changes with a learned cost model (Section 5).

Data sites play a key role in system performance as they execute transactions and physical design changes. A data site must efficiently maintain replicas, support changing the partition a data item belongs to, and the mastership of partitions. Data sites must also ensure that transactions observe a consistent state of the database in the presence of physical design changes. To achieve these goals, we develop an update propagation scheme and concurrency control algorithm that support efficient transaction execution and physical design changes (Section 4).

3.1. Physical Design Change Operations

MorphoSys supports a variety of dynamic physical design change operations. These operations are flexible building-blocks that MorphoSys effectively combines to produce efficient distributed database physical designs.

We define a data partition as a range of data items based on their *row id*'s. Partition p contains all data items with *row id*'s in the inclusive range $(start(p), end(p))$. The master copy of p is located at site $S_i = master(p)$, with replicas on the (possibly empty) set of sites $\{S_j | j \neq i\} = replicas(p)$.

Figure 2 exemplifies our physical design change operations by example, using two data sites. Figure 2a shows the initial design, while Figures 2b–2e illustrates the series of physical design changes that culminate in a new physical design.

In the initial physical design (Figure 2a), transaction T_1 writes data item 3, and transaction T_2 writes data item 7, hence both update the partition $(0, 9)$ located on data site 1. Thus, there is physical lock contention on the partition even though the updates logically do not conflict. *Splitting* the partition, a dynamic partitioning operation, changes the data partition that data items belong to and dissipates the contention (Figure 2b) as transactions T_1 and T_2 subsequently update disjoint partitions. Formally, $split(p, k)$ creates partitions p_L and p_H that contain the data items in p with *row id*'s less than k and greater than or equal to k, respectively. That is, $p_L = (start(p), k - 1)$ and $p_H = (k, end(p))$. $split(p, k)$ also removes the original partition p from the system so that data items continue to belong to exactly one partition.

In Figure 2c, three transactions execute on data site 1, including the read-only transaction T_3, adding load to the site. As replicas service read-only transactions, dynamically replicating to *add a replica* of partition $(5, 9)$ to data site 2 allows T_3 to execute there, thereby distributing load among sites. Formally, if $S_j \neq master(p)$ and $S_j \notin replicas(p) = R$ then $add_replica(p, S_j)$ sets $replicas(p) = R \cup \{S_j\}$.

In Figure 2d transaction T_2 updates data items 7 and 12. To guarantee single-site transaction execution, MorphoSys

ensures that one data site contains the master copies of the partitions containing data items 7 and 12 — partitions $(5, 9)$ and $(10, 19)$ — by dynamically changing the location of the master of partition $(5, 9)$ from data site 1 to data site 2 via *remastering*. Single-site transaction execution ensures that transactions are not blocked waiting for distributed state during 2PC's uncertain phase that increases lock holding time while blocking local and distributed transactions [3, 22, 29, 38]. Observe that data site 2 was previously replicating partition $(5, 9)$, which ensures remastering is efficient [3, 38]. Formally, if $S_i = master(p)$, and $S_j \in replicas(p) = R$ then $remaster(p, S_j)$ sets $S_j = master(p)$ and sets $replicas(p) = R \setminus \{S_j\} \cup \{S_i\}$.

Replicas require space for storage and their maintenance consumes resources. MorphoSys dynamically *removes replicas* of partitions if there is little benefit in maintaining a replica, such as when no transactions read from the replica, as in Figure 2e for partition $(5, 9)$ at data site 1. Formally if $S_j \in replicas(p) = R$, then $remove_replica(p, S_j)$ sets $replicas(p) = R \setminus \{S_j\}$.

Finally, to reduce the metadata overhead of tracking partitions, MorphoSys *merges* co-accessed partitions together, as shown in Figure 2e for partitions $(5, 9)$ and $(10, 19)$ creating partition $(5, 19)$. Formally, if $end(p_L) = start(p_H) - 1$, $master(p_L) = master(p_H)$, and $replicas(p_L) = replicas(p_H)$ all hold, then $merge(p_L, p_H)$ creates a single partition $p = (start(p_L), end(p_H))$, and removes partitions p_L and p_H.

The examples in Figure 2 show the benefits of physical design changes. MorphoSys performs changes prudently by considering the workload to avoid unnecessary changes or changes that it will undo in the immediate future.

4. TRANSACTION EXECUTION AND PHYSICAL DESIGN CHANGE

In this section, we describe MorphoSys' novel concurrency control and update propagation mechanisms and how they provide efficient physical design changes and transaction execution. We begin by presenting the design to achieve efficient transaction execution followed by the mechanics of the physical design operators. Appendix B includes correctness proofs of MorphoSys' protocols.

4.1. Transaction Isolation and Concurrency

MorphoSys supports ACID *transactions* and provides strong-session snapshot isolation (SSSI) [13], which is a popular isolation level for distributed and replicated database systems [3, 7, 13, 56]. SSSI guarantees snapshot isolation (SI) but also prevents transaction inversion and ensures that client transactions always see all updates from their previous transactions. In SSSI, every transaction T is assigned a begin timestamp such that T sees the updates made by transactions with earlier commit timestamps. SSSI systems select begin timestamps that are larger than the latest commit timestamps of prior transactions submitted by the client to prevent transaction inversion. SSSI systems also ensure that if two transactions update the same data item and have overlapping timestamps, only one transaction can commit [6]. SSSI is layered on top of SI, which allows distinct begin and commit transaction timestamps and write skew, and thus is weaker than serializability in which all operations come at the same point in time in a serialization history [6, 13].

MorphoSys *decouples read and write operations* as these operations do not conflict under SSSI; SSSI writes create new consistent logical snapshots, based on the commit timestamp, while reads occur from logically consistent snapshots indicated by the begin timestamp. This decoupling allows concurrent execution of reads at replicas while data sites apply propagated updates. MorphoSys uses multi-versioning [14, 33] to efficiently decouple reads and writes.

All of MorphoSys' operations, including transactions and physical design change operations, occur on a *per partition* basis. That is, operations access a partition or its metadata only if accessed in a transaction's read or write set or as part of a design change operator. Per partition operations minimize contention and blocking as the system accesses only relevant partitions, and replicas wait for only the necessary updates to relevant partitions to preserve consistency.

4.2. Partition-Based Multi-version Concurrency Control

Guided by our design requirements of supporting session consistency with SI, decoupling reads and writes, and performing operations on a per partition basis, we propose a novel concurrency control algorithm. Our algorithm maintains per partition version information and a lock per partition. Any update to a partition by transactions or physical design change operators acquires the partition's lock using lock ordering to avoid deadlocks. Consequently, at most one writer to a partition executes at a time, which prevents unnecessary transactional aborts. Concurrent writes and design changes to different partitions are supported since these are conflict-free with locks held on a per partition basis. Dynamic partitioning, through *split* and *merge* operators, ameliorates contention within and across partitions. Read operations leverage multi-versioning to execute freely without acquiring partition locks.

MorphoSys uses per partition version information and tracks transactional dependencies to implement a dependency-based concurrency control protocol [44, 45], which allows MorphoSys to apply updates in parallel and on a per partition basis at replicas (Section 4.3). To track partition dependencies, each partition p maintains a *version number* $v(p)$. Multiple versions of each data item d are kept, and each version is called a *versioned data item*, consisting of a version number $v(d)$ that coincides with the version number $v(p)$ of the partition p that contains d when d was updated. When a transaction T updates data item d in a partition p, it creates a new version of the data item and its associated data. On commit, T increments the partition version number $v(p)$ and assigns that number as $v(d)$ for the new versioned data item.

In addition to storing per partition version numbers, MorphoSys maintains dependency information to ensure transactions read from a consistent snapshot of data. In particular, for version $v(p_i)$ in partition p_i, MorphoSys stores the version number of partition p_j belonging to the same logical snapshot as $v(p_i)$, which we denote as $depends(p_i, v(p_i), p_j)$.

To track dependencies, MorphoSys uses the following *dependency recording rule*. If a transaction T updates data items in partitions p_i and p_j, then when T commits MorphoSys assigns partition version numbers $v(p_i)$ and $v(p_j)$. MorphoSys then records $v(p_j)$ as $depends(p_i, v(p_i), p_j)$, and similarly $v(p_i)$ as $depends(p_j, v(p_j), p_i)$. To capture any existing transitive dependencies, MorphoSys extends the recording rule to the rule presented in Equation 1, which applies to any partition p_k that MorphoSys has not already recorded dependencies for as part of the transaction update, and partitions p_i and p_j in the write set. Equation 1 captures direct dependencies that involve partitions updated in the same

transaction, and transitive dependencies that are inherited from previous transactions. Taking the maximum of a partition's direct and inherited dependencies ensures transactions observe consistent state.

$$depends(p_i, v(p_i), p_k) = max_j(depends(p_i, v(p_i) - 1, p_k), depends(p_j, v(p_j) - 1, p_k)) \quad (1)$$

MorphoSys efficiently maintains tracked dependencies by storing them in recency order with partition metadata at a data site. As OLTP transactions typically access a small amount of data relative to the database size [15, 36], tracked dependencies are often small, while long-running transactions accessing many partitions induce more dependencies for the system to track. To ensure tracked dependencies do not grow unbounded, MorphoSys garbage collects versioned data items and dependency information of partitions with version numbers lower than a watermark version number, which is the smallest most recent version number at any site.

4.2.1. Strong Session Snapshot Isolation

To enforce SSSI, MorphoSys uses the recorded dependencies to guarantee that transactions read data from a consistent snapshot state using the *consistent read rule*: a transaction T reads versions $v(p_i)$ of every partition p in its read set such that $v(p_i) \geq depends(p_j, v(p_j), p_i)$ if p_j is also in T's read set. T reads the latest version of data item d such that $v(d) \leq v(p_i)$.

MorphoSys uses the *depends* relationship as the basis of transaction timestamps. When a transaction T intends to read partitions $\{p_r\}$ and write partitions $\{p_w\}$, T acquires write locks on each p_w. T then reads the version number for each partition $v(p)$ in its read and write set, and stores this value as $T^B(p)$. As defined by the consistent read rule, T then determines the version $T^B(p_i)$ of each partition such that $T^B(p_i) \geq depends(p_j, T^B(p_j), p_i)$ for p_i and p_j in T's read and write set. The values $\{p, T^B(p)\}$ form T's begin timestamp T^B. T then reads and writes transactionally.

On commit, MorphoSys generates a commit timestamp T^C by incrementing $v(p_w)$ for each p_w in T's write set and follows the dependency recording rule. Mutual exclusion of partition writes ensures that the updated version $v(p_w) = T^B(p_w) + 1$, which is assigned to $T^C(p_w)$, and along with $T^B(p_r)$ for each read partition forms the commit timestamp.

Appendix A includes more details on our concurrency control protocol, which ensures that long forks [56] are not allowed under SSSI, and describes how session timestamps prevent transaction inversions.

4.3. Update Propagation

To support dynamic replication of lazily maintained partitions, MorphoSys propagates and applies updates on a per partition basis. If a transaction T updates a partition p then the data site maintains a redo buffer of T's changes. When T commits, for each updated partition p, the data site serializes the changes made to p, consisting of the updated data items, p's version number ($v(p)$), and the recorded dependencies (T^C). The data site writes this serialized update to a per partition *redo log*. Data sites replicating p subscribe to the partition's log, asynchronously receive the serialized update, and apply the update as a *refresh transaction*.

Replicas ensure a snapshot consistent state by applying T's refresh transaction to partition p only after applying all previous updates to the partition, based on the partition version number. The refresh transaction installs T's updates to p by creating new versioned data items and makes the update visible by incrementing p's version number.

MorphoSys' design choices reduce the overhead of maintaining replicas because: (i) multi-versioning allows concurrent execution of read-only transactions and refresh transactions on replica partitions, and (ii) tracking transaction dependencies allows per partition execution of refresh transactions, which eliminates blocking on updates to other partitions.

4.4. Physical Design Change Execution

To execute the five physical design change operators efficiently, we integrate them with the concurrency control algorithm and the update propagation protocol.

4.4.1. Adding and Removing Replicas

MorphoSys leverages multi-versioning to efficiently add a partition replica by snapshotting the partition's master state without acquiring locks. A data site snapshots a partition p by reading the last written position in its redo log, the version number of the partition ($v(p)$), p's dependency information, and a snapshot of all data items in the partition at version $v(p)$. At the newly created replica, the data site installs this snapshot of the partition, records the version number, and subscribes to updates to the partition beginning from the last known position in the redo log. At this point, the data site continues to apply propagated updates.

A data site removes a replica of a partition by stopping its subscription to the partition's updates. The data site then deletes the partition structure so future transactions do not access the partition. MorphoSys uses reference-counted data structures [27] to access partitions, which ensures that ongoing transactions can read from the removed partition replica without blocking the physical design change operation.

4.4.2. Splitting and Merging Partitions

MorphoSys' partition-based concurrency control executes the splitting and merging of partitions as transactions. These transactions update only partitions' metadata and not any data items. Hence, when splitting a partition p to create new partitions p_L and p_H, the data site mastering p acquires a partition lock on p. Thus, no other updates to p can take place during the split. Then, the data site creates new partitions p_L and p_H and assigns them both a version number equal to the version number of the original partition $v(p)$. The data site logically copies the reference-counted data items from p to the corresponding new partition, p_L or p_H. Committing the split operation induces a dependency between the three partitions (p, p_L, p_H) and results in p_L and p_H receiving p's dependency information, which ensures that transactional accesses to the new partitions observe a consistent state. On commit, the data site removes p. The bidirectional tracking of the *depends* relationship among partitions allows MorphoSys to split partitions without updating other partitions' state, thus reducing overhead.

Data sites replicating partition p observe the split operation in the redo log. Replicas apply the split as a refresh transaction as outlined at the master data site, subscribe to updates of the newly created partitions p_L and p_H and remove their subscription to p. While a split is in progress, ongoing read transactions can access p but all subsequent transactions run on the newly created p_L and p_H.

Data sites merge partitions by following the reverse of splitting a partition, with two differences. First, when merging two partitions p_L and p_H to create partition p, the data

site assigns the version of p as the maximum of p_L's and p_H's version numbers. Second, at a replica, the merge operation does not complete until both p_L and p_H are at the same state (as indicated by version numbers) as when the merge occurred at the master site.

4.4.3. Changing Partition Mastership

MorphoSys changes the mastership of a partition from site S_i to S_j as a metadata-only operation via update propagation of the mastership change information to all partition replicas. The old master site S_i executes the request to release the mastership of a partition p as an update transaction. This transaction changes the status of the partition to a replica, and writes an update containing the mastership change information to the partition's redo log. Afterwards, no update transactions to partition p can execute at Site S_i. Site S_j does not become the master of the partition until it applies the propagated update from the old master site S, and hence all prior updates to the partition.

MorphoSys' concurrency control and update propagation protocol reduce blocking times when changing mastership as the new master waits only for the partition undergoing remastering to reach the correct state. By contrast, prior approaches [3, 38] require all data items in the system to reach the same, or later, state as the prior master.

5. PHYSICAL DESIGN STRATEGIES

As a workload executes, MorphoSys automatically decides: (i) how to alter its physical design with the aforementioned operators, and (ii) when to do so. The transaction router makes these decisions by quantifying the expected benefit of the feasible design changes and greedily executing the set of changes with the greatest expected benefit. We quantify the benefit of a physical design change by modelling the effect that a design change will have on the future workload as well as the cost of performing the change. The transaction router uses a workload model (Section 5.1) and learned costs of transactions and physical design changes (Section 5.2) to make its design decisions (Section 5.3).

5.1. Workload Model

MorphoSys continuously captures and models the transactional workload to make design decisions. The transaction router samples submitted transactions and captures data item read and write access frequencies. For a partition p, the transaction router maintains the probability of reads $R(p)$, and writes $W(p)$, to the partition compared to all partition accesses. The transaction router computes $R(p)$ (or $W(p)$) by dividing the per partition read (write) count by the running count of all reads and writes to all partitions.

As data item accesses are often correlated [3, 10, 53], MorphoSys tracks data item co-access likelihood. MorphoSys leverages these statistics to determine the effect of a proposed physical design change on future transactions (Section 5.3). Formally, $P(r(p_2)|r(p_1))$ represents the probability that a transaction reads partition p_2 given the transaction also reads p_1. We define similar statistics for write-write ($P(w(p_2)|w(p_1))$), read-write ($P(r(p_2)|w(p_1))$), and write-read ($P(w(p_2)|r(p_1))$) co-accesses. MorphoSys adapts its model to changing workloads using adaptable damped reservoir sampling [5]. The reservoir determines sampling of transactions, generation of their access statistics and expiration of samples based on a configurable time window. MorphoSys uses statistics from these transactions in the reservoir to adapt its design to workload changes.

5.2. Learned Cost Model

Given a physical design, the transaction router estimates the costs of executing transactions and applying physical design changes using a learned cost model. This cost model predicts the latency of design change and transaction execution operations. MorphoSys' implementation of these operations (Section 4) translates to a natural system latency decomposition: *waiting for service* at a site, *waiting for updates*, *acquiring locks*, *reading and writing* data items, and *commit*. Thus, we decompose the cost model into five corresponding cost functions (Table 1) that MorphoSys learns and combines to predict the latency of operations.

The transaction router learns these cost functions continuously using linear regression models because they are easy to interpret, do not require large amounts of training data, and are efficient for both inference and model updates [19]. Our linear regression models ($\mathcal{F}$) consume a vector of numeric inputs (x_i) and output a scalar prediction (y) by taking the sum of the inputs, each scaled by a learned weight (ω_i). In general, we favour cost functions with few input parameters as such functions tend to be robust and accurate (Section 7.3.6). The transaction router continuously updates the weights in the learned cost functions based on its predictions and corresponding observed latencies.

Next, we discuss the five cost functions in Table 1 and how we combine them to predict latency of operations. Section 5.3 details how we use the cost functions to compute the expected benefit of different physical designs.

5.2.1. Waiting for Service

The load at a site determines how quickly the site services a request, with a site under heavy load taking longer. As MorphoSys is an in-memory system, and thus CPU bound [23, 35], we model data site load based on average CPU utilization, which the transaction router polls for regularly. Hence, we model the service time at a data site S as $\mathcal{F}_{service}(c_s = cpu_util(S))$. To compute the service time, we subtract the latency of the operation measured at the data site from the observed latency at the transaction router. Hence, network latency is captured in the service time. All operations take into account the service time at a site. In the case of operations that involve multiple sites (*remaster*, *add_replica*), we compute $\mathcal{F}_{service}$ for each involved site.

5.2.2. Waiting for Updates

Recall that remastering does not complete until all necessary updates have been applied to the partition being remastered (Section 4.4). Similarly, transactions may wait when reading a replica partition to observe a consistent state. The waiting time depends on the number of propagated updates needed and the relative frequency of those updates, both of which implicitly include an estimate of the network latency. We model the waiting time at a partition p as $\mathcal{F}_{wait_updates}(v_n, v_r, u_s)$, where v_n is the version of the partition required at the replica, v_r is the current version of the replica partition, and u_s is the fraction of updates applied at the replica S relevant to partition p. We determine u_s using Equation 2; the denominator represents the total likelihood of updates to partitions that have replicas at S, which MorphoSys aggregates.

$$u_s = \frac{W(p)}{\sum_{p_i : S \in replicas(p)} W(p_i)} \tag{2}$$

Table 1: Cost functions and their use in predicting costs for transactions and physical design change operators.

Cost Function	Arguments	Operations					
		transaction	*remaster*	*split*	*merge*	*add_replica*	*remove_replica*
$\mathcal{F}_{service}(c_s)$	c_s = CPU utilization of site S	✓	✓	✓	✓	✓	✓
$\mathcal{F}_{wait_updates}(v_n, v_r, u_s)$	v_n = partition version necessary v_r = version of partition at replica u_s = fraction of updates to partition compared to all replicas at site S	✓ (If reads)	✓				
$\mathcal{F}_{lock}(w)$	w = write contention of partitions	✓ (If writes)	✓	✓	✓		
$\mathcal{F}_{read_write}(r_d, w_d)$	r_d (w_d) the number of data items read (written)	✓				✓	
$\mathcal{F}_{commit}(r_p, w_p)$	r_p (w_p) the number of partitions read (written)	✓	✓	✓	✓		

We use the session timestamp to determine v_n for transactions while remastering uses the latest polled version from the old master. As data sites apply updates in parallel that a transaction may require, we consider the largest predicted $\mathcal{F}_{wait_updates}$ from all partitions in the read set.

5.2.3. *Acquiring Locks*

Recall from Section 4.2 that update operations acquire per partition locks. Hence, increased write contention on a partition results in longer partition lock acquisition time. Thus, we predict lock acquisition time as $\mathcal{F}_{lock}(w)$ where w represents write contention. MorphoSys estimates w using $W(p)$, the probability of writes to a partition. For operations that access multiple partitions such as transactions or partition merging, data sites acquire locks sequentially. We define $w = \sum_p W(p)$ for all relevant p, i.e., partitions p_L, p_H for *merge*, and all partitions in a transaction's write set.

5.2.4. *Reading and Writing Data Items*

The primary function of transactions is to read/write data. Hence, we estimate the time spent reading and writing data items using the cost function $\mathcal{F}_{read_write}(r_d, w_d)$, where r_d and w_d represent the number of data items read and written, respectively. The transaction router determines r_d and w_d from a transaction's read and write set. Adding a replica of a partition requires reading a snapshot of the partition and installing it into the newly created replica. Hence when estimating the latency of adding a replica, we also consider $\mathcal{F}_{read_write}$ using the number of data items in the partition as the number of data items read and written.

5.2.5. *Commit*

Data sites commit operations by updating partition version numbers and dependency information (Section 4). The number of updates depends on the number of partitions involved in the operation. We use the number of partitions read (r_p) and written (w_p) by the operation to estimate the commit time as $\mathcal{F}_{commit}(r_p, w_p)$. Given that adding and removing a replica of a partition does not entail committing, we omit commit latency for these operations.

5.2.6. *Putting it Together*

The transaction router predicts the latency of each operation by summing the values of the cost functions shown in Table 1 that are associated with the operation. For example, to predict the latency of a *split* at a site, MorphoSys uses the recorded CPU utilization of the site (c_s) and write contention of the partition (w) to compute $\mathcal{F}_{service}(c_s) + \mathcal{F}_{lock}(w) + \mathcal{F}_{commit}(0, 3)$. The system tracks the latency of the entire split operation, as well as the portion of the time spent in the network waiting for service, locking and committing. Given these observed latencies and latency predictions (e.g. $\mathcal{F}_{service}(c_s)$), the transaction router uses stochastic gradient descent [52] to update its cost-functions.

5.3. Physical Design Change Decisions

MorphoSys makes physical design and routing decisions to improve system performance and to execute transactions at one site. The transaction router considers each data site as a candidate for transaction execution and develops a physical design change *plan* for the site. A site's plan consists of a set of physical design change operators that allow the transaction to execute at the site, including adding or removing replicas, splitting or merging partitions, and remastering. For a data site S, the transaction router computes the cost of executing the plan, $C(S)$, and expected benefit, $E(S)$. The transaction router selects the data site, and plan, that maximize the net benefit $net(S) = \lambda \cdot E(S) - C(S)$. The magnitude of $\lambda(> 0)$ controls the relative importance of the expected benefit of the plan. We will use the *split* operation of partition $(0, 9)$ from Figure 2b as a running example in this section to illustrate MorphoSys' decisions.

The cost of executing a plan $C(S)$ is the sum of the execution costs of each of the plan's operators, as defined in Section 5.2. A low execution cost indicates that the plan will execute quickly. The input parameters for these costs correspond to the statistics from the existing physical design. In the *split* operator example, MorphoSys estimates $C(S)$ by computing $\mathcal{F}_{service}(c_1) + \mathcal{F}_{lock}(W_{(0,9)}) + \mathcal{F}_{commit}(0, 3)$, where c_1 represents the CPU utilization at data site 1 and $W_{(0,9)}$ represents contention of the partition being split.

To determine the expected benefit $E(S)$, we predict the latency of transactions under the current physical design and subtract the predicted latency of transactions under the physical design that would result from executing the plan. A good plan decreases predicted latencies, which increases $E(S)$ and thus increases $net(S)$. To determine $E(S)$, we sample transactions (Section 5.3.1) and use our learned cost model with input parameters that correspond to the plan's new physical design of the database (Section 5.3.2). Consider the effect splitting partition $(0, 9)$ has on transactions T_1 and T_2 from Figure 2b. First, we predict the latency of T_1 and T_2 executing at site 1 under the current physical design. As both transactions update partition $(0, 9)$, we estimate $\mathcal{F}_{lock}(W_{(0,9)})$ as part of the transaction latency, which we call $L_{current}$. We then estimate the effect of splitting the partition into $(0, 5)$ and $(6, 10)$, each of which has less write contention than partition $(0, 9)$. Then, we use the contention of partitions $(0, 5)$ and $(6, 10)$ to estimate the latency of transactions T_1 and T_2 on the future design, which we call L_{future}. Finally, we compute $E(S)$ as $L_{current} - L_{future}$. If $E(S)$ is positive, then the design change is predicted to reduce

latency and improve system performance. Design changes with non-positive $E(S)$ values are unlikely to improve system performance and are thus avoided.

5.3.1. Sampling Transactions

Computing the expected benefit of a design change plan ideally requires knowledge of future transactions to be submitted to the system, which is not available. Thus, MorphoSys draws samples of transactions from its reservoir of previously submitted transactions to emulate a workload of future transactions, for example T_1 and T_2, from our running example. MorphoSys also generates emulated transactions based on its workload model to ensure robustness in design decisions. To generate these transactions, we select a partition p_1 at random following the partition access frequency distribution. Then, we sample a second partition p_2 co-accessed with p_1 and generate four transactions that access p_1 and p_2, based on all combinations of read and write co-accesses, and weight any expected benefit by the likelihood of co-access.[1] If we generate a transaction T that reads p_1 and updates p_2, then we weigh the expected benefit to T by $R(p_1) \times P(w(p_2)|r(p_1))$, when computing $E(S)$. We select data item accesses to the partition uniformly at random.

5.3.2. Adjusting Cost Model Inputs

The transaction router predicts the latency effect of physical design changes by predicting changes to inputs of its cost model. This is done by considering how design changes affect CPU utilization, update application rate, and contention.

Recall that $\mathcal{F}_{service}(c)$ predicts the time spent waiting for a data site to service a request. Data sites use resources to perform database reads, writes, and apply propagated updates. MorphoSys predicts CPU utilization based on the frequency of reads (r_S), writes (w_S), and propagated updates applied (u_S) using the cost function $\mathcal{F}_{CPU}(r_S, w_S, u_S)$. Remastering a partition p, and adding or removing partition replicas affect r_S, w_S and u_S based on the probability of reads and writes to p. We use the output of $\mathcal{F}_{CPU}$ as input to $\mathcal{F}_{service}$ when predicting the time spent waiting for a data site to service a transaction. By considering $\mathcal{F}_{service}$, MorphoSys favors designs that distribute the load to all data sites, which minimizes wait time.

Splitting a partition reduces the probability of reads and writes to the newly created partitions, which store fewer data items. The reverse holds for merging partitions. To reduce tracking overheads, we assume uniform accesses to data items within a partition when modelling the effects of a design change on write contention (w in $\mathcal{F}_{lock}$), and update frequency (f_S in $\mathcal{F}_{wait_updates}$). If the design change occurs, then over time, the partition statistics reflect the partition access likelihood. Partition splits and merges also change the number of partitions accessed by a transaction, which we consider when predicting commit latency ($\mathcal{F}_{commit}$).

A physical design change may enable future transactions to execute at a single site without further design changes. We encourage such design changes as they account for data locality in the workload by incorporating the expected benefit of not needing future design changes. To do so, we predict the latency of previously required physical design changes, which the plan saves, and add these savings to the expected benefit. Conversely, we discourage plans that induce designs precluding single-site transactions by subtracting the predicted latency of future physical design changes from the plan's expected benefit. Hence, MorphoSys avoids generating design changes that it could shortly undo.

5.3.3. Generating Plans

The transaction router generates a physical design change plan for a site by adding operations necessary for single-site transactions: remastering and adding replicas of partitions in the write and read sets, or removing replica partitions to satisfy space constraints. The transaction router then adds further beneficial design changes to the plan: splitting or merging partitions, and remastering or adding replicas of partitions co-accessed with written and read partitions. The partition split in Figure 2b is an example of a beneficial design change, while the partition remaster in Figure 2d is necessary for single-site execution.

The transaction router considers partitions in the transaction's read or write set as candidates for splitting or merging. If a partition in the read or write set has above-average access probability, indicating contention on the partition, and the split is beneficial then the transaction router adds the partition split to the plan. Conversely, merging infrequently accessed partitions reduces the number of partitions in the system, which reduces metadata overheads. Thus, if a partition in the read or write set, and one of its neighbouring partitions, have below-average access likelihood and it is possible and beneficial to merge the two partitions, then we add the merge operation to the plan. Considering access frequencies ensures that MorphoSys does not undo the effects of a split with a merge, or vice versa, in the immediate future unless the access pattern changes significantly.

When a plan for a site includes remastering, addition or removal of a partitions replica, MorphoSys *piggybacks* other design change operations for correlated partitions. Piggybacking operations promotes data locality and future single-site transactions. For example, when remastering partition p_1, MorphoSys tries to piggyback the remastering of a partition p_2 frequently co-written with p_1, which occurs when both of $P(w(p_1)|w(p_2))$ and $P(w(p_2)|w(p_1))$ are high. This probability-driven remastering with piggybacking amortizes partition remastering cost while promoting co-location of co-accessed partitions. Similarly, if remastering p_1, MorphoSys will try to piggyback replica addition of a frequently read partition p_2 if $P(w(p_1)|r(p_2))$ and $P(r(p_2)|w(p_1))$ are high.

MorphoSys removes replica partitions when they are no longer beneficial to maintain or to satisfy memory constraints. If a data site is within 5% of its memory limit, then the transaction router removes replica partitions by computing the expected benefit of removing a partition and iteratively removes partitions with the least expected benefit first until the memory constraint is satisfied.

MorphoSys' generated plans use our cost and workload models to produce design changes that reduce contention, encourage data locality and single-site transactions, and maximize expected benefit when compared to execution costs — all without immediately undoing or redoing changes.

6. THE MORPHOSYS SYSTEM

We implemented MorphoSys following the architecture of Figure 1. Our implementation includes (from Sections 4 and 5) the concurrency control protocol, update propagation scheme, physical design change operators, workload model, learned cost model, and physical design strategies.

---

[1] p_1 and p_2 can represent the same partition if a transaction accesses the same partition more than once.

MorphoSys uses Apache Kafka [32] as the redo log, and cooperatively applies propagated updates at data sites. Application of updates is shared between the thread receiving updates from the Kafka redo log and transaction execution threads waiting for specific partition versions.

The transaction router tracks the state of each data partition in per table concurrent hash-table structures for efficient lookups. This state contains a partitions range of data items, the location of its master copy, its replica locations, and partition access frequencies. The transaction router updates partition state upon the successful completion of a design change. To minimize latency, the transaction router executes design change plan operations in parallel.

MorphoSys uses its redo logs to guarantee fault-tolerance. Data sites write all transaction updates and physical design changes to the log on commit. A data site recovers by consulting an existing replica or stored checkpoint of data and replaying redo logs from the last known position of the log associated with that partitions state. The transaction router recovers by checking data sites to determine their data partitions and whether they are master or replica copies.

7. EXPERIMENTAL EVALUATION

We now present our experimental results that show that MorphoSys' ability to dynamically change the physical design of a database significantly improves system performance.

7.1. Evaluated Systems

We evaluated MorphoSys against five alternative distributed database systems that employ state-of-the-art dynamic, or popular static, physical designs. We implemented these comparative systems in MorphoSys using strong-session snapshot isolation and multi-version concurrency control to ensure an apples-to-apples comparison. Recall that MorphoSys is designed to deliver superior performance irrespective of its initial physical design (Appendix D.2). Thus, in each experiment, MorphoSys starts with an initial physical design containing *no replicas* and a *randomized* master placement of partitions, an *unknown* workload model, and *must learn* its cost model from scratch. By contrast, as described next, we advantage MorphoSys' competitors by using *a priori* knowledge of the workload to optimize their initial physical design.

Clay dynamically partitions data based on access frequency to balance load [53]. Clay performs this repartitioning periodically, with a default period of 10 seconds. Unlike MorphoSys, Clay does not replicate data and uses 2PC to execute transactions that access data mastered at multiple sites [53]. Clay begins with an initial master placement so that each site masters the same number of data items, such as by warehouse in TPC-C [53]. **Adaptive Replication (ADR)** is a widely used algorithm to dynamically determine what data to replicate, and where [24, 40, 42, 64]. Like Clay, ADR uses 2PC for multi-site transactions. We advantage ADR with offline master partition placement using Schism [10]. Schism is a state-of-the-art offline tool that uses a workload's data access patterns to generate partitioning and placement of master copies of data items and their replicas such that distributed transactions are minimized while distributing load. **Single-Master** is a popular fully and lazily replicated database architecture [2, 8, 12, 31, 66] that statically places all master copies of partitions at a single master site. All updates execute at the single-master site, while replicas execute read-only transactions. **Multi-Master** is a common fully-replicated database architecture [8, 62, 66] that uses Schism's master partition placement, and 2PC to execute update transactions accessing data mastered at multiple sites. **DynaMast** is a fully-replicated database that dynamically remasters data items to guarantee single-site transactions [3]. Unlike MorphoSys, DynaMast does not dynamically replicate or partition data. DynaMast begins each experiment with the same starting master placement as Clay in which masters are uniformly placed among sites. Finally, we compare MorphoSys with the off-the-shelf commercial version of the partitioned, database system **VoltDB** that uses a static physical design [58]. We favour VoltDB using Schism's initial design and follow VoltDB's benchmarking configuration [51] that reduces durability to optimize performance by disabling synchronous commit to reduce commit latency and delaying snapshots to reduce overheads. VoltDB is also configured to use the maximum available hardware resources on each node as well as the optimized executable.

7.2. Benchmark Workloads

Our experiments execute on up to 16 data sites each having 12-cores and 32 GB of RAM, as well as a transaction router machine, and two machines that run Apache Kafka. A 10 Gbps network connects the machines. All results are averages of at least five, 5-minute OLTPBench [15] runs with 95% confidence intervals shown as bars around the means.

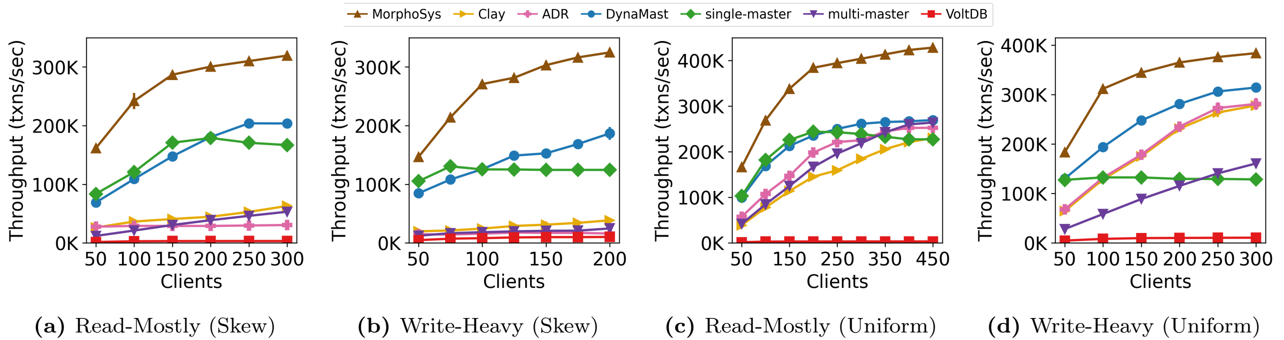
We conduct experiments using the YCSB, TPC-C, Twitter and SmallBank benchmark workloads [1,9,15] as they contain multi-data item transactions and access correlations representative of real-world workloads. In **YCSB**, we use two transactions: *multi-key read-modify-write* (RMW), which reads and updates three keys, and *scan*, which reads 200 to 1000 sequentially ordered keys. In YCSB we use skewed (Zipfian) and uniform access patterns, and 50 million data items. The industry standard **TPC-C** models a complex order-entry transactional workload. Our evaluation uses two update intensive transactions (*New-Order* (45%), *Payment* (45%)) and the read-only *Stock-Level* (10%) transaction. **Twitter** models a social networking application, featuring heavily skewed many-to-many relationships among users, their tweets, and followers. Consequently, Twitter's predominantly read-intensive workload (89%) contains transactions with complex accesses spread across the social graph data. **SmallBank** models a banking application, with 1 million bank accounts and five standard transactions that are a mix of multi-data item updates (40%), multi-data item reads (15%) and single data item updates (45%).

7.3. Results

We now discuss MorphoSys' experimental results for varying access patterns, load, and read-write mixes.

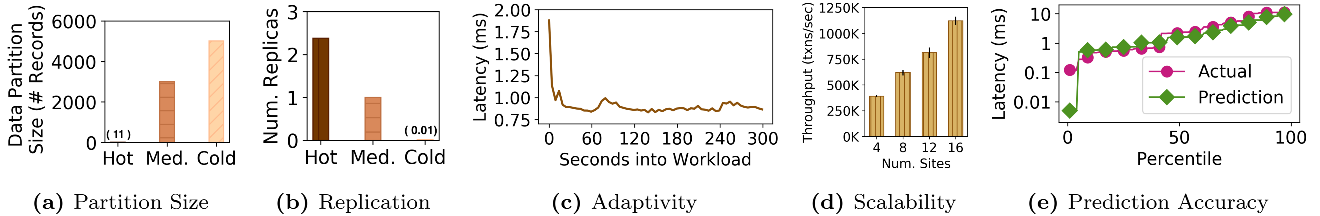
7.3.1. Workloads With Skew

Skewed workloads generate contention and load imbalance, both of which MorphoSys mitigates using dynamic physical design changes. To evaluate MorphoSys' effectiveness under skewed data accesses, we used YCSB workloads with read-mostly (50% scans, 50% multi-key RMW) and write-heavy (10% scans, 90% multi-key RMW) transaction mixes with Zipfian access skew. Throughput results for both workloads (Figures 3a and 3b) demonstrate that MorphoSys delivers significantly better performance, about 98× to 1.75× higher throughput than the other systems as it dissipates contention and balances load by dynamically repartitioning heavily accessed and contended data items into smaller partitions.



(a) Read-Mostly (Skew) **(b)** Write-Heavy (Skew) **(c)** Read-Mostly (Uniform) **(d)** Write-Heavy (Uniform)

Figure 3: Performance results for YCSB, showing throughput as the number of clients increases.



(a) Partition Size **(b)** Replication **(c)** Adaptivity **(d)** Scalability **(e)** Prediction Accuracy

Figure 4: Performance results showing MorphoSys's design decisions, scalability, adaptivity and prediction accuracy.

In Figure 4a, we classify data items as hot if they are in the top 10% of the most frequently accessed data, medium if in the top 30%, and the remaining as cold data. As Figure 4a shows, on average, MorphoSys groups the hottest data items into small partitions containing up to 100 data items. By contrast, MorphoSys groups infrequently accessed or cold data items into partitions that, on average, contain 5000 records. Such dynamic partitioning of data reduces lock acquisition time by more than a factor of 7, from nearly 850 μs to 110 μs compared to the other systems.

In addition to dynamic partitioning, MorphoSys employs dynamic replication. Figure 4b shows the replication factor for data items with different access frequencies. In contrast to the fully replicated DynaMast, single-master and multi-master architectures, MorphoSys replicates frequently accessed data items but avoids replicating infrequently accessed data. By keeping, and maintaining, fewer replicated data items, MorphoSys uses less compute resources to apply propagated updates, thereby freeing up resources to improve throughput by 2.6× over the single-master architecture in the write-heavy workload (Figure 3b). Although multi-master fully replicates data, and ADR dynamically adds replicas of frequently read partitions, they both suffer heavily from the compounding effects of contention as a result of static partitioning and distributed transaction coordination. MorphoSys combines both dynamic replication and partitioning while guaranteeing single-site transactions resulting in throughput improvements of up to 13× over ADR and multi-master.

Both Clay and MorphoSys dynamically partition to mitigate the effects of contention. However, unlike Clay, MorphoSys replicates hot partitions frequently to distribute read load among sites and make remastering more efficient. MorphoSys groups together colder co-accessed data items to reduce the metadata needed to track partitions, dependencies, and version histories. MorphoSys converges to its final design faster than Clay as MorphoSys uses every transaction as an opportunity to make design changes while guaranteeing single-site execution, in contrast to Clay's periodic operation and use of expensive 2PC. Thus, MorphoSys improves throughput over Clay by 8.5× and 5× for the update-intensive (Figure 3b) and read-heavy (Figure 3a) workloads, respectively. The scan-heavy workload exacerbates the effects of distributed reads in VoltDB as it requires enqueueing the scan operator on every site blocking all other transactions from executing due to VoltDB's single-threaded execution model [51, 58]. VoltDB's static design cannot adapt to heavy skew effects, unlike MorphoSys that improves throughput over VoltDB by almost 100×.

By taking a holistic approach to dynamic physical design and considering all 3 factors, namely, partitioning, replication and mastering, MorphoSys outperforms its competitors that consider only one of these multiple aspects of design.

7.3.2. *Workloads with Uniform Access Patterns*

We next evaluate MorphoSys in the presence of a uniform access pattern. In this workload, MorphoSys groups data items into partitions containing approximately 3000 data items, and on average replicates these partitions to one replica. This partial replication reduces the computational resources needed to maintain replicas when compared to fully replicated systems (single-master, multi-master and DynaMast). Hence, in the read-mostly case (Figure 3c), MorphoSys improves throughput over fully replicated systems in the range 1.85× to 1.6×.

A replica partition in MorphoSys supports read transactions and flexibly provides mastership opportunity when deciding on master placement. MorphoSys uses this flexibility to guarantee single-site transactions and to judiciously amortize the cost of design changes. As such, MorphoSys eliminates costly distributed coordination that Clay, ADR, VoltDB, and multi-master continually incur.

Clay initiates data repartitioning only when it detects an imbalance in partition accesses. In this workload, Clay rarely detects any imbalance in accesses, and thus rarely repartitions data. Hence, Clay must continually execute costly multi-site transactions, which in the case of scans, are susceptible to stragglers. By contrast, MorphoSys performs physical design changes as transactions execute, and co-locates co-accessed data together via dynamic replication and mastering to execute single-site transactions.

ADR dynamically adds replicas, which allows for efficient single-site scan execution, thus improving performance over Clay but falls short of MorphoSys. By considering master placement, replication, and data partitioning, MorphoSys improves throughput over ADR and Clay by almost 1.9×.

Next, we stress the systems' ability to balance update load among data sites using a write-heavy YCSB workload (Figure 3d). MorphoSys achieves even load distribution among sites in the distributed system by predicting the time spent waiting for service at a site, which is primarily determined by site load (Section 5.2). Evenly distributing load improves throughput by 3× as compared to single-master, which executes all updates at one master site. By contrast, multi-master must rely on Schism's offline analysis to ensure even routing of requests among all sites. Despite balancing the load, multi-master fully replicates data and requires distributed transaction coordination; thus, MorphoSys improves throughput over multi-master by 2.4×. ADR does not frequently replicate in this write-heavy workload and improves performance compared to multi-master, but MorphoSys remains unmatched outperforming ADR by 1.35×. The shorter read-modify-write transactions improve VoltDB's throughput compared to the scan-heavy workload. However, like ADR, VoltDB must coordinate transactions with 2PC, hence MorphoSys improves throughput over VoltDB by over 38×.

Clay and DynaMast both aim to balance update load, but come with significant shortcomings that MorphoSys addresses. As in the read-mostly case, MorphoSys' dynamic and partial replication of data reduces the overhead of maintaining replicas that DynaMast incurs. While doing so, MorphoSys still ensures the flexibility necessary to support dynamic mastership placement. Clay makes a static replication decision of never to replicate, but suffers as it does not reduce distributed transaction coordination through effective master placement. In this update-intensive uniform workload, MorphoSys' comprehensive physical design strategies and efficient execution result in 1.4× and 1.2× throughput improvement over Clay and DynaMast, respectively.

7.3.3. *Workloads with Complex Transactions*

We focus our evaluation now on TPC-C, a workload that contains complex transactions simulating an order-entry application. This workload features correlated data accesses to warehouses and districts. However, MorphoSys has no knowledge of this pattern and must learn this workload model and create an effective distributed physical design. Figure 5a shows that MorphoSys has the lowest average latency of transactions in the TPC-C workload, reducing latency by between 99% and 50% over its competitors. MorphoSys significantly reduces tail latency, as shown in Figure 5b, with reductions ranging from 167× to 6×.

To understand why MorphoSys reduces latency, we examined the physical design decisions made by MorphoSys as well as the core properties of the *New-Order* transaction that has the highest latency of all TPC-C transactions. The *New-Order* transaction reads the warehouse data item to determine the purchase tax and updates the district with the next order identifier. Schism determines that the best master placement is based on the warehouse identifier of the district and customer as 90% of *New-Order* transactions access data local to the customer's warehouse. The remaining 10% execute *cross-warehouse* transactions.

Examining the physical design decisions made by MorphoSys reveals that it creates single data item partitions for the warehouse and district tables, and replicates these partitions to multiple sites. This data partitioning supports parallel execution of *New-Order* transactions on different districts, while replication ensures efficient dynamic mastering for cross-warehouse transactions. Additionally, we observed that MorphoSys heavily replicates the read-only items table as its cost model correctly predicts that there is little overhead required to maintain this table. MorphoSys selectively replicates the remaining tables including the frequently updated customer and stock tables. For these tables, MorphoSys adds and removes replicas of partitions to balance system load and ensure single-site transaction execution.

The TPC-C results show the benefit of MorphoSys' comprehensive physical design strategies. MorphoSys' dynamic formation of data partitions produces per district partitions that reduces contention when assigning the next order identifier and decreases *New-Order* transaction latency. Increasing the scale factor which controls the number of warehouses while maintaining constant contention allows for increased load, which improves throughput. However, increasing the scale factor increases the amount of data stored in the system. Partial and dynamic replication allow MorphoSys to selectively replicate and store more data than fully replicated systems. Thus, MorphoSys supports a higher scale factor and improves throughput over the fully replicated systems by between 48× to 5.5× (Figure 5c). By considering master placement and guaranteeing single-site transactions, MorphoSys improves throughput over systems that require distributed transactions by 900× to 48×.

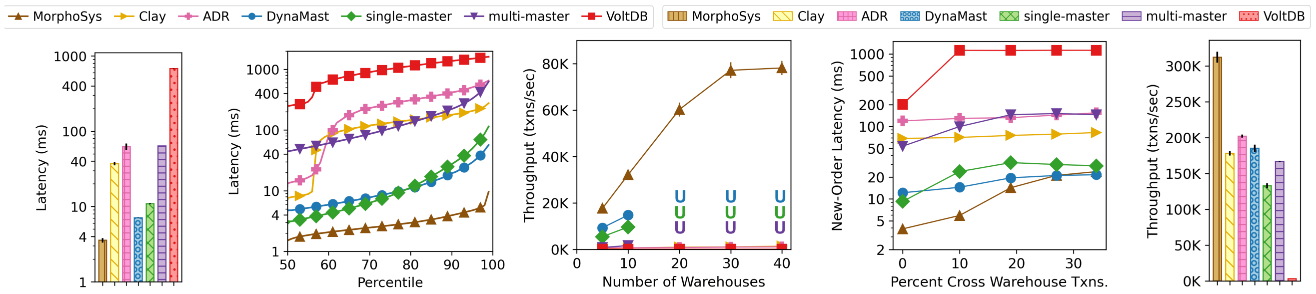
In Figures 6a and 6b, we show throughput and tail latency for the Twitter workload. MorphoSys achieves between 32× and 3× greater throughput than its competitors and reduces tail latencies by between 58× and 4×. MorphoSys' tail latency reductions primarily come from reducing the time spent waiting for updates to tweets in the *GetTweetsFromFollowing* transaction by maintaining per partition update queues. As in the read-mostly YCSB workload, ADR and multi-master behave similarly, with ADR replicating the most frequently accessed data. However, MorphoSys guarantees single-site reads, unlike ADR, and hence does not suffer from straggler effects due to multi-site scans. As Clay and VoltDB cannot replicate, they suffer even more from these straggler effects.

7.3.4. *Adaptivity*

Next, we present MorphoSys' ability to adapt to workload change in terms of shifting data accesses and load imbalance. Such workload shifts occur, for instance, due to changes in trends on a social network, or shifts in popularity of stocks on the stock exchange [17, 26, 34, 46, 49, 65].

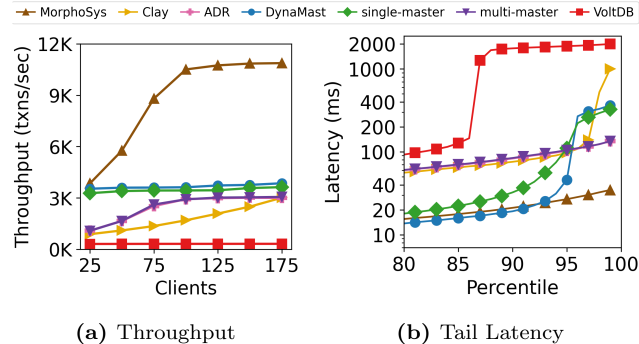
The TPC-C experiment in Figure 5d shows how MorphoSys' adaptivity allows it to cater to different workloads. The figure depicts *New-Order* latency as we increase the percentage of cross-warehouse transactions. When the percentage is zero, MorphoSys' latency is one-quarter of its closest competitor. When the percentage of cross-warehouse transactions reaches about one-third, data locality decreases and MorphoSys' latency approaches that of single-master's, decreasing MorphoSys' relative benefit. Thus, when data locality is low, MorphoSys adapts its physical design by increasingly mastering data at a single site to avoid undoing and redoing physical design changes while distributing load among sites as much as possible.

To highlight MorphoSys' ability to react to workload changes, we experimented with a shifting hotspot [60], a



(a) TPC-C Latency **(b)** TPC-C Tail Latency **(c)** TPC-C Throughput **(d)** New-Order Latency **(e)** SmallBank Tput

Figure 5: Performance results for the TPC-C and SmallBank workloads. In Figure 5c, U's indicate that fully replicated systems (DynaMast, single-master and multi-master) were *unable* to run due to memory constraints.



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

Figure 6: Performance results for the Twitter workload.

phenomenon that frequently occurs in transactional workloads [25, 41]. We induce hotspots with a skewed YCSB workload and shift the center of the skew to a different part of the database every 60 seconds. This challenging workload causes MorphoSys to change its physical design to mitigate the effects of skew and load imbalance. Figure 4c shows the average latency over five minutes and four workload shifts. MorphoSys learns its initial design within the first 30 seconds, at which time MorphoSys reaches its minimum latency that is a reduction of the initial latency by almost 60%, illustrating the benefit of design changes. When a hotspot shifts, MorphoSys' latency increases by at most 20% from its minimum, returning to the minimum latency within 20 seconds on average as MorphoSys quickly splits partitions, adds replicas of the hot partitions and remasters to balance the load and mitigate the hotspot. These results show MorphoSys' effectiveness to rapidly adapt to workload changes due to its learned workload and cost models, and low overhead design changes (Section 7.3.6).

7.3.5. *Scalability*

We scale the number of data sites from 4 to 16 in increments of 4 while also scaling the number of clients (60 per data site), and measure peak throughput using the read-heavy uniform YCSB workload. As shown in Figure 4d, MorphoSys improves throughput by nearly 3× as the number of data sites grows from 4 to 16. MorphoSys achieves this near-linear scalability because its dynamic physical design effectively distributes transaction load among sites, minimizes replication overhead, eliminates distributed transaction coordination, and, as we show next, has low overhead.

7.3.6. *Overhead and Model Accuracy*

To understand MorphoSys' overheads, including planning and executing physical design changes, we evaluated performance using the SmallBank benchmark. Transactions in SmallBank access at most two data items using a uniform data access pattern. Thus, the time spent executing transaction logic is small, making it easier through relative comparison to identify where time goes in the system. Figure 5e shows the maximum throughput for the SmallBank workload. Observe that MorphoSys outperforms its competitors by between 104× and 1.5×, indicating that MorphoSys' dynamism incurs little overhead. The performance of VoltDB is severely limited by distributed update transactions and the single-threaded execution model that blocks non-conflicting transactions belonging to the same thread of execution.

A latency breakdown for both transactions and physical design changes is included in Appendix D.1. MorphoSys spends the plurality of time (43%) executing transaction logic, just 10% of its time in the transaction router, and a mere 3% of the time executing physical design changes. MorphoSys' low overheads result from amortizing the cost of design changes over many transactions and executing changes in parallel when they occur. On average, MorphoSys executes 30 design changes for every 1000 transactions, taking 6 ms to execute per design change.

Recall from Section 5.2 that MorphoSys uses a cost model to predict operation latencies. Figure 4e examines the cost model's accuracy by comparing the actual and predicted execution costs (latencies) of the *split* design change operator. The predicted latency closely tracks the actual latency with a coefficient of determination (R^2) of 0.81. This result indicates that our cost model captures design change costs with high accuracy while being easy to interpret and efficient to train.

8. CONCLUSION

We presented MorphoSys, a distributed database system that automatically modifies its physical design to deliver excellent performance. MorphoSys integrates three core aspects of distributed design: grouping data into partitions, selecting a partition's master site, and locating replicated data. MorphoSys makes comprehensive design decisions using a learned cost model and efficiently executes design changes dynamically using a partition-based concurrency control and update propagation scheme. MorphoSys improves performance by up to 900× over prior approaches while precluding the use of static designs requiring prior workload information. We expect MorphoSys ability to generate and adjust distributed physical designs on-the-fly without prior workload knowledge to pave the way for the development of self-driving distributed database systems.

Acknowledgements: Funding for this project was provided by NSERC, WHJIL, CFI, and ORF.

9. REFERENCES

[1] The Transaction Processing Council. TPC-C Benchmark (Revision 5.11)., 2010.

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

[3] M. Abebe, B. Glasbergen, and K. Daudjee. DynaMast: Adaptive dynamic mastering for replicated systems. In *IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1381–1392. IEEE, 2020.

[4] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 ACM International Conference on Management of Data (SIGMOD)*, pages 583–598. ACM, 2016.

[5] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 541–556. ACM, 2017.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD)*, 1995.

[7] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB Journal - The International Journal on Very Large Data Bases (VLDBJ)*, 23(6):987–1011, 2014.

[8] P. Chairunnanda, K. Daudjee, and M. T. Özsu. Confluxdb: Multi-master replication for partitioned snapshot isolation databases. *PVLDB*, 7(11):948–958, 2014.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 2010 ACM symposium on Cloud Computing (SoCC)*, pages 143–154. ACM, 2010.

[10] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[11] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 666–679, 2019.

[12] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *IEEE 20th International Conference on Data Engineering (ICDE)*, pages 424–435. IEEE, 2004.

[13] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 715–726, 2006.

[14] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD)*, pages 1243–1254, 2013.

[15] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[16] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD)*, pages 299–313. ACM, 2015.

[17] E. Escobedo. The impact of hyped ipo's on the market. Technical report, University of Northern Iowa, 2015.

[18] F. Faerber, A. Kemper, P.-Å. Larson, J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems. *Foundations and Trends® in Databases*, 8(1-2):1–130, 2017.

[19] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[20] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *ACM SIGMOD Record*, 25(2):173–182, 1996.

[21] H. Gupta. Selection of views to materialize in a data warehouse. In *International Conference on Database Theory (ICDT)*, pages 98–112. Springer, 1997.

[22] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017.

[23] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM International Conference on Management of Data (SIGMOD)*. ACM, 2008.

[24] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 2013 Symposium on Cloud Computing (SoCC)*, page 13. ACM, 2013.

[25] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 ACM International Conference on Management of Data (SIGMOD)*, pages 651–665. ACM, 2019.

[26] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (OSDI)*, pages 167–181, 2013.

[27] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363, 1986.

[28] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design continuums and the path toward self-designing key-value stores that know and learn. *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[29] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD)*, pages 603–614, 2010.

[30] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[31] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *IEEE 27th International Conference on Data Engineering (ICDE)*, pages 195–206. IEEE, 2011.

[32] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[33] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4), 2011.

[34] J. Ledlie and M. Seltzer. Distributed, secure load balancing with skew, heterogeneity and churn. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 2, pages 1419–1430. IEEE, 2005.

[35] V. Leis, A. Kemper, and T. Neumann. Scaling htm-supported database transactions to many cores. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(2):297–310, 2015.

[36] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 1659–1674. ACM, 2016.

[37] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. Adaptdb: adaptive partitioning for distributed joins. *PVLDB*, 10(5):589–600, 2017.

[38] Y. Lu, X. Yu, and S. Madden. Star: Scaling transactions through asymmetric replication. *PVLDB*, 12(11):1316–1329, 2019.

[39] S. Madden. From databases to big data. *IEEE Internet Computing*, 16(3):4–6, May 2012.

[40] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching: towards a new global caching architecture. *Computer Networks and ISDN systems*, 30(22-23):2169–2177, 1998.

[41] A. Nazaruk and M. Rauchman. Big data in capital markets. In *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD)*, pages 917–918. ACM, 2013.

[42] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[43] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *PVLDB*, 10(10):1106–1117, 2017.

[44] V. Padhye, G. Rajappan, and A. Tripathi. Transaction management using causal snapshot isolation in partially replicated databases. In *IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS)*, pages 105–114. IEEE, 2014.

[45] V. Padhye and A. Tripathi. Causally coordinated snapshot isolation for geographically replicated data. In *IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*, pages 261–266. IEEE, 2012.

[46] Y. Qian, X. Shao, and J. Liao. Pre-ipo hype by affiliated analysts: Motives and consequences. *Social Science Research Network*, 2019.

[47] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, pages 430–441, New York, NY, USA, 2013. ACM.

[48] T. Rabl and H.-A. Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 315–330. ACM, 2017.

[49] J. Ratkiewicz, S. Fortunato, A. Flammini, F. Menczer, and A. Vespignani. Characterizing and modeling the dynamics of online popularity. *Physical review letters*, 105(15):158701, 2010.

[50] K. Ren, D. Li, and D. J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *PVLDB*, 12(11):1747–1761, 2019.

[51] D. Rolfe. Voltdb and ycsb. `http://www.voltdb.com/blog/2019/10/16/voltdb-and-ycsb/`, 2019. Accessed: 2019-12-16.

[52] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[53] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *PVLDB*, 10(4):445–456, 2016.

[54] A. Shanbhag, A. Jindal, S. Madden, J. Quiane, and A. J. Elmore. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, pages 229–241. ACM, 2017.

[55] A. Sharov, A. Shraer, A. Merchant, and M. Stokely. Take me to your leader!: online optimization of distributed storage configurations. *PVLDB*, 8(12):1490–1501, 2015.

[56] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 385–400. ACM, 2011.

[57] M. Stonebraker, S. Madden, and P. Dubey. Intel "big data" science and technology center vision and execution plan. *ACM SIGMOD Record*, 42(1):44–49, May 2013.

[58] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[59] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 ACM International Conference on Management of Data (SIGMOD)*, pages 205–219. ACM, 2018.

[60] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB*, 8(3):245–256, 2014.

[61] P. Tözün, I. Pandis, R. Johnson, and A. Ailamaki. Scalable and dynamically balanced shared-everything oltp with physiological partitioning. *The VLDB JournalThe International Journal on Very Large Data Bases (VLDBJ)*, 22(2):151–175, 2013.

[62] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 1041–1052. ACM, 2017.

[63] H. Voigt, W. Lehner, and K. Salem. Constrained dynamic physical database design. In *IEEE 36th International Conference on Data Engineering (ICDE) Workshop*, pages 63–70. IEEE, 2008.

[64] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(2):255–314, 1997.

[65] C. Wu, V. Sreekanti, and J. M. Hellerstein. Autoscaling tiered cloud storage in anna. *PVLDB*, 12(6):624–638, 2019.

[66] S. Wu and B. Kemme. Postgres-r (si): Combining replica control with concurrency control based on snapshot isolation. In *IEEE 21st International Conference on Data Engineering (ICDE)*, pages 422–433. IEEE, 2005.

[67] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi. Global-scale placement of transactional data stores. In *Proceedings of the 21st International Conference on Extending Database Technology (EDBT)*, pages 385–396, 2018.

APPENDIX

A. MorphoSys ' CONCURRENCY CONTROL PROTOCOL

We now expand on the description of MorphoSys' concurrency control protocol from the description in Section 4.2. We first explain how MorphoSys modifies transaction timestamps to eliminate transaction inversions and long fork anomalies. Then we present the complete concurrency control algorithm.

To prevent transaction inversion, MorphoSys maintains per client session state called a session timestamp. A client, C, maintains session state $C^S = \{p, \max_T T^C(p)\}$ for all transactions T submitted by the client, which represents the latest version number of all partitions it has accessed. MorphoSys initially sets $T^B(p) = C^S(p)$ for every partition p in T's read and write set. Hence, when T executes, it observes the state at least as up-to-date as the last observed state.

MorphoSys enforces a total commit order that transactions and data sites follow to avoid the long fork anomaly [56]. The long fork anomaly is allowed under *parallel* snapshot isolation, but not under snapshot isolation [6]. Hence, long forks cannot occur in strong session snapshot isolation. The long fork anomaly occurs as a consequence of the write skew anomaly, which is allowed under all forms of snapshot isolation. If transactions T_1 and T_2 experience write skew, then the long fork anomaly occurs if subsequent transactions can possibly observe both states: T_1's effects but not T_2's effects, and T_2's effects but not T_1's effect. If the long fork anomaly is eliminated, the system can only observe one of: T_1's effects but not T_2's effects, or T_2's effects but not T_1's effects. That is, there is a total commit order: either T_1 commits before T_2 or T_2 commits before T_1.

MorphoSys uses the transaction router to enforce the commit order. To enforce a commit order, the transaction router tracks the *committed version numbers* of partitions. When transactions are routed to data sites, these versions numbers are used to enforce a total commit order as part of the begin and commit timestamp information. Hence when two transactions T_1 and T_2 give rise to write skew, as allowed in SI, where say T_2 commits after T_1, T_2 will have a greater commit timestamp than T_1. Hence, subsequent transactions observe one of three database states: (i) the initial state in which no transactions have committed, (ii) T_1's update (but not T_2's), or (iii) both T_1 and T_2's updates. Transactions cannot see a state that includes T_2's update but not T_1's update; thus, long fork is avoided.

MorphoSys' concurrency control algorithm is presented in Algorithm 1. MorphoSys initializes a transaction's begin and commit timestamp to the session timestamp to ensure the transaction observes state at least as up to date as the latest previously observed state from the same session (Line 3). To ensure transactions respect a total commit order, the transaction router initializes the begin timestamps to the *commit version number* (Line 6). At the data site, the transaction locks the partitions in the write set (Line 12) to ensure mutual exclusion of updates. As there may have been in-flight updates to the partition, the data site updates the begin timestamp for partitions in the write set (Line 14). Then, the data site ensure the transaction observes a consistent snapshot by following the *consistent read rule* (Lines 19 and 20). Once the transaction has a consistent begin timestamp, MorphoSys constructs the commit timestamp (Lines 24 to 26) by copying the begin timestamp for read partitions, and incrementing the begin timestamp for write partitions. Once begin and commit timestamps are assigned, a transaction can execute its logic by writing new versioned data items (Line 28) and reading the data items belonging to the snapshot determined by the begin timestamp (Line 29). When the transaction commits, the data site builds the new *depends* relationship following the *dependency recording rule* (Lines 31 to 35), and records this dependency information (Line 37). The data site then makes any updates visible by updating the partition version (Line 38). Finally, the transaction router, updates the commit number (Line 45) followed by the session timestamp (Line 46) so that subsequent transactions abide by the commit order and avoid the long fork anomaly, and within a session clients avoid transaction inversions.

B. CORRECTNESS OF STRONG SESSION SNAPSHOT ISOLATION

We now prove that MorphoSys preserves strong-session snapshot-isolation (SSSI). We first prove that MorphoSys preserves snapshot-isolation (SI) without physical design change operators (Appendix B.1). Then we consider the effect that physical design change operators have on SI (Appendix B.2). Finally, we put these two aspects of the proof together to prove MorphoSys provides that SSSI (Appendix B.3).

B.1. Snapshot Isolation

We now prove that MorphoSys provides SI in the absence of physical design changes.

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

Proof. Let T_1^B be transaction T_1's begin timestamp and T_2^C be transaction T_2's commit timestamp. If $T_2^C \leq T_1^B$ then there exists a partition p such that $T_2^C(p) \leq T_1^B(p)$. Suppose that T_2 updated a data item d in partition p, then as stated in Section 4.2, T_2 creates a new version of d assigned value $T_2^C(p)$. Since there are no physical design changes, then when T_1 performs a read operation on data item d, it occurs in partition p. As described in Section 4.2, when T_1 performs a read operation on d, in partition p, T_1 reads the largest version of d, denoted as $v(d)$, such that $v(d) \leq T_1^B(p)$ holds. Given that $T_2^C(p) \leq T_1^B(p)$, and T_2 performed an update, then there must exist a $v(d)$ such that $v(d) = T_2^C(p)$. Thus, when T_1 performs its read $T_2^C(p) \leq v(d)$ must hold. Hence, T_1 must observe T_2's update, or some later update to the data item. □

Lemma 2. *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. We say that two transactions T_1 and T_2 have overlapping begin and commit timestamps, if there exists a partition p that was in both of T_1 and T_2's read or write sets, and the begin and commit version numbers for p (that is $T_1^B(p), T_1^C(p), T_2^B(p), T_2^C(p)$) overlap. Recall, from Section 4.2 that for read-only transactions T, $T^B = T^C$. Additionally, if a transaction T updates a partition p, then $T^B(p) + 1 = T^C(p)$, as an update transaction acquires a mutually exclusive partition lock. Using this information we now prove, by way of contradiction, that the lemma holds.

Assume both transactions are updates and update the same data item, otherwise the lemma is trivially false. Additionally, as no physical design changes occur, then we

Algorithm 1 MorphoSys Concurrency Control Protocol

Input: Transaction T's sorted write set $\{p_w\}$, read set $\{p_r\}$, and session timestamp, C^S
Output: The updated session timestamp

```
1: // Begin
2: // At the transaction router
3: T^B = C^S
4: {p} = sort {p_w} ∪ {p_r}
5: for p ∈ {p} do
6:    read lock p metadata
7:    T^B(p) = p.commit_version_number
8: unlock {p} partition metadata
9: // At the data site
10: for p ∈ {p} do
11:    wait for p's version to reach T^B(p)
12:    if p ∈ {p_w} then
13:       write lock p
14:       T^B(p) = p's version number
15: partitions_to_check = {p}
16: for p_i ∈ partitions_to_check do
17:    remove p_i from partitions_to_check
18:    for p_j ∈ {p} do
19:       if T^B(p_i) < depends(p_j, T^B(p_j), p_i) then
20:          T^B(p_i) = depends(p_j, T^B(p_j), p_i)
21:          T^C(p_i) = T^B(p_i)
22:          wait for p's version to reach T^B(p)
23:          insert p_i into partitions_to_check
24: T^C = T^B
25: for p ∈ {p_w} do
26:    T^C(p) = T^B(p) + 1
27: // Transaction Logic
28: Write versioned data items in partition p_w with version
    T^C(p_w)
29: Read the largest versioned data items d in partition p_r,
    such that v(d) ≤ T^B(p_r)
30: // Commit
31: loc_depends = T^C
32: for p_i ∈ {p} do
33:    for p_j ∈ depends(p_i, T^B(p_i)) do
34:       if depends(p_i, T^B(p_i), p_j) > loc_depends(p_j) then
35:          loc_depends(p_j) = depends(p_i, T^B(p_i), p_j)
36: for p ∈ {p_w} do
37:    depends(p_i, T^C(p_i)) = loc_depends
38:    set p's version to T^C(p)
39:    unlock p
40: // At the transaction router
41: for p ∈ {p} do
42:    if p ∈ {p_w} then
43:       wait for p.commit_version_number to reach T^B(p)
44:       write lock p metadata
45:       p.commit_version_number = T^C(p)
46:    C^S(p) = max(T^C(p), C^S(p))
47: unlock {p_w} partition metadata
48: return C^S
```

have that d is in the same partition p. Then, we have that $T^B(p)+1 = T^C(p)$ for both T_1 and T_2. In the first case, we have that $T_1^B(p) \le T_2^B(p) < T_2^C(p) \le T_1^C(p)$. In the second case, we have that $T_1^B(p) \le T_2^B(p) < T_1^C(p) \le T_2^C(p)$.

Observe that in both cases, for both of these inequalities to be true, and for $T^B(p)+1 = T^C(p)$ to hold for both T_1 and T_2, then we must have that $T_1^B(p) = T_2^B(p)$ and $T_1^C(p) = T_2^C(p)$. However, as stated in Section 4.2 transactions acquire partition locks before reading the partition version number and constructing T_B and release the locks after setting T_C and updating the partition version number. As we assume no physical design changes, then T_1 and T_2 must execute at the same site. Thus one of T_1 or T_2 will acquire the partition lock first, so $T_1^B(p) \ne T_2^B(p)$ must hold. However this inequality is a contradiction with $T_1^B(p) = T_2^B(p)$ □

Lemmas 1 and 2 satisfy the requirements of SI, under the assumption of the correctness of the begin and commit timestamps. We now prove that our begin and commit timestamps produce a consistent snapshot of data items across partitions.

LEMMA 3. *MorphoSys' transactions begin and commit timestamps produce snapshot consistent state. That is, if T_2 updates data items in partitions p_1 and p_2, and T_1 reads data items in partitions p_1 and p_2, then either $T_2^C \le T_1^B$, and hence T_1 will observe T_2's updates, or $T_2^C > T_1^B$ and T_1 will not observe T_2's updates.*

PROOF. Observe that for this proof to hold, either $T_2^C(p) \le T_1^B(p)$ must hold for both p_1 and p_2, in which case it follows from Lemma 1 that T_1 observes T_2's updates, or, $T_2^C(p) > T_1^B(p)$ must hold for both p_1 and p_2, in which case T_1 will not observe T_2's updates.

Recall, from Section 4.2, that if T_2 updates partitions p_1 and p_2, then the update results in a *depends* relationship between p_1 and p_2. That is, if $T_2^C(p_1)$ and $T_2^C(p_2)$ are the versions of p_1 and p_2 updated by T_2, then MorphoSys records $T_2^C(p_2) = depends(p_1, T_2^C(p_1), p_2)$ and $T_2^C(p_1) = depends(p_2, T_2^C(p_2), p_1)$. It follows that there are three cases in the construction of T_1's begin timestamp.

Case 1: Suppose that T_1 initially selects $T_1^B(p)$ such that $T_2^C(p) \le T_1^B(p)$ for both p_1 and p_2. This case trivially results in $T_2^C \le T_1^B$, hence T_1 observes both of T_2's updates.

Case 2: Suppose that T_1 initially selects $T_1^B(p)$ such that $T_2^C(p) > T_1^B(p)$ for both p_1 and p_2. This case trivially results in $T_2^C > T_1^B$, hence T_1 does not observe either of T_2's updates.

Case 3: Suppose, without loss of generality, that T_1 initially selects $T_1^B(p_1)$ and $T_1^B(p_2)$, such that $T_2^C(p_1) > T_1^B(p_1)$ and $T_2^C(p_2) \le T_1^B(p_2)$ holds. Following the consistent read rules, T_1 updates $T_1^B(p_i)$ such that $T_1^B(p_i) \ge depends(p_j, T_1^B(p_j), p_j)$ for all p_i and p_j in its read set. Setting p_i as p_1 and p_j as p_2, we get that $T_1^B(p_1) \ge depends(p_2, T_1^B(p_2), p_1)$. As $T_1^B(p_2) \ge T_2^C(p_2)$, and the depends relationship always uses the max operator (Equation 1), then we have that $depends(p_2, T_1^B(p_2), p_1) \ge depends(p_2, T_2^C(p_2), p_1)$. Recall that $depends(p_2, T_2^C(p_2), p_1) = T_2^C(p_1)$, so combining all equations we get that $T_1^B(p_1) \ge T_2^C(p_1)$. Thus, we have that $T_1^B(p) \ge T_2^C(p)$ for both p_1 and p_2, a contradiction as we selected $T_2^C(p_1) > T_1^B(p_1)$. Hence, we have that $T_2^C \le T_1^B$, and T_1 observes both of T_2's updates.

□

LEMMA 4. *There exists a total commit order between transactions T_1 and T_2.*

PROOF. We define the total commit order for transactions T_1 and T_2, by considering four cases.

Case 1: For all p, $T_1^C(p) < T_2^C(p)$, thus T_1 commits before T_2.

Case 2: For all p, $T_1^C(p) = T_2^C(p)$, thus T_1 and T_2 have the same commit time, and hence are placed in the same position in the total order. Note that in this case, at most one of T_1 and T_2 are update transactions.

Case 3: There exists p_1 and p_2, such that $T_1^C(p_1) < T_2^C(p_1)$ and $T_1^C(p_2) = T_2^C(p_2)$. Then T_1 commits before T_2.

Case 4: There exists p_1 and p_2, such that $T_1^C(p_1) < T_2^C(p_1)$ and $T_1^C(p_2) > T_2^C(p_2)$. Such a scenario arises from write skew, and as outlined in Appendix A the transaction router determines the commit order. Transactions reading partitions p_1 and p_2 can observe one of four states, neither of T_1 or T_2's updates, both of T_1 and T_2's updates, T_1's updates, but not T_2', or T_2's updates both not T_1's. Recall that the transaction router uses locking (Algorithm 1) to ensure that all transactions can observe the same three states, by eliminating one of: T_1's updates but not T_2's, or T_2's update but not T_1's. This elimination determines the commit order: if T_1's update can be observed but not T_2, then T_1 commits before T_2. Otherwise, T_2, commits before T_1.

We have outlined all four possible cases given transaction commit timestamps and defined a commit order; hence there exists a commit order. □

Together, Lemmas 1, 2, 3 and 4 prove that MorphoSys satisfies the requirements of SI, when there are no physical design changes.

B.2. Physical Design Changes and Snapshot Isolation

Appendix B.1 proved that MorphoSys provides SI when there are no physical design change operators. We now prove that MorphoSys provides SI in the presence of these physical design change operators. We make four critical observations about the proofs in Appendix B.1, and the changes that arise in the presence of physical design changes.

First, removing a replica does not change the correctness of the proofs, as a replica partition is no longer present at a data site.

Second, adding a replica does not change the correctness of the proofs. Recall from Section 4.4.1, that MorphoSys installs a snapshot of the replicated partition that includes the partition version number, the *depends* relationship, and versioned data items. Hence, as replicas execute only read-only transactions, all of the state necessary to ensure the correctness of Lemmas 1 and 3 exist at the newly created replica, or in the case of Lemma 4, at the transaction router.

Third, to prove Lemma 2, we assumed that updates to partitions occurred on the same master data site. This assumption does not hold if the system remasters a partition. Hence, we need only prove that a partition cannot be updated at two data sites concurrently, as a consequence of remastering. Recall, from Section 4.4.3, that remastering occurs transactionally, and hence the *remaster* operator acquires the partition lock. Additionally, after the old master site releases the mastership of a partition, it can no longer service update transactions to the partition. Furthermore, the new master site does not become the new master until it applies the propagated update releasing the mastership from the old master and all previous updates to the partition. Consequently, no update transactions to the partition occur while it is being remastered. Thus, given two transactions that update the same data item (and thus partition), either the updates both occur at the same site, proven correct in Lemma 2, or one transaction (T_1) updates the data item at the old master, and the other (T_2) at the new master. However, because the new master blocks updates until all previous updates to the partition are applied, we have that $T_1^B(p) < T_1^C(p) \leq T_2^B(p) < T_2^C(p)$, hence the updates do not overlap.

Fourth, in Lemmas 1 and 2 we assumed that data items always belong to the same partition. This assumption does not hold if the system splits partitions apart, or merges them together. Recall from Section 4.4.2, that MorphoSys performs these operations while holding the partition locks, consequently, after the *split* or *merge* operator completes subsequent transactions execute on the newly created partitions. Additionally, these physical design change operators assign the newly created partition's version numbers as the maximum of the original partitions' version number and induce a *depends* relationship among the partitions. Thus, as shown in the proof in Lemma 3, subsequent transactions generate transaction begin timestamps that observe snapshot consistent state. Consequently, the dependency relationship determines the transactions total commit order, as we can place the transaction either occur before or after the *split* or *merge* operation.

Given our fourth observation, we must prove that updates to the same data item do not occur concurrently, in the presence of *split* or *merge* operations. Similar to the remastering case, updates to the same data item either occur in the same partition, or, before and after a *split* or *merge* operation, and thus in different partition. In the former case, Lemma 2 holds. Considering the latter case, without loss of generality, suppose T_1 updates data item d in partition p, p is *split* into p_L and p_H, and T_2 updates d that is now contained in p_L. By definition, we have that $T_1^B(p) < T_1^C(p)$ and $T_2^B(p_L) < T_2^C(p_L)$. By construction of p_L, we also have that the intial partition version of p_L, $v(p_L) \geq T_1^C(p)$, and T, hence $T_1^C(p) \leq T_2^B(p_L) < T_2^C(p_L)$. Additionally, as p does not exist for T_2, but is stored as part of the *depends* relationship, we have that $T_1^C(p) \leq \textit{depends}(p_L, T_1^B(p_L), p) = T_2^B(p)$. Hence, we have $T_1^B(p) < T_1^C(p) \leq T_2^B(p) = T_2^C(p)$ and thus the updates do not occur concurrently. A similar argument follows for the *merge* operator.

Given our four observations and associated proofs, it is clear that MorphoSys provides SI in the presence of physical design changes.

B.3. Enforcing Strong Session Snapshot Isolation

We now prove that MorphoSys provides SSSI by proving the session requirement of SSSI.

THEOREM 1. *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.*

PROOF. Recall from Section 4.2.1 and Appendix A, that MorphoSys tracks a session timestamp C^S, composed of the maximum observed $T^C(p)$ for all transactions T in the same session, and accessed partitions p. Thus $T_1^C \leq C^S$. Furthermore, MorphoSys uses this session timestamp as the initial transaction begin timestamp, before updating it based on observed partition version numbers, blocking if necessary. Thus $C^S(p) \leq T_2^B(p)$. Combining the two inequalities, we have that $T_1^C \leq T_2^B$, as required. □

Theorem 1, together with Lemma 1, prove that if 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 observes any state observed or created by T_1. Given that MorphoSys provides SI, in addition to the session requirements of SSSI, then MorphoSys guarantees SSSI.

C. FORMAL DEFINITIONS

In Section 3, we outlined the definition of a partition, the requirements for transactions, and the five physical design change operators. We now formalize these definitions.

A partition of data p contains all data items with *row id*'s that fall in the inclusive range $(start(p), end(p))$. The partition p has its master copy located at site $S_i = master(p)$, and replicas placed at a (possibly empty) set of sites $\{S_j | j \neq i\} = replicas(p)$.

A transaction must specify its read and write set. We formally define the data items in the read set as $\{d_r\}$, and the data items in the write set as $\{d_w\}$. Given these data items, the transaction router identifies the set of partitions $\{p_r\}$ and $\{p_w\}$ as the read and write set, respectively. To identify a partition given a data item d, the transaction router finds the partition p such that $start(p) \leq d \leq end(p)$ holds. A transaction can execute at a site S if for all p_w in the write set $master(p_w) = S$ holds, and if for all p_r in the read set $S \in \{master(p_r)\} \cup replicas(p_r)$ holds

The definition of our five physical design change operators is as follows.

split: given a partition p, and k such that $start(p) < k \leq end(p)$, then $split(p, k)$, creates new partitions p_L and p_H and removes p. Partitions p_L and p_H are defined such that $p_L = (start(p), k-1)$ and $p_H = (k, end(p))$, $master(p_L) = master(p_H) = master(p)$, and $replicas(p_L) = replicas(p_H) = replicas(p)$.

merge: given partitions p_L, and p_H then $merge(p_L, p_H)$, creates new partition p, and removes p_L and p_H. For *merge* to succeed, p_L and p_H must satisfy the following $end(p_L) = start(p_H) - 1$, $master(p_L) = master(p_H)$, and $replicas(p_L) = replicas(p_H)$. Partition p is defined such that $p = (start(p_L), end(p_H))$, $master(p) = master(p_L) = master(p_H)$, and we set $replicas(p) = replicas(p_L) = replicas(p_H)$.

add replica: given partition p, with $replicas(p) = R$, and S_j, such that $S_j \neq master(p)$ and $S_j \notin R$ then $add_replica(p, S_j)$ sets $replicas(p) = R \cup \{S_j\}$.

remove replica: given partition p, with $replicas(p) = R$, and S_j, such that $S_j \in R$ then $remove_replica(p, S_j)$ sets $replicas(p) = R \setminus \{S_j\}$.

remaster: given partition p, with $replicas(p) = R$, and S_j, such that $S_j \in R$, and $master(p) = S_i$, then $remaster(p, S_j)$, sets $master(p) = S_j$, and sets $replicas(p) = R \setminus \{S_j\} \cup \{S_i\}$.

D. ADDITIONAL EXPERIMENTAL RESULTS

D.1. System Overheads

Recall from Section 7.3.6 that we used SmallBank with its short transactions to assess the overheads of MorphoSys. Table 2 breaks down SmallBank transaction latency in MorphoSys. Observe that the system spends the plurality of time (43%) executing transaction logic. At data sites, just 25% of overall transaction latency is spent on MorphoSys' concurrency control, including waiting for any necessary updates, locking, and recording dependencies during commit. Given the small transaction footprint, this translates to low overhead as the latency is comparable to the amount of time that transactions spend in the network. This low latency is a consequence of our partition-based concurrency control and update propagation scheme. Finally, transactions spent just 10% of their time at the transaction router, including an average of just 3.3% of time executing physical design changes. This small overhead results from MorphoSys amortizing the cost of design changes over many transactions, and executing design changes in parallel when they do occur.

Table 3 shows the relative frequency and average latency of each of the physical design operators. On average, MorphoSys executes a physical design change operator 30 times for every 1000 transactions, taking just over 6 ms to execute. The most expensive physical design change operators require physical copying of data, as in the case of adding a replica of a partition or waiting for all updates to arrive at the soon to be designated new master.

Table 2: Transaction latency breakdown within MorphoSys.

Operation	Avg. Latency	Percent of Txn. Time
Locating Partitions	27 ± 1.0 μs	2.1%
Plan Generation	49 ± 9.2 μs	3.8%
Workload & Cost Model	13 ± 6.8 μs	1.0 %
Design Change	42 ± 2.9 μs	3.3 %
Network & Queuing	252 ± 51 μs	19.8%
Locking	114 ± 39 μs	8.9%
Waiting for Updates	65 ± 5.9 μs	5.1%
Transaction Logic	550 ± 97 μs	43.0%
Commiting	158 ± 8.4 μs	12.4%
Total	1270 ± 310 μs	100%

Table 3: Design change operator frequency and latency.

Operator	**Frequency** (per 1000 Txns.)	Avg. Latency
split	3.9 ± 0.3	3.7 ± 0.67 ms
merge	0.15 ± 0.06	8.3 ± 1.7ms
remaster	14.4 ± 0.5	33.3 ± 2.3 ms
add_replica	12.1 ± 0.4	40.4 ± 0.4 ms
remove_replica	0.12 ± 0.01	0.79 ± 0.04 ms
Total	30.7 ± 1.2	6.1 ± 11.7 ms

D.2. Different Intial Physical Designs

In our experiments in Section 7 we provided ADR, VoltDB, and multi-master with offline *a priori* knowledge of the workload, when initializing the physical design, by using Schism [10] (Section 7.1). We advantaged DynaMast and Clay by balancing the number of partitions mastered at each site. However, for MorphoSys, the initial physical design was completely random. To examine the effect of these different initial physical designs on performance, we ran experiments with our skewed read-mostly YCSB workload and measured peak throughput under two initial physical designs: Schism and a randomized design. Additionally, for the initial randomized design, we measured the time it took for the system to converge to within the confidence interval of its peak throughput. These results are presented in Table 4.

As shown in Table 4, MorphoSys has the highest throughput under both initial physical designs. Furthermore, with a randomized initial physical design, MorphoSys converges quickly (in 23 seconds) to within 2% of its throughput when initialized with *a priori* knowledge. Without a priori workload knowledge, ADR and Clay reach only about two-thirds of their peak throughput. Additionally, ADR and Clay take nearly 3× as long as MorphoSys to converge to their peak

Table 4: YCSB Read-Mostly Skew Throughput, using two different initial physical designs: Schism [10] and random. We additionally show the time the system takes to converge to peak throughput for each initial physical design. Single-master uses the same physical design for both Schism and random, as it always places master copies of partitions on a single node.

System	**Schism Avg. Throughput (txn/sec)**	**Random Avg. Throughput (txn/sec)**	**Schism Time to Peak Throughput**	**Random Time to Peak Throughput**
MorphoSys	**309 ± 11k**	**302 ± 11k**	6 s	23 s
Clay	61.7 ± 2.9k	41.6 ± 1.5k	3 s	58 s
ADR	29.7 ± 3.0k	18.5 ± 1.3k	3 s	82 s
DynaMast	210 ± 3.5k	198 ± 2.2k	5 s	122 s
single-master	170 ± 1.5k	168 ± 3.5k	2 s	2 s
multi-master	46.1 ± 3.4k	21.8 ± 1.1k	4 s	9 s
VoltDB	3.21 ± 0.02k	1.98 ± 0.02k	7 s	6 s

Table 5: Effect of record size on physical design operations latency.

Metric **Record Size (Bytes)**	*split* Avg. Latency	*merge* Avg. Latency	*add_replica* Avg. Latency	*remove_replica* Avg. Latency	*remaster* Avg. Latency
1	6.95 ± 1.9 ms	19.1 ± 2.0 ms	56.5 ± 1.2 ms	3.07 ± 1.3 ms	50.7 ± 9.1 ms
10	7.05 ± 1.5 ms	19.8 ± 2.4 ms	58.7 ± 2.0 ms	3.51 ± 2.3 ms	52.9 ± 7.1 ms
100	7.29 ± 2.1 ms	20.3 ± 4.0 ms	62.2 ± 2.3 ms	4.65 ± 2.7 ms	61.4 ± 8.6 ms
1000	7.31 ± 1.5 ms	21.3 ± 3.6 ms	69.1 ± 1.3 ms	4.96 ± 1.4 ms	64.9 ± 10.5 ms
Relative Change	5.17%	11.5%	22.3%	61.5%	28.0%

throughput, a consequence of performing design changes periodically. MorphoSys converges to its peak throughput faster than these systems because it uses every transaction as an opportunity to make physical design changes, in contrast to the periodic design changes made by ADR and Clay. DynaMast takes 5× longer than MorphoSys to reach peak throughput when initialized with a random physical design. Finally, as stated in Section 7.3.1, by taking a holistic approach to distributed physical design and considering all of dynamic partitioning, replication and mastering, MorphoSys outperforms its competitors that consider only one of these aspects of physical design.

Single-master, multi-master and VoltDB, all reach their peak throughput in a short period of time as they do not change their physical designs in response to a workload. Note that the throughput of single-master is nearly identical in the two experiments, as the single-master architecture imposes a single physical design: a single site masters every partition, and each partition is replicated at all other sites. By contrast, the throughput of multi-master and VoltDB drop by 50% and 40%, respectively, when a random physical design is used compared to Schism's physical design. Throughput degrades for these systems as the frequency of distributed transactions increases in the randomized initial physical design, compared to Schism's physical design that aims to place partitions to minimize distributed transactions.

D.3. Effects of Record Size

To understand the effect that record size has on both system performance and the physical design change operators, we experimented with our read-mostly, skewed YCSB workload. We use YCSB for this experiment, as it allows us to easily control the size of each data item. Table 5 shows the results of this experiment as we vary the record sizes by a hefty 3 orders of magnitude, i.e., from 1 byte to 1000 bytes.

As the record sizes increase, the average latency of splitting and merging partitions remains mostly the same, with only 5% and 11% increases, respectively. The split and merge operations are not dependent on record size, because they operate on metadata of partitions; they do not need to read or write any of the records.

Adding a replica requires physically reading record data, and sending it over the network. Thus, as the data size increases, so too does the time taken to add a replica. Similarly, removing a replica partition must free the memory associated with the replica record, which results in increased latency for larger records. Finally, remastering may need to wait for the system to propagate and apply updates that take longer for larger records. Thus the latency of remastering increases as the record size increases.